UNIVERSITY OF CALGARY

Universally Quantified Type Inference

by

Saina Daneshmandjahromi

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

NOVEMBER, 2025

Abstract

Haskell's type system is based on the Hindley–Milner type system, which supports only rank-1 types. As functional languages have evolved, higher-rank types have been recognized as being useful for expressing more general abstract functions. Peyton Jones et al. introduced a type inference algorithm for universally quantified types in Haskell. Full type inference for higher-ranked types is undecidable and consequently it often requires explicit type annotations to facilitate type inference. In this work, we propose a new algorithm for inferring these types. Our approach is demonstrated using a minimal programming language syntax and provides a modular method for inferring higher-ranked types. The inference process is organized into two distinct steps: generating type equations and then solving them. By separating the collection of equations from their resolution, this thesis gives a more modular inference algorithm, which is slightly more general than the bidirectional system introduced by Peyton Jones.

Acknowledgments

I sincerely thank my supervisor, Dr. Robin Cockett, for his guidance and support throughout my research. I also thank my committee members, Dr. Philip Fong and Dr. Richard Zach, for their time, valuable feedback, and insights, and my lab mates for proofreading my thesis and offering helpful suggestions. I am grateful to my friends for making this journey enjoyable. Their patience, encouragement, and advice have meant a lot to me.

Finally, with all my heart, I thank my parents, Saleh and Rozi, and my siblings, Sasan and Sanaz, for their endless love, sacrifices, and support. They have been my greatest source of strength and motivation, carrying me through every challenge to the completion of this journey.

To Rozi

and Saleh ♡

Table of Contents

Chapter 1

Introduction

Haskell's type system is derived from the Hindley-Milner type system [1, 2]. In the original Hindley–Milner system, type variables may be universally quantified only at the outermost level: these are called rank-1 types (see Section 1.4) [3]. In these types, universal quantifiers cannot appear within function arguments or return types. While rank-1 types suffice for most practical programming purposes, arbitrary ranked types allow the expression of more sophisticated abstractions.

In roughly 2007, Haskell's type system was extended to support universally quantified types within function arguments and return types [3]. However, the algorithm used for Haskell's typing, which is based on a bidirectional system, is not straightforward. The aim of this thesis is to propose a simpler and more comprehensible type inference system. The inference algorithm we propose proceeds in two distinct stages: generating type equations and then solving them. We claim that breaking the process into two steps results in a more transparent algorithm.

Another aspect of the algorithm developed in this thesis is that it has been developed with proofs, based on the type system of System F. The algorithm we present here is slightly stronger than Haskell's: it can type check everything Haskell can, and it also handles some minor cases that Haskell cannot.
The basic ideas for supporting universally quantified types came from System F. However, to make System F compatible with practical programming languages, a modified version has been introduced, which we call system S. Type checking and inference for this modified system is known to be undecidable [4]. As a result, a complete type inference algorithm for higher-rank types cannot be achieved, and, indeed, the type inference system for Haskell often needs hints in the form of annotations [5] in order

to complete type inference tasks.

Several use cases for higher rank types which cannot be typed without type annotations are presented in [3]. One simple example for which Haskell cannot infer the type is:

```
g x = (x [True, False], x "abc")
```

However, if type annotations are used, the function can be typed, by Haskell:

```
g :: (forall a. [a] -> [a]) -> ([Bool], [Char])
g x = (x [True, False], x "abc")
```

Higher rank types are also useful in the implementation of folds and unfolds, where recursion schemes involve functions that must operate uniformly over polymorphic structures (see [6], [7]).

The function foldr "(fold right)", which is defined in Haskell, is a higher-order function that recursively processes a list from the right. For finite lists, foldr guarantees termination, provided its arguments terminate. Foldr takes a function, an initial accumulator value, and a list. For each element, it applies the function to the element and the result of folding the rest of the list.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

The function append, which takes two lists and returns a single list containing all the elements of the first list followed by all the elements of the second list, can be defined using foldr as follows:

```
append :: [a] -> [a] -> [a]
append xs ys = foldr (:) ys xs
-- append [1,2,3] [4,5] = [1,2,3,4,5]
```

The term for the fold that appends two lists can be represented by a circular proof box.

$$\dfrac{\dfrac{\forall X \qquad f : X \to L(A) \to L(A)}{u : 1 \to L(A) \to L(A) \qquad \dfrac{f : X \to L(A) \to L(A) \quad (:) : A \to L(A) \to L(A)}{f\ (:) : A \times X \to L(A) \to L(A)}}{1 + A \times X \to L(A) \to L(A)}}{\text{append} : L(A) \to L(A) \to L(A)}$$

This equivalently says that

```
append_box :: forall X. ( X -> L(A) -> L(A)) -> ((1+ A x X) -> L(A) -> L(A))
append_box = \f x y. case x of nil -> y
                              a:as -> a: (f as y)
```

with this formulation, one can define the usual append function as

```
append =
append_box append = \x y. case x of nil -> y
                                    a:as -> a: (append as y)
```

which uses recursion and is guaranteed to terminate provided the list `x` is finite.

1.1 Thesis outline

A brief summary of each chapter of the thesis is provided below to give an overview of the thesis.

In Chapter 2, we introduce the simply typed λ-calculus and then discuss the Hindley–Milner type system, which extends the simply typed λ-calculus with restricted forms of universal quantification. Our goal is to design a system that is more expressive than Hindley–Milner and capable of supporting higher-ranked types. We compare the Hindley–Milner type system with the CaMPL type inference system. Although the expressive power of the two inference systems are the same, our approach to type inference follows CaMPL's. CaMPL separates the tasks into two steps: first collecting type equations that the term implies, and then solving the type equations to obtain a type of the term.

To provide a foundation for generating equations, the System F type checking rules are presented in Chapter 3. This serves as the basis for type checking, with universal quantificatons. We then adapt system F to obtain System S, a system which is more suitable for typing programs. In designing a higher-ranked type system for System S, we observe that some terms can be assigned more than one type. To capture this, we define a relation called subsumption (denoted $\preceq$). A type T subsumes a type S, written $T \preceq S$, if whenever a term t can be typed with type T, it can also be typed with type S. These subsumption relations establish the foundation for specification and generalization in our type inference system. We show that subsumptions can be used to augment system S with coercions.

In Chapter 4, we generalize subsumptions to define type equations and show how to manipulate these type equations. We then define the rules for collecting such equations from a given term, and prove that if, a term can be type-checked in System F with a type, then that type is a solution to the generated equations for that term.

In Chapter 5, we discuss the notion of a candidate solution to type equations and define the verification rules to determine whether a candidate solution is valid.

Finally, in Chapter 6, we introduce a small-step type inference system, which applies small simplifications to the equations at each step, and prove that this small-step inference is sound. We then use this system to define a big-step inference system, which simplifies the type equations using multiple small steps. Since the big-step system is still non-deterministic, we finally refine that into a deterministic algorithm for type inference.

1.2 Contributions

This thesis makes several contributions to the study and implementation of type inference for higher-rank types. First, it introduces a novel algorithm capable of inferring types in systems where universal quantifiers may appear nested within function arguments or return types. The algorithm separates the inference process into an equation generation phase and an equation solving phase, providing a sound, deterministic, and implementable approach despite the undecidability of the general problem. Second, the thesis presents a formal development of the underlying type system, System S, based on System F, incorporating a subsumption relation $T \preceq S$ to handle terms with multiple valid types and formalizing these as coercions. Third, it provides rules for generating type equations from terms and

verifying candidate solutions, ensuring that any type derivable in System F corresponds to a solution of the generated equations. Finally, the proposed algorithm is at least as powerful as Haskell's type system, being able to type check all terms that Haskell can, as well as additional minor cases, while simplifying the bidirectional inference process by generating equations which are subsequently solved. Together, these contributions advance both the theoretical understanding and practical implementation of higher-rank type inference.

1.3 Related work

There are several papers on quantified type systems that support higher-ranked types [8, 3, 9, 5]. All of these systems introduce a notion of subsumption, which allows a term to be assigned more than one type. Most of them define inference rules to determine the type of a term. However, interestingly, Zhao et al. [5] use a different approach: they express type inference as a sequence of reductions $\Gamma \longrightarrow \Gamma'$. In their system, Γ is handled like a stack, where each step removes the first judgment for processing. Most of these systems generate the type of a term inductively as they traverse it. However, the system presented by Serrano et al.[9] first generates type constraints and then solves them, a method influenced by ML type inference [10] and originally introduced by Fuh and Mishra in 1980 [11]. The difference between Serrano et al.'s system [9] and ours is that, unlike our system, they employ a bidirectional type inference approach, similar to that of Peyton Jones et al. The bidirectional technique is described in detail by Dunfield and Krishnaswami [12] (see Section 1.3.1).

In the remainder of this section, we compare our algorithm with Peyton Jones's approach to the design of the Haskell type inference system, emphasizing the techniques they employ and how they differ from those used in our method.

1.3.1 Bidirectional system

A key aspect of the type inference system of Haskell is its use of a bidirectional type inference. Bidirectional type inference combines top-down (checking) and bottom-up (synthesis) inference modes [12]. In this system, if the type of an expression is already known, the algorithm switches to type checking mode to verify that the expression matches the expected type; if the type is not known, it switches to

synthesis mode to infer the type from the expression itself. While this approach is effective, it makes type inference more complex to understand due to the switching between inference modes.
In the algorithm we develop bidirectional inference system is absorbed into a uniform process of generating and solving equations.

1.3.2 Implicit quantification

Haskell uses implicit quantification [13], meaning that even if a user does not explicitly intend to universally quantify type variables, Haskell enforces this automatically. This means that if a user writes the following function:

Example 1.3: using the identity function in Haskell

```
id :: a -> a
id x = x
f = (id 1, id 'c')
```

This enforces the type of `id` to be $\forall a.a \rightarrow a$, making the function `f` typeable. While this implicit approach simplifies type annotations, it might not align with the user's intent. In this work, we employ an alternative approach in which $\forall$ quantifiers must be made explicit, respecting the user's specifications regarding type variables that are not universally quantified.
Therefore, Example 1.3.2 cannot be typed in our system unless the user defines `id` with the type $\forall a.a \rightarrow a$, as follows:

```
id :: forall a. a -> a
id x = x
f = (id 1, id 'c')
```

1.3.3 Cases beyond Haskell's type inference

There are cases where Haskell cannot infer the type of a term, but our algorithm can. The following example demonstrates this:

```
\x -> (x :: Int -> Int, x :: forall A. A -> A)
```

where the inferred type for x will be $\forall A.A \to A$.

There are cases, however, where our type inference algorithm, like Haskell's, is unable to infer the type of the term.

Consider the following example:

```
\x -> (x :: Int -> Int, x :: Bool -> Bool, x)
```

If the context provided $x : \forall A.A \to A$, this expression would be typeable, since A could be instantiated as both Int and Bool.

1.4 Rank of a type

Types can be classified based on their ranks, which indicates the depth at which universal quantifiers appear within a type. The rank $r(T)$ of a type T is defined recursively as follows [13]:

(1) Base Case: If A is a type variable or a primitive type (such as `Int` or `Char`), then:

$$r(A) = 0.$$

(2) Function Types: For function types $T \to U$,

$$r(T \to U) = \begin{cases} r(U), & \text{if } r(T) = 0, \\ \max(r(T) + 1, r(U)), & \text{otherwise.} \end{cases}$$

(3) Universal Quantification: For universally quantified types $\forall A.T$,

$$r(\forall A.T) = \max(1, r(T)).$$

The following examples illustrate how the rank is computed:

Example 1.4.1 (Rank of `Int`)**.** Since `Int` is a base type and does not involve any function or quantification:

$$r(\texttt{Int}) = \mathbf{0}.$$

Example 1.4.2 (Rank of $\forall A.\, A \to \texttt{Int}$)**.** First compute the rank of $A \to \texttt{Int}$:

$$r(A \to \texttt{Int}) = r(\texttt{Int}) = 0 \quad (\text{since } r(A) = 0).$$

Then apply the universal quantification rule:

$$r(\forall A.\, A \to \texttt{Int}) = \max(1, 0) = \mathbf{1}.$$

Example 1.4.3 (Rank of $(\forall A.\, A \to \texttt{Int}) \to \texttt{Int}$)**.** From the previous example, we know $r(\forall A.\, A \to \texttt{Int}) = 1$. Applying the function rule:

$$r((\forall A.\, A \to \texttt{Int}) \to \texttt{Int}) = \max(1 + 1, 0) = \mathbf{2}.$$

Example 1.4.4 (Rank of $(((\forall A.\, A \to \texttt{Int}) \to \texttt{Int}) \to \texttt{Int})$)**.** Using the result from the previous example:

$$r((((\forall A.\, A \to \texttt{Int}) \to \texttt{Int}) \to \texttt{Int})) = \max(2 + 1, 0) = \mathbf{3}.$$

In the following, we contrast Rank-1 and Rank-2 types, providing examples in Haskell

Example 1.4.5 (Map Function)**.** The map function is a Rank-1 example. Its implicit type is:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

This is equivalent to the explicitly quantified version:

```
map :: forall a b. (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Rank-2 types allow functions to take polymorphic functions as arguments. That is, the argument itself can have a type like $\forall a.a \to a$.

Example 1.4.6 (a Rank-2 Function)**.** Here, in order to apply the function x to both the type Bool and Char, its type must be $\forall a.\, a \to a$.

```
f :: (forall a. a -> a) -> (Bool, Char)
f x = (x True, x 'A')
```

Chapter 2

From Simply Typed λ-Calculus to CaMPL

This chapter develops the basic algorithms used to assign types to expressions in functional programming languages. We begin with the simply typed λ-calculus, and the basic type theory for checking the types of terms. We then describe the Hindley–Milner system [1], which extends the simply typed lambda calculus with limited universal quantification. This leads to a discussion of Algorithm W introduced by Damas and Milner [1], which was developed for type inference in the Hindley–Milner system.
Finally, we present the type inference algorithm used in the Categorical Message Passing Language (CaMPL). In CaMPL, type inference is separated into two phases: it first generates type equations from the program and then it solves the type equations [14]. This is the basic approach which will be used in the thesis to describe universally quantified type inference.

2.1 Simply typed λ-calculus

A sequent such as $\Gamma \vdash t : T$ is called a type judgment. Each type judgment consists of a type context Γ, and a term t with its type T, written as $t : T$. A type context is a list of non-repeating variables with their types:

$$\Gamma := x_1 : T_1,\ x_2 : T_2,\ \ldots,\ x_n : T_n$$

The order of these variable-type associations does not matter [15]. In simply typed λ-calculus, types are either type variables (i.e., T) or arrow types (i.e., $T_1 \to T_2$).
If a type judgment can be derived using the rules of the simply typed λ-calculus (see Figure 2.1.1), we call it a valid type judgment.

The rules of simply typed λ-calculus can be used as a type checking system. These rules will be applied to a type judgment, to verify that a given term has a specific type. Each rule in this system describes a method of constructing a term. The projection rule (prj) looks up the type of a variable from the context, the application rule (app) forms a function application by applying a function to an argument, and the abstraction rule (abs) forms a function by abstracting over a variable with a given type [15, 16, 17, 18]. Figure 2.1.1 shows the inference rules for the simply typed λ-calculus.

$$\frac{x : P \in \Gamma}{\Gamma \vdash x : P}\ \text{prj}$$

$$\frac{\Gamma \vdash f : P \to Q \quad \Gamma \vdash t : P}{\Gamma \vdash (f\ t) : Q}\ \text{app}$$

$$\frac{\Gamma,\ x : P \vdash t : Q}{\Gamma \vdash \lambda x.\, t : P \to Q}\ \text{abs}$$

Figure 2.1.1. Simply typed λ-calculus rules

The following basic examples show how simply typed λ-calculus rules can be applied to construct a valid type judgement:

Example 2.1.2. The identity function $\lambda x.\, x$ can be typed as $A \to A$:

$$\frac{\dfrac{x : A \in x : A}{x : A \vdash x : A}\ \text{prj}}{\vdash \lambda x.\, x : A \to A}\ \text{abs}$$

Example 2.1.3. The curried first projection $\lambda x\, y.\, x$ has the type $A \to B \to A$:

$$\frac{\dfrac{\dfrac{x : A \in x : A, y : B}{x : A,\ y : B \vdash x : A}\ \text{prj}}{x : A \vdash \lambda y.\, x : B \to A}\ \text{abs}}{\vdash \lambda x\, y.\, x : A \to B \to A}\ \text{abs}$$

2.2 Hindley-Milner

The Hindley–Milner (HM) system is a type system for the λ-calculus that supports a restricted form of universal quantification, where the quantifier appears only at the outermost level of a type (known as a rank-1 type; see Section 1.4). It was originally described by Damas and Milner [1]. It is also explained in

[19], to which I shall refer. Hindley-Milner only supports universal quantification ($\forall$) at the outermost level of a type (rank-1 types), but does not allow nested or higher-rank quantification. For example, the identity function in Example 2.1.2 can be typed as $\forall X.\, X \to X$, which means that identity function can take a value of any type X and returns a value of the same type. The function works uniformly for all types, such as Int, Bool, or even lists [1, 19].

2.2.1 Hindley-Milner types

Figure 2.2.1 defines the syntax of types used in the Hindley-Milner system. A type T, referred to as a monomorphic type, is either a type variable or a function type. The extended type syntax T', known as a polymorphic type, includes universally quantified types of the form $\forall X.\, T'$, where the quantifier $\forall X$ has scope over the entire type. Thus, such quantifiers are permitted only at the outermost level and cannot appear within function types [1, 19].

$$
\begin{array}{rlll}
T ::= & X & & \text{(type variable)} \\
 & T \to T & & \text{(function type)} \\
T' ::= & T & & \text{(monomorphic type)} \\
 & \forall X.T' & & \text{(quantified type)}
\end{array}
$$

Figure 2.2.1. Hindley-Milner system types

2.2.2 Hindley–Milner type checking and type assignment system

The Hindley–Milner type checking rules extend the simply typed λ-calculus by adding the let rule, which corresponds to the cut rule[1], as well as the inst and gen rules to allow universal quantifiers only at the outermost level, called rank-1 type (see Figure 2.2.2).

The inst rule allows a rank-1 type to be instantiated to a more specific type (see Definition 2.2.3), while the gen rule generalizes a monomorphic type, which is either a type variable or a function type, to produce a rank-1 polymorphic type [19].

In Figure 2.2.2, $\mathrm{FV}(\Gamma)$ denotes the set of free type variables in Γ, and the relation $\sqsubseteq$ is defined in Definition 2.2.3.

[1] In sequent calculus, the cut rule can be written as: if $A \vdash B$ and $B \vdash C$, then $A \vdash C$.

$$\frac{\Gamma \vdash t : P' \quad \Gamma, x : P' \vdash s : Q}{\Gamma \vdash \text{let } x = t \text{ in } s : Q}\ \text{let}$$

$$\frac{\Gamma \vdash x : P' \quad P' \sqsubseteq P}{\Gamma \vdash x : P}\ \text{inst}$$

$$\frac{\Gamma \vdash x : T' \quad X \notin \text{FV}(\Gamma)}{\Gamma \vdash x : \forall X.T'}\ \text{gen}$$

Figure 2.2.2. Hindley-Milner type checking system

Definition 2.2.3. The phrase "P' is more general than T'", written as $P' \sqsubseteq T'$, means that we can instantiate type variables in P' to obtain T'. (see [19])

Example 2.2.4. The following examples demonstrates examples in which the relation $\sqsubseteq$ between two types is valid or invalid.

(1) Instantiating $X := \text{int}$:

$$\forall X.\, X \to X \sqsubseteq \text{int} \to \text{int}$$

(2) Instantiating $X := \text{bool}$:

$$\forall X.\, X \to X \sqsubseteq \text{bool} \to \text{bool}$$

(3) The following does not hold:

$$\forall X.\, X \to X \not\sqsubseteq \text{bool} \to \text{int}$$

Example 2.2.5. Given the typing context $\Gamma = n : \text{int},\ \text{id} : \forall X.\, X \to X$, we can type check $\text{id}\ n : \text{int}$ using the following derivation:

$$\dfrac{\dfrac{\dfrac{\text{id} : \forall X.\, X \to X \in \Gamma}{\Gamma \vdash \text{id} : \forall X.\, X \to X}\ \text{prj} \quad \forall X.\, X \to X \sqsubseteq \text{int} \to \text{int}}{\Gamma \vdash \text{id} : \text{int} \to \text{int}}\ \text{inst} \quad \dfrac{n : \text{int} \in \Gamma}{\Gamma \vdash n : \text{int}}\ \text{prj}}{\Gamma \vdash \text{id}\ n : \text{int}}\ \text{app}$$

Example 2.2.6. We can type check $\text{let } f = \lambda x.x \text{ in } f\ f : X \to X$

$$\dfrac{\dfrac{\dfrac{\dfrac{x : X \in \Gamma, x : X}{\Gamma, x : X \vdash x : X}\ \text{prj}}{\Gamma \vdash \lambda x.x : X \to X \quad X \notin \Gamma}\ \text{abs}}{\Gamma \vdash \lambda x.x : \forall X.\, X \to X}\ \text{gen} \quad \dfrac{\Pi_1 \quad \Pi_2}{\Gamma, f : \forall X.\, X \to X \vdash f\ f : X \to X}\ \text{app}}{\Gamma \vdash \text{let } f = \lambda x.x \text{ in } f\ f : X \to X}\ \text{let}$$

with Π_1 being:

$$\dfrac{\dfrac{f : \forall X.\, X \to X \in \Gamma, f : \forall X.\, X \to X}{\Gamma, f : \forall X.\, X \to X \vdash f : \forall X.\, X \to X}\ \text{prj} \quad \forall X.\, X \to X \sqsubseteq (X \to X) \to (X \to X)}{\Gamma, f : \forall X.\, X \to X \vdash f : (X \to X) \to (X \to X)}\ \text{inst}$$

with Π_2 being:

$$\dfrac{\dfrac{f : \forall X.\, X \to X \in \Gamma, f : \forall X.\, X \to X}{\Gamma, f : \forall X.\, X \to X \vdash f : \forall X.\, X \to X}\ \text{prj} \quad \forall X.\, X \to X \sqsubseteq X \to X}{\Gamma, f : \forall X.\, X \to X \vdash f : X \to X}\ \text{inst}$$

Given a context Γ and a term t, the Hindley–Milner type checking system does not provide a direct method for determining a type T' such that $\Gamma \vdash t : T'$. Thus, the Algorithm W is used to find the type T'. Given Γ and t, if Algorithm W succeeds, it produces a substitution S and a type T' such that $S\ \Gamma \vdash t : T'$ [1, 20].

The key difference between this type assignment algorithm and the one used in CaMPL, which will be discussed in the next sections, is that in CaMPL there are two distinct stages of generating type equations and solving them. Algorithm W does not explicitly generate equations; instead, it proceeds inductively through the structure of the term inferring the principal type of each subterm.

2.3 CaMPL

CaMPL is a concurrent functional programming language that supports rank-1 types [14]. While CaMPL includes many sequential and concurrent constructs, we focus on a small subset of its sequential constructs, specifically variables, abstractions, applications, and let bindings, because these are the terms considered in the Hindley-Milner type system. We refer to this fragment as Basic sequential CaMPL, which omits the more complex sequential constructs, making it easier to compare CaMPL's type inference rules with those of Hindley-Milner.

2.3.1 Basic sequential CaMPL types

The basic sequential types in CaMPL are presented in Figure 2.3.1. CaMPL does not support explicit universal quantifiers (i.e., $(\forall A.A) \to (\forall B.B)$). However, once the type of a function is determined, it can be called elsewhere in the program and its existential type variables are treated as universally quantified

(corresponding to the gen rule in the Figure 2.2.1). Each time the function is called, fresh type variables are instantiated in place of these universally quantified types (corresponding to the inst rule in the Figure 2.2.1).

$$\begin{array}{rll} T ::= & X & \text{(type variable)} \\ & T \to T & \text{(function type)} \end{array}$$

Figure 2.3.1. Basic sequential CaMPL types

2.3.2 Type inference in CaMPL

The type checking rules for Basic Sequential CaMPL are the same as the Hindley–Milner system (see Figure 2.2.2), but exclude the inst and gen rules that are used to support explicit rank-1 types. Therefore, it includes the prj, abs, and app rules from the simply typed λ-calculus and the let rule.

The type inference process used by CaMPL consists of two distinct phases [14].

1. Generating type equations: During this phase, type equations are generated to capture the constraints imposed on the types by the structure of the term being analyzed. To generate type equations for a term, we perform the rules in Figure 2.3.2. At each step, fresh type variables may be introduced, along with constraints that relate them to previously introduced types. We express this with the following form:

$$\exists X_1, \ldots, X_n.\ T_1 = T_1',\ \ldots,\ T_m = T_m'$$

where the freshly introduced type variables are listed on the left and the type constraints they participate in are on the right. In Figure 2.3.2, the type checking rules for the basic sequential CaMPL are modified to derive a list of type equations that any valid typing judgment must satisfy [15]. The algorithm used to solve these type equations is described in the next section.

$$\frac{}{\Gamma, t : A \vdash t : B \quad \langle A = B \rangle}\ \text{prj}$$

$$\frac{x : A, \Gamma \vdash t : B \ \langle E_1 \rangle}{\Gamma \vdash \lambda x.t : Q \quad \langle \exists A, B.Q = A \to B, E_1 \rangle}\ \text{abs}$$

$$\frac{\Gamma \vdash t : A \ \langle E_1 \rangle \quad \Gamma \vdash u : B \ \langle E_2 \rangle}{\Gamma \vdash t\ u : Q \ \langle \exists A, B.A = B \to Q, E_1, E_2 \rangle}\ \text{app}$$

$$\frac{\Gamma, x : A \vdash t : B \ \langle E_1 \rangle \quad \Gamma \vdash u : C \ \langle E_2 \rangle}{\Gamma \vdash \text{let x = u in t} : Q \ \langle \exists A, B, C.A = C, B = Q, E_1, E_2 \rangle}\ \text{let}$$

Figure 2.3.2. Type equation generation for basic sequential CaMPL

Example 2.3.3. Consider the term $\lambda xy.x$ that has the following derivation:

$$\dfrac{\dfrac{\dfrac{}{x : A, y : C \vdash x : D \quad E_2 = \langle A = D \rangle}\ \text{prj}}{x : A \vdash \lambda y.x : B \quad E_1 = \langle \exists C, D.B = C \to D, E_2 \rangle}\ \text{abs}}{\vdash \lambda xy.x : Q \quad \langle \exists A, B.Q = A \to B, E_1 \rangle}\ \text{abs}$$

The resulting type equation is:

$$\exists A, B.Q = A \to B, \langle \exists C, D.B = C \to D, \langle A = D \rangle \rangle$$

2. Solving type equations: The collected type equations are then solved to compute the most general type for the term. We may express what must be done to solve type equations by giving a set of rules for simplifying type equations:

Matching: An occurrence of an equation of the form $F(t_1, \ldots, t_n) = F(t'_1, \ldots, t'_n)$ can be replaced by the equations $t_1 = t'_1, \ldots, t_n = t'_n$. Repeatedly apply this reduction until one argument is a variable. The result is a formula of the form:

$$\exists X_1, \ldots, X_n.\ X_1 = T_1, \ldots, X_n = T_n$$

Eliminating existentials: Existentially bound variables for which there is a substitution can be eliminated by substitution: $\exists X.(X \equiv T, E)$ can be rewritten as $E[T/X]$. Keep repeating until there are

no more opportunities for elimination. If successful, this results is an equation of the form:

$$\exists X_1, ..., X_n.T_1 = Q, ..., T_n = Q$$

In this step one may also manipulate the scopes as follows:

$$\exists x.\exists y.E \equiv \exists y.\exists x.E$$

$$E_1, \exists y.E_2 \equiv \exists y.(E_1, E_2) \quad \text{if } y \notin E_1$$

It is important to note that this step might fail in the case of the occurs check, i.e., when there is an equation of the form $X = T$ with $X \in T$.

When the solving process starts, all variables except the principal variable Q should be quantified. The variable Q is intended to hold the most general type. After solving and eliminating all bound variables, it is possible to end up with multiple substitutions for Q (e.g., $\exists A, B, C.\ Q = T_1,\ Q = T_2,\ \ldots,\ Q = T_n$). In this case, one must keep only a single substitution for Q. A common approach is to select the first substitution $Q = T_1$ and use it to replace all other occurrences of Q, yielding $\exists A, B, C.\ Q = T_1,\ T_1 = T_2,\ \ldots,\ T_1 = T_n$. The process then continues, typically involving further variable elimination, until only one equation for Q remains [15].

2.4 Substitution lists

In the eliminating existentials step described above, if we accumulate the substitutions applied to eliminate each existential variable, we obtain a list of substitutions as the result. This list of substitutions we obtain after applying this algorithm has certain properties, which we describe below.
Consider a list of substitutions of the form $[S_1/X_1, \ldots, S_n/X_n]$.

Non-circularity: This list of substitutions is non-circular, meaning that X_i does not occur in S_i for $i \in \{1, \ldots, n\}$.

Left-linearized: This substitution list is left-linearized, meaning that it can be applied sequentially (with the first substitution applied first). This means that later substitutions in the list may affect the results of earlier ones, but not vice versa.

Parallelized: It can be transformed into a parallel substitution by replacing each substitution term with all later substitutions [21]. Formally $[S_1/X_1, \ldots, S_n/X_n]$ parallelizes to:

$$[S_1/X_1[S_2/X_2, \ldots, S_n/X_n],\ S_2/X_2[S_3/X_3, \ldots, S_n/X_n],\ \ldots,\ S_{n-1}/X_{n-1}[S_n/X_n], S_n/X_n]$$

Example 2.4.1. Consider the following equation:

$$\exists A, B.\ Q = A \to B,\ \langle \exists C, D.\ B = C \to D,\ \langle A = D \rangle \rangle$$

which is generated for the term $\lambda xy.x$ (see Example 2.3.3). In the process of solving this equation: First, eliminate D:

$$\exists A, B.\ Q = A \to B,\ \langle \exists C.\ B = C \to A \rangle$$

Manipulate the scopes:

$$\exists A, B, C.\ Q = A \to B, \quad B = C \to A$$

Then, eliminate B:

$$\exists A, C.\ Q = A \to (C \to A)$$

Now, if this was a function named `fstproj`, we would save its type as

$$\forall A, C.\ A \to (C \to A)$$

in the symbol table, corresponding to the gen rule in the Hindley-Milner type system. Whenever we want to call `fstproj`, we can instantiate A and C with fresh type variables, corresponding to the inst rule in Hindley-Milner.

Example 2.4.2. Consider `fstproj` to have type $\forall A, C.\ A \to (C \to A)$ and to be present in the symbol table. We want to perform type inference for the term (`fstproj 1 'c'`).

$$\frac{\Pi_1 \quad \dfrac{}{1 : \text{Int}, \text{`c'} : \text{Char} \vdash (\text{`c'}) : B\ E_2 = \langle B = \text{Char} \rangle}\ \text{prj}}{1 : \text{Int}, \text{`c'} : \text{Char} \vdash (\texttt{fstproj}\ 1\ \text{`c'}) : Q\ \langle \exists A, B.A = B \to Q, E_1, E_2 \rangle}\ \text{app}$$

with Π_1 being:

$$\frac{\Pi_2 \quad \Pi_3}{1 : \text{Int}, \text{`c'} : \text{Char} \vdash (\texttt{fstproj}\ 1) : A\ \langle \exists C, D.C = D \to A, E_3, E_4 \rangle}\ \text{app}$$

with Π_2 being:

$$\frac{}{1 : \text{Int}, \text{`c`} : \text{Char} \vdash \texttt{fstproj} : C\ E_3 = \langle \exists E, F.C = E \to (F \to E) \rangle}\ \text{prj}$$

with Π_3 being:

$$\frac{}{1 : \text{Int}, \text{`c`} : \text{Char} \vdash 1 : D\ E_4 = \langle D = \text{Int} \rangle}\ \text{prj}$$

After collecting and solving the equations, the type of the term (`fstproj 1 'c'`) is Int. This example illustrates how fresh type variables are introduced when calling a function from the symbol table.

2.5 Summary

In this chapter, we saw that the Hindley–Milner type checking system does not provide an algorithm for inferring types. To handle this, Algorithm W was introduced for type assignment. CaMPL has the same expressiveness as Hindley–Milner, but unlike Algorithm W, its type inference is organized into two distinct steps: collecting and solving type equations.

Chapter 3

System F and Subsumption

In this chapter, we introduce system F which is a basic system for working with higher rank types. We then adapt system F to make it compatible with programming, which gives us system S. We introduce the notion of subsumption, which plays a vital role in defining the rules of type checking and inference. We also show how subsumptions may be used to augment system S with coercion rules. Finally, we prove that coercion rules are valid in system S.

3.1 System F

System F, also known as the second-order lambda calculus, is an explicitly typed extension of the simply typed lambda calculus that introduces support for higher-rank types. This system serves as a theoretical foundation for polymorphic type systems, where universal quantifiers can appear inside function arguments or return types [22].

3.1.1 System F syntax

Figure 3.1.1 shows the types of system F.

$$
\begin{array}{rlr}
T ::= & X & \text{(type variable)} \\
 & T \to T & \text{(function type)} \\
 & \forall X.\, T & \text{(universal type)}
\end{array}
$$

Figure 3.1.1. System F types

Figure 3.1.2 illustrates the terms of system F. These extend the simply typed λ-calculus with a type abstraction and type application.

$t ::=$	x	(variable)
	$\lambda x : T.\, t$	(abstraction)
	$t\ t$	(application)
	$\Lambda X.\, t$	(type abstraction)
	$t\ [T]$	(type application)

Figure 3.1.2. System F terms

In system F a type abstraction $\Lambda X.\, t$ can be applied to a type T, written as $(\Lambda X.t)[T]$, and forms a redex that evaluates to $t[T/X]$. This is the β-reduction for type application. Thus, applying a type abstraction $\Lambda X.t$ to a type T substitutes all occurrences of X in t with T. This is analogous to the β reduction for terms, written as $(\lambda x : T.t)s$, which evaluates to $t[s/x]$. Thus, applying a lambda abstraction $\lambda x : T.t$ to a term s can be reduced by substituting all occurrences of x in t with s [17].

Example 3.1.3. Consider the polymorphic identity function

$$\text{id} = \Lambda X.(\lambda x : X.x).$$

Applying this function to the type Nat:

$$\text{id}[\text{Nat}] = (\Lambda X.(\lambda x : X.x))[\text{Nat}],$$

reduces, using the β reduction for type application, to

$$\lambda x : \text{Nat}.x.$$

Thus, id[Nat] specializes the polymorphic identity function to the identity function on natural numbers.

3.1.2 Context in system F

A context, denoted by Γ, is constructed by adding new type variables or term variable bindings.
A type variable can be added at any point in the context, written as Γ, T. To add a term variable binding, written as $\Gamma := \Gamma, x : T$, the type T must be constructed from the type variables already declared in Γ.

The formal rules for forming contexts are presented in Figure 3.1.4.

$$\frac{}{X, \Gamma \vdash_{\text{type}} X}\ \text{tp} \qquad \frac{}{X, \Gamma \vdash x : X}\ \text{term}$$

$$\frac{\Gamma \quad \text{context}}{\Gamma, X \quad \text{context}}\ \text{tpadd} \qquad \frac{\Gamma \vdash_{\text{type}} X}{\Gamma, x : X \quad \text{context}}\ \text{termadd}$$

$$\frac{\Gamma, X \vdash_{\text{type}} T}{\Gamma \vdash_{\text{type}} \forall X.\, T}\ \text{forall} \qquad \frac{\Gamma \vdash_{\text{type}} T \quad \Gamma \vdash_{\text{type}} T'}{\Gamma, (x : T \to T') \quad \text{context}}\ \text{arrow}$$

$$\frac{\Gamma_1, X, Y, \Gamma_2 \quad \text{context}}{\Gamma_1, Y, X, \Gamma_2 \quad \text{context}}\ \text{exch}_{\text{tptp}} \qquad \frac{\Gamma_1, x : T, X, \Gamma_2 \quad \text{context}}{\Gamma_1, X, x : T, \Gamma_2 \quad \text{context}}\ \text{exch}_{\text{termtp}}$$

$$\frac{\Gamma, x : X, y : Y \vdash e : T}{\Gamma, y : Y, x : X \vdash e : T}\ \text{exch}_{\text{termterm}} \qquad \frac{\Gamma_1, Y, x : X, \Gamma_2 \quad \text{context} \quad \Gamma_1 \vdash_{\text{type}} X}{\Gamma_1, x : X, Y, \Gamma_2 \quad \text{context}}\ \text{exch}_{\text{tpterm}}$$

Figure 3.1.4. System F context formation, type construction and exchange rules

There are three types of judgments used in Figure 3.1.4:

- Context judgment ($\Gamma \quad$ context) : this says Γ is a context.
- Type judgment ($\Gamma \vdash_{\text{type}} T$) : this says type T can be constructed from the context Γ.
- Term judgment ($\Gamma \vdash x : T$) : this associates a term variable to a type.

The context formation rules shown in Figure 3.1.4 are discussed in detail below.

Context extension: The context can be extended by a type or a term variable :

(1) "tpadd": adds a type variable to the context.

$$\frac{\Gamma \quad \text{context}}{\Gamma, X \quad \text{context}}\ \text{tpadd}$$

(2) "termadd": adds a term variable with its type to the context. The type assigned to a term must be constructed from the type variables already declared in Γ [23].

$$\frac{\Gamma \vdash_{\text{type}} T}{\Gamma, x : T \quad \text{context}}\ \text{termadd}$$

Type construction: Universally quantified types and arrow types can be constructed using the existing types in the context:

(1) "forall": introduces a universally quantified type ($\forall$) into a derivation. The forall rule imposes a condition: the quantified type variable must not appear free in the context outside of its binding. This ensures that the following derivation is valid:

$$\frac{\Gamma, X \vdash_{\text{type}} T}{\Gamma \vdash_{\text{type}} \forall X.\, T}\ \text{forall}$$

In contrast, the "forall" rule cannot be applied to

$$\Gamma, X, x : X \vdash_{\text{type}} T,$$

because in this case the context contains a term variable binding $x : X$ that depends on the type variable X. As a result, X cannot appear last in the context: the type X assigned to x must be constructible from the existing context Γ.

(2) "arrow": constructs a function type $(A \to B)$ from two types, A and B.

$$\frac{\Gamma \vdash_{\text{type}} T \quad \Gamma \vdash_{\text{type}} T'}{\Gamma, (x : T \to T') \quad \text{context}}\ \text{arrow}$$

It is important to note that in this chapter we only consider arrow types. In the next chapters, when defining type equations, we use a more general form of type constructors that includes arrow types.

Exchange: The rules "exchtp1", "exchtp2", "exchterm1", and "exchterm2" allow reordering of type and term variables in the context, while ensuring that term variable declarations only use types derivable from the context in which they are introduced.

After presenting the context formation rules, we now state that the following substitution properties also hold for System F contexts [23].

(1) cut: eliminates a term variable from the context by substituting another term.

$$\frac{\Gamma \vdash y : T \quad \Gamma, z : T \vdash t : T'}{\Gamma \vdash t[y/z] : T'}\ \text{cut}$$

(2) cut_{type}: eliminates a type variable by substituting another type.

$$\frac{\Gamma \vdash_{\text{type}} S \quad \Gamma,\ T \vdash_{\text{type}} T'}{\Gamma \vdash_{\text{type}} T'[S/T]}\ \text{cut}_{\text{type}}$$

(3) $\text{cut}_{\text{tpterm}}$: replaces a type inside the type of a term variable with another type.

$$\frac{\Gamma \vdash_{\text{tpterm}} S \quad \Gamma,\ X \vdash t : T'}{\Gamma \vdash t[S/X] : T'[S/X]}\ \text{cut}$$

Example 3.1.5. The following shows how one may apply context formation rules:

(1) In the following context:

$$\Gamma := Z, x : \forall A.\, A \to A, Y, y : Y \to Z, z : Z,$$

the exchange rule $\text{exch}_{\text{tpterm}}$ cannot be applied to reorder Y and $y : Y \to Z$, because the type $Y \to Z$ depends on Y being in the context before y is added. Moving y before Y would make the context ill-formed, since the type $Y \to Z$ would no longer be derivable.

However, the rule $\text{exch}_{\text{tpterm}}$ can be applied to reorder Z and $x : \forall A.\, A \to A$, because the type $\forall A.\, A \to A$ does not depend on Z. The following inference is valid:

$$\frac{Z, x : \forall A.\, A \to A, Y, y : Y \to Z, z : Z \quad \text{context}}{x : \forall A.\, A \to A, Z, Y, y : Y \to Z, z : Z \quad \text{context}}\ \text{exch}_{\text{tpterm}}$$

(2) $\Gamma := x : \forall A.\, A$ is a valid context. The following derivation shows the formation of this context:

$$\dfrac{\dfrac{\dfrac{}{A \vdash_{\text{type}} A}\ \text{tp}}{\vdash_{\text{type}} \forall A.\, A}\ \text{forall}}{x : \forall A.\, A \quad \text{context}}\ \text{termadd}$$

(3) $\Gamma := Y, x : Y \to Y, z : \forall X.\, X$ is a valid context. The following demonstrates how this context can be constructed:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Y \vdash_{\text{type}} Y}\,\text{tp} \quad \dfrac{}{Y \vdash_{\text{type}} Y}\,\text{tp}}{Y, x : Y \to Y \quad \text{context}}\,\text{arrow}}{Y, x : Y \to Y, X \quad \text{context}}\,\text{tpadd}}{Y, x : Y \to Y, X \vdash_{\text{type}} X}\,\text{tp}}{Y, x : Y \to Y \vdash_{\text{type}} \forall X.\, X}\,\text{forall}}{Y, x : Y \to Y, z : \forall X.\, X \quad \text{context}}\,\text{termadd}$$

(4) $\Gamma := Z, x : \forall A.\, A \to A, Y, y : Y \to Z, z : Z.$ is a valid context. This context can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Z, A \vdash_{\text{type}} A}\,\text{tp} \quad \dfrac{}{Z, A \vdash_{\text{type}} A}\,\text{tp}}{Z, A \vdash_{\text{type}} A \to A}\,\text{arrow}}{Z \vdash_{\text{type}} \forall A.\, A \to A}\,\text{forall}}{Z, x : \forall A.\, A \to A \quad \text{context}}\,\text{termadd}}{Z, x : \forall A.\, A \to A, Y \quad \text{context}}\,\text{tpadd}}{Z, x : \forall A.\, A \to A, Y \vdash_{\text{type}} Y}\,\text{tp} \qquad \dfrac{}{Z, x : \forall A.\, A \to A, Y \vdash_{\text{type}} Z}\,\text{tp}}{Z, x : \forall A.\, A \to A, Y, y : Y \to Z \quad \text{context}}\,\text{arrow}$$

3.1.3 System F typing rules

Figure 3.1.6 defines the typing rules of System F. The rules T-Var$_F$, T-Abs$_F$, and T-App$_F$ correspond to the typing rules for variables, lambda abstractions, and applications in the simply typed λ-calculus. The rules T-TAbs$_F$ and T-TApp$_F$ extend this system with type abstraction and type application, enabling the construction and use of polymorphic functions [24].

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}_F$$

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T.t : T \to T'}\ \text{T-Abs}_F$$

$$\frac{\Gamma \vdash f : T \to T' \quad \Gamma \vdash t : T}{\Gamma \vdash f t : T'}\ \text{T-App}_F$$

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}\ \text{T-TAbs}_F$$

$$\frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t[T'] : T[T'/X]}\ \text{T-TApp}_F$$

Figure 3.1.6. System F typing rules

Example 3.1.7. The polymorphic identity function $\text{id} = \Lambda X.\,(\lambda x : X.\,x)$, discussed in Example 3.1.3, can be derived using the typing rules of System F, as shown below:

$$\dfrac{\dfrac{\dfrac{x : X \in X, x : X}{X, x : X \vdash x : X}\ \text{T-Var}_F}{X \vdash \lambda x : X.\,x : X \to X}\ \text{T-Abs}_F}{\vdash \Lambda X.\,\lambda x : X.\,x : \forall X.\,X \to X}\ \text{T-TAbs}_F$$

Figure 3.1.8 adds two typing rules to the main rules of System F, resulting in System $F_{\eta+\text{annotation}}$. The first rule, Annot_F, introduces type annotations for terms. Since system F does not support annotations, this rule allows writing terms of the form $(t :: T)$, which enforce t to be of type T. The following is an example of Annot_F rule:

$$\frac{A, x : A \vdash x : A}{A, x : A \vdash (x :: A) : A}\ \text{Annot}_F$$

The second rule is, the η rule, η_F [25]. This rules captures the fact that when a variable x does not occur free in a term t, the function $\lambda x.t\ x$ behaves identically to t, and so should be regarded as equal to t. Explicitly, we have:

$$(\lambda x.t\ x)\ s = (t\ x)[s/x] = t\ s$$

That is, applying $\lambda x.t\ x$ to any term s yields the same result as applying t to s. As a result, $\lambda x.t\ x$ and t behave in the same way, and therefore they have the same type.

$$\frac{\Gamma \vdash \lambda x : T.t\, x : T \to T' \quad x \notin FV(t)}{\Gamma \vdash t : T \to T'}\ \eta_{\mathrm{F}}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash (x :: T) : T}\ \text{Annot}_F$$

Figure 3.1.8. System $F_{\eta\text{+annotation}}$

The reverse of η_F rule is also provable in this system as follows:

$$\frac{\dfrac{\dfrac{t : T \to T' \in t : T \to T'}{T, T', t : T \to T', x : T \vdash t : T \to T'}\ \text{T-Var}_F \quad \dfrac{x : T \in x : T}{T, T', x : T \vdash x : T}\ \text{T-Var}_F}{T, T', t : T \to T', x : T \vdash t\, x : T'}\ \text{T-App}_F}{T, T', t : T \to T' \vdash \lambda x : T.t\, x : T \to T'}\ \text{T-Abs}_F$$

Making the η_F rule a two-way rule.

3.2 System S

System F does not align directly with programming language syntax, because it requires explicit type abstractions ($\Lambda X.t$) and type applications ($t\,[T]$) in the term syntax. In contrast, most languages support implicit polymorphism with type inference, so programmers do not write type-level constructs directly. By removing explicit types from system F while keeping its polymorphic power, we get a new system which we will refer to as system S. Type inference and type checking in system S is proved to be undecidable [25].

3.2.1 System S syntax

The types in System S are identical to those in System F, as shown in Figure 3.2.1.

T	::=	X	type variable
		$T \to T$	function type
		$\forall X.\, T$	universal type

Figure 3.2.1. System S types

Figure 3.2.2 presents the terms of system S, which, unlike those of system F, omit type quantification and type application but allow type annotations.

t	::=	x	variable
		$\lambda x.t$	abstraction
		$t\ t$	application
		$t :: T$	annotation

Figure 3.2.2. System S terms

The context formation rules in system S are identical to those in system F, and therefore are not discussed further in this section (see Figure 3.1.4).

3.2.2 System S typing rules

Figure 3.2.3 shows the transformation of system $F_{\eta+\text{annotation}}$ typing rules into system S, a form compatible with programming languages by removing explicit type application and type abstraction at the term level.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}_F$$

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x{:}\cancel{T}.t : T \to T'}\ \text{T-Abs}_F$$

$$\frac{\Gamma \vdash f : T \to T' \quad \Gamma \vdash t : T}{\Gamma \vdash f\ t : T'}\ \text{T-App}_F$$

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \cancel{\Lambda X}.t : \forall X.T}\ \text{T-TAbs}_F$$

$$\frac{\Gamma \vdash t : \forall X.T \quad T' \text{ type}}{\Gamma \vdash t\cancel{[T]} : T[T'/X]}\ \text{T-TApp}_F$$

$$\frac{\Gamma \vdash \lambda x{:}\cancel{T}.tx : T \to T' \quad x \notin FV(t)}{\Gamma \vdash t : T \to T'}\ \eta_F$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash (x :: T) : T}\ \text{Annot}_F$$

Figure 3.2.3. System $F_{\eta\text{+annotation}}$ to system S transformation

Definition 3.2.4 (Erasure of term-level type quantification and application)**.** If t is a term in system $F_{\eta\text{+annotation}}$, then the term $\tilde{t}$ in system S is obtained by removing the explicit type quantifications and application at the term level from t, as illustrated in Figure 3.2.3.

The typing rules of system S are presented in Figure 3.2.5. Unlike the T-TApp$_F$ rule in system F, type substitution in system S is implicit in the T-TApp rule, rather than being made explicit at the level of terms.

Another key distinction between the two systems is the removal of ΛX from the rules T-Abs and T-TAbs in System S. This modification eliminates type dependencies from the term syntax.

System S also includes type annotation and the η-rule. These are inherited from system $F_{\eta\text{+annotation}}$ (see Figure 3.1.8).

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}$$

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x.t : T \to T'}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash f : T \to T' \quad \Gamma \vdash t : T}{\Gamma \vdash f\,t : T'}\ \text{T-App}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash t_2 : \forall X.T_2}\ \text{T-TAbs}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_1 \quad \Gamma \vdash_{\text{type}} T_2}{\Gamma \vdash t_1 : T_1[T_2/X]}\ \text{T-TApp}$$

$$\frac{\Gamma \vdash \lambda x.t\,x : T \to T' \quad x \notin FV(t)}{\Gamma \vdash t : T \to T'}\ \eta$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash (x :: T) : T}\ \text{Annot}$$

Figure 3.2.5. System S typing rules

The reverse of η rule is also provable in this system as following:

$$\frac{\dfrac{T, T', x : T \vdash t : T \to T' \quad T, T', x : T \vdash x : T}{T, T', x : T \vdash t\,x : T'}\ \text{T-App}}{T, T' \vdash \lambda x.t\,x : T \to T'}\ \text{T-Abs}$$

Example 3.2.6. The polymorphic identity function in system S can be written as $(\lambda x.\, x :: \forall A.\, A \to A)$, and the following derivation shows how this term can be derived in system S.

$$\frac{\dfrac{\dfrac{\dfrac{x : A \in x : A}{A, x : A \vdash x : A}\ \text{T-Var}}{A \vdash \lambda x.\, x : A \to A}\ \text{T-Abs}}{\vdash \lambda x.\, x : \forall A.\, A \to A}\ \text{T-TAbs}}{\vdash (\lambda x.\, x :: \forall A.\, A \to A) : \forall A.\, A \to A}\ \text{Annot}$$

3.3 Subsumption in system S

Definition 3.3.1. Subsumption. We say that a type A subsumes another type A' in system S, written $A \preceq A'$, if we have the following implication:

$$\frac{\Pi}{\Gamma \vdash t : A} \quad \Rightarrow \quad \frac{\Pi_{A \preceq A'}}{\Gamma_{A \preceq A'} \vdash t : A'}$$

That is, whenever there is a valid proof that $t : A$, this proof can be transformed into a valid proof that $t : A'$. Such a transformation may require an algorithmic adjustment of the context and derivation, denoted with $A \preceq A'$. In particular, this adjustment may consist of adding a type variable to the context or substituting type variables within the context, as discussed below.

3.3.1 Substitutions are subsumption

The most basic subsumption rule is given by a type substitution: $A \preceq A[\Sigma]$. In this case, the transformation is simply a type substitution:

$$\frac{\Pi}{\Gamma \vdash t : A} \quad \Rightarrow \quad \frac{\Pi[\Sigma]}{\Gamma[\Sigma] \vdash t : A[\Sigma]}$$

This relation corresponds to the use of $\sqsubseteq$ in the Hindley–Milner system. (see Definition 2.2.3)

Theorem 3.3.2 (Substitution is an example of Subsumption)**.** The following holds for all terms in system S:

$$\frac{\Pi}{\Gamma \vdash t : A} \quad \Rightarrow \quad \frac{\Pi[\Sigma]}{\Gamma[\Sigma] \vdash t : A[\Sigma]}$$

Proof. To verify that Theorem 3.3.2 holds for all terms in system S, we proceed by structural induction on the proof of the terms in system S, where the structure of terms corresponds to the typing rules of the system.

T-Var: The following should hold:

If $x : T$ is valid, $x : T[\Sigma]$ is also valid.

(i.e., $T \preceq T[\Sigma]$)

The corresponding proof transformation is:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var} \quad \Rightarrow \quad \frac{x : T[\Sigma] \in \Gamma[\Sigma]}{\Gamma[\Sigma] \vdash x : T[\Sigma]}\ \text{T-Var}$$

which is the base case of induction.

T-Abs: The following should hold:

$$\text{If } \lambda x.t : (T \to S) \text{ is valid, } \lambda x.t : (T \to S)[\Sigma] \text{ is also valid.}$$
$$(\text{ i.e., } (T \to S) \preceq (T \to S)[\Sigma]\ )$$

The corresponding proof transformation is:

$$\frac{\dfrac{\Pi}{\Gamma, x : T \vdash t : S}}{\Gamma \vdash \lambda x.t : (T \to S)}\ \text{T-Abs} \qquad \Rightarrow \qquad \frac{\dfrac{\Pi[\Sigma]}{\Gamma[\Sigma], x : T[\Sigma] \vdash t : S[\Sigma]}}{\Gamma[\Sigma] \vdash \lambda x.t : (T \to S)[\Sigma]}\ \text{T-Abs}$$

and $\Pi[\Sigma]$ inductively holds.

T-App: The following should hold:

$$\text{If } ft : T \text{ is valid, } ft : T[\Sigma] \text{ is also valid.}$$
$$(\text{ i.e., } T \preceq T[\Sigma]\ )$$

The corresponding proof transformation is:

$$\frac{\dfrac{\Pi_1}{\Gamma \vdash f : (S \to T)} \quad \dfrac{\Pi_2}{\Gamma \vdash t : S}}{\Gamma \vdash ft : T}\ \text{T-App} \qquad \Rightarrow \qquad \frac{\dfrac{\Pi_1[\Sigma]}{\Gamma[\Sigma] \vdash f : (S \to T)[\Sigma]} \quad \dfrac{\Pi_2[\Sigma]}{\Gamma[\Sigma] \vdash t : S[\Sigma]}}{\Gamma[\Sigma] \vdash ft : T[\Sigma]}\ \text{T-App}$$

and $\Pi_1[\Sigma], \Pi_2[\Sigma]$ inductively hold.

T-TAbs: The following should hold:

$$\text{If } t : \forall X.T \text{ is valid, } t : \forall X.T[\Sigma] \text{ is also valid.}$$
$$(\text{ i.e., } (\forall X.T) \preceq (\forall X.T)[\Sigma]\ )$$

The corresponding proof transformation is:

$$\dfrac{\dfrac{\Pi}{\Gamma, X \vdash t : T}}{\Gamma \vdash t : (\forall X.T)}\ \text{T-TAbs} \qquad \Rightarrow \qquad \dfrac{\dfrac{\Pi[\Sigma]}{\Gamma[\Sigma], X \vdash t : T[\Sigma]}}{\Gamma[\Sigma] \vdash t : (\forall X.T)[\Sigma]}\ \text{T-TAbs}$$

and $\Pi[\Sigma]$ holds inductively. In the substitution set Σ, the type variable X will not be substituted, as this type variable does not appear in Γ (see Figure 3.1.6).

T-TApp: The following should hold:

$$\text{If } t : T\ [S/X] \text{ is valid, } t : T\ [S/X][\Sigma] \text{ is also valid.}$$
$$(\text{ i.e., } T\ [S/X] \preceq T\ [S/X][\Sigma]\ )$$

The corresponding proof transformation is:

$$\dfrac{\dfrac{\Pi}{\Gamma \vdash t : (\forall X.T)}}{\Gamma \vdash t : (T[S/X])}\ \text{T-TApp} \qquad \Rightarrow \qquad \dfrac{\dfrac{\Pi[\Sigma]}{\Gamma[\Sigma] \vdash t : (\forall X.T)[\Sigma]}}{\Gamma[\Sigma] \vdash t : (T[S/X])[\Sigma]}\ \text{T-TApp}$$

and $\Pi[\Sigma]$ inductively holds.

□

3.3.2 Instantiating quantified types are subsumptions

The other subsumption rule we consider is given by eliminating $\forall$ bindings: $\forall A.T \preceq T$

$$\dfrac{\Pi}{\Gamma \vdash t : \forall A.T} \qquad \Rightarrow \qquad \dfrac{\Pi'}{\Gamma, A \vdash t : T}$$

Theorem 3.3.3 (Eliminating forall bindings gives Subsumption)**.** The following holds for all the terms of system S:

$$\text{If } t : \forall A.T \text{ is valid, } t : T \text{ is also valid.}$$
$$(\text{i.e., } \forall A.T \preceq T)$$

Proof. Π' is a valid proof for $t : T$, if the type variable A is added into the context.

$$\frac{\dfrac{\Pi'}{\Gamma, A \vdash t : T}}{\Gamma \vdash t : \forall A.T}\ \text{T-TAbs} \qquad \Rightarrow \qquad \frac{\Pi'}{\Gamma, A \vdash t : T}$$

□

3.4 Coercion in system S

The subsumption rules discussed in the previous section can be used to introduce coercions into type system S:

$$\frac{A \preceq A'}{x : A \vdash x : A'}\ \text{coerce}$$

As two forms of subsumptions have been considered, the coercion rules can occur in two forms in a proof:

(1) coercion for substitution:

$$\frac{T \preceq T[\Sigma]}{x : T \vdash x : T[\Sigma]}\ \text{coercesub}$$

Note that for each substitution S/X in Σ, the type variable X does not occur in S.

(2) coercion for eliminating foralls:

$$\frac{\forall X.T \preceq T}{x : \forall X.T \vdash x : T}\ \text{coerceall}$$

A simple example of cutting with a coercion is as follows:

$$\frac{\Gamma \vdash t : A \qquad \dfrac{A \preceq B}{x : A \vdash x : B}\ \text{coerce}}{\Gamma \vdash t : B}\ \text{cut}$$

In a proof, cutting with a coercion causes a translation of the proof and context of the coerced term. The introduction of coercions hides the proof transformations, which is convenient but, perhaps, not completely transparent. However, using coercions in system S is proved to be valid in Theorem 3.4.1.

Theorem 3.4.1. Any judgment that can be derived using the coerceall and coercesub rules can also be derived without using them.

Proof. We proceed in the following stages.

First, we show that if a judgment involves a single occurrence of the coercesub rule, then that judgment can also be derived without using coercesub (Lemma 3.4.2). Next, we establish the same result for a single occurrence of the coerceall rule (Lemma 3.4.3).

We then observe that when multiple coercions are present, the order in which they are removed becomes significant (Example 3.4.4). To handle this, we first remove all coerceall steps. We explain how these can be eliminated in an order that ensures they do not interfere with one another. Finally, once all coerceall rules are eliminated, we prove in Theorem 3.4.5 that the remaining coercesub rules can be removed in any order. □

Lemma 3.4.2. Any judgment that can be derived using a single coercesub rule can also be derived without using it.

Proof. We assume that there is a valid proof of $\Gamma' \vdash s : T'$ in system S using a single coercesub. We then show that there will be a valid proof for the term s without the use of coercesub.

A coercesub can occur on the left or right of a proof:

coercesub (right): If the coercesub rule cut into the proof from right:

$$\dfrac{\dfrac{\Pi}{\Gamma \vdash t : A} \quad \dfrac{A \preceq A'}{\Gamma, x : A \vdash x : A'}\ \text{coercesub}}{\dfrac{\Gamma \vdash t : A'}{\Gamma' \vdash s : T'}\ \Pi'}\ \text{cut}$$

where $A' = A[\Sigma]$. It needs to be shown that s will still be a valid term, if one does not use the coercesub rule.

$$\dfrac{\dfrac{\dfrac{\Pi[\Sigma]}{\Gamma[\Sigma] \vdash t : A[\Sigma]}}{\Gamma[\Sigma] \vdash t : A'[\Sigma]}\ \cancel{\text{cut}}}{\Gamma'[\Sigma] \vdash s : T'[\Sigma]}\ \Pi'[\Sigma]$$

Since $A' = A[\Sigma] = A'[\Sigma]$, this proof is valid.

coercesub (left): If the coercesub rule cut into the proof from left:

$$\dfrac{\dfrac{A \preceq A'}{\Gamma, x : A \vdash x : A'}\ \text{coercesub} \quad \dfrac{\Pi}{\Gamma, x : A' \vdash t : T}}{\dfrac{\Gamma, x : A \vdash t : T}{\Gamma' \vdash s : T'}\ \Pi'}\ \text{cut}$$

where $A' = A[\Sigma]$. It needs to be shown that s will still be a valid term, if one does not use the coercesub rule.

$$\dfrac{\dfrac{\dfrac{\Pi[\Sigma]}{\Gamma[\Sigma], x : A'[\Sigma] \vdash t : T[\Sigma]}}{\Gamma[\Sigma], x : A[\Sigma] \vdash t : T[\Sigma]}\ \cancel{\text{cut}}}{\Gamma'[\Sigma] \vdash s : T'[\Sigma]}\ \Pi'[\Sigma]$$

Since $A' = A[\Sigma] = A'[\Sigma]$, this proof is valid.

□

Lemma 3.4.3. Any judgment that can be derived using a single coerceall rule can also be derived without using them.

Proof. A coerceall can occur on the left or right of a proof:

coerceall (right): If the coerceall rule cut into the proof from left:

$$\dfrac{\dfrac{\Pi}{\Gamma \vdash t : \forall X.T}\ \text{coerceall} \quad \dfrac{\forall X.T \preceq T}{\Gamma, x : \forall X.T \vdash x : T}}{\dfrac{\Gamma \vdash t : T}{\Gamma' \vdash s : T'}\ \Pi'}\ \text{cut}$$

It needs to be shown that s will still be a valid term, if one does not use the coerceall rule.

If one consider cut to be the T-TApp rule:

$$\dfrac{\dfrac{\dfrac{\Pi}{\Gamma \vdash t : \forall X.T}\ \text{tpadd}}{\Gamma, X \vdash t : \forall X.T}}{\dfrac{\Gamma, X \vdash t : T[X/X]}{\Gamma', X \vdash s : T'}\ \Pi'}\ \text{T-TApp}$$

s will still be a valid term.

coerceall (left): if the coerceall rule cut into the proof from right:

$$\dfrac{\dfrac{\forall X.T \preceq T}{\Gamma, x : \forall X.T \vdash x : T}\ \text{coerceall} \quad \dfrac{\Pi}{\Gamma, x : T \vdash t : S}}{\dfrac{\Gamma, x : \forall X.T \vdash t : S}{\Gamma' \vdash s : T'}\ \Pi'}\ \text{cut}$$

It needs to be shown that s will still be a valid term, if one does not use the coerceall rule.

If we consider the proof to be as following:

$$\dfrac{\dfrac{\Pi}{\Gamma, x : \forall X.T \vdash t : S}\ \cancel{\text{cut}}}{\Gamma' \vdash s : T'}\ \Pi'$$

One only needs to modify the proof Π when the variable x is pulled from the context. The following derivation shows that it is possible to reconstruct the judgment $x : \forall X.T$ by reasoning from $x : T$, provided that the type variable X is available in the context. This is achieved by reconstructing the type $\forall X.T$ through applying context formation rules (see Figure 3.1.4). Therefore, even without using a cut, we may equivalently assume $x : T$ in the context used in Π.

$$\dfrac{\dfrac{\Gamma, x : \forall X.T \vdash x : \forall X.T}{\Gamma, x : \forall X.T \vdash t : S}\ \Pi}{\Gamma' \vdash s : T'}\ \Pi' \quad \Rightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : T}\ \text{term}}{\Gamma, X \vdash_{\text{type}} T}\ \text{tp}}{\Gamma \vdash_{\text{type}} \forall X.T}\ \text{forall}}{\Gamma, x : \forall X.T \vdash x : \forall X.T}\ \text{termadd}}{\Gamma, x : \forall X.T \vdash t : S}\ \Pi}{\Gamma' \vdash s : T'}\ \Pi'$$

□

Example 3.4.4. This example demonstrates that eliminating coercesubs before coerceallss may fail, whereas eliminating coerceallss before coercesubs (the reverse ordering) succeeds.

Consider a proof involving two coercions, a coercesub and a coerceall:

$$\dfrac{\dfrac{\Pi}{\Gamma \vdash t : \forall X.T} \quad \dfrac{\dfrac{\forall X.T \preceq T}{\Gamma, x : \forall X.T \vdash x : T}\ \text{coerceall}}{\Gamma \vdash t : T}\ \text{cut}_2 \quad \dfrac{T \preceq S}{\Gamma, x : T \vdash x : S}\ \text{coercesub}}{\Gamma \vdash t : S}\ \text{cut}_1$$

If one attempts to eliminate the coercesub rule first, the substitution Σ, such that $S = T[\Sigma]$, must be applied to the proof, resulting in

$$\dfrac{\dfrac{\Pi[\Sigma]}{\Gamma[\Sigma] \vdash t : \forall X.T[\Sigma]} \quad \dfrac{\forall X.T[\Sigma] \preceq S[\Sigma]}{\Gamma[\Sigma], x : \forall X.T[\Sigma] \vdash x : S[\Sigma]}\ \text{coerceall}}{\dfrac{\Gamma[\Sigma] \vdash t : T[\Sigma]}{\Gamma[\Sigma] \vdash t : S[\Sigma]}\ \cancel{\text{cut}_1}}\ \text{cut}_2$$

This fails because the subtyping judgment $\forall X.T[\Sigma] \preceq S[\Sigma]$ is not derivable under the defined coercion rules.

If instead the coerceall rule is eliminated first:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi}{\Gamma \vdash t : \forall X.T}\ \text{tpadd}}{\Gamma, X \vdash t : \forall X.T}\ \text{T-TApp}}{\Gamma, X \vdash t : T[X/X]} \quad \dfrac{T \preceq S}{\Gamma, x : T \vdash x : S}\ \text{coercesub}}{\Gamma, X \vdash t : S}\ \text{cut}_1$$

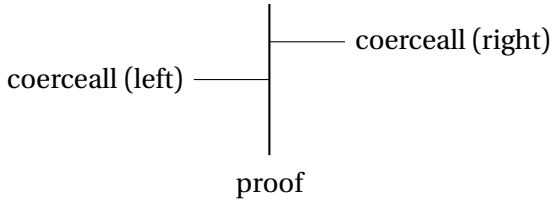
Now, we can eliminate the coercesub rule as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi[\Sigma]}{\Gamma[\Sigma] \vdash t : \forall X.T[\Sigma]}\ \text{tpadd}}{(\Gamma, X)[\Sigma] \vdash t : \forall X.T[\Sigma]}\ \text{T-TApp}}{(\Gamma, X)[\Sigma] \vdash t : T[X/X][\Sigma]}\ \cancel{\text{cut}_1}}{(\Gamma, X)[\Sigma] \vdash t : S[\Sigma]}$$

This yields a valid derivation, demonstrating the order that coercion rules are eliminated matters.

Now that we know all coerceall rules must be eliminated before the coercesub rules, the next question is whether we should eliminate the coerceall rules occurring on the left first or those occurring on the right.

Coerceall rules may appear in different positions within the proof:



Removing a left coerceall alters the proof above it and only when the cut variable must be pulled out from the context. In contrast, a right coerceall only affects the proof locally : the corresponding cut is replaced by a T-TApp step. In both cases, any coercesub steps in the proof remain unaffected.

Therefore, to avoid interference between multiple coerceall instances, one must eliminate them starting from the leaves of the proof and proceed upward toward the root. This ensures that each removed coerceall does not affect other coerceall instances that have yet to be eliminated.
Once all coerceall rules have been removed, the remaining coercions in the proof are all of the form coercesub. These can then be eliminated in any order, which is proved in Lemma 3.4.5.

Lemma 3.4.5. Coercesubs can be eliminated in any order.

Proof. We prove this, by induction on the number of coercesub steps. For the base case, a single coercesub can be eliminated directly, as proved in Lemma 3.4.2. For the inductive step, assume that any proof containing k coercesub steps can be transformed into a coercion-free proof. Consider a proof with $k+1$ coercesub steps. Removing one coercesub introduces a substitution into the remaining proof. The remaining k coercions are preserved under this substitution, and by the inductive hypothesis, the resulting proof with k substitutions can be transformed into a coercion-free derivation. Therefore, by induction, any proof containing coercesub steps can be rewritten into a valid coercion-free proof.

□

Remark 3.4.6. Given a subsumption $A \preceq A'$, a typing derivation $\Gamma, x : A' \vdash t : T$, and a term s such that $\Gamma \vdash s : A$, the judgment

$$\Gamma \vdash t[s/x] : T$$

is derivable using a coercion from A to A':

(1) By coercing the variable:

$$\dfrac{\dfrac{\Pi'}{\Gamma \vdash s : A} \qquad \dfrac{\dfrac{\dfrac{A \preceq A'}{x : A \vdash x : A'}\ \text{coerce} \qquad \dfrac{\Pi}{\Gamma, x : A' \vdash t : T}}{\Gamma, x : A \vdash t : T}\ \text{cut}}{\Gamma \vdash t[s/x] : T}\ \text{cut}$$

(2) By coercing the term:

$$\dfrac{\dfrac{\dfrac{\Pi'}{\Gamma \vdash s : A} \qquad \dfrac{\dfrac{A \preceq A'}{\Gamma \vdash s : A'}\ \text{coerce}}{\Gamma \vdash s : A'}\ \text{cut} \qquad \dfrac{\Pi}{\Gamma, x : A' \vdash t : T}}{\Gamma \vdash t[s/x] : T}\ \text{cut}}$$

This way of interpreting coercions provides an important guarantee: when $A \preceq A'$, it is safe to substitute a term s of type A into a variable of type A' [3].

Next, we discuss how to manipulate coercion rules. This is important because type equations, discussed in the next chapter, are more general forms of coercions introduced in this chapter. In particular, coercions have so far only involved arrow types, whereas type equations use more general type constructors that include arrow types as a special case.

Lemma 3.4.7. The following holds for subsumptions:

(i) Every type subsumes itself (Reflexivity):

$$A \preceq A$$

(ii) Subsumptions are closed under transitivity (Transitivity):

$$\frac{A_1 \preceq A_2, A_2 \preceq A_3}{A_1 \preceq A_3}\ \text{cut}$$

(iii) The subsumption relation is not anti-symmetric (Non-antisymmetry):

If $A_1 \preceq A_2$ and $A_2 \preceq A_1$, it does not necessarily follow that $A_1 = A_2$.

(iv) The contravariant rule for the arrow type [26]:

$$\frac{A' \preceq A, \quad T \preceq T'}{A \to T \preceq A' \to T'}\ \text{arrow}$$

Proof. Each case of the rules above can be verified, as follows.

(i) The reflexitivity is immediate.

(ii) This can be proved by the following derivation, which shows that the subsumption $A_1 \preceq A_3$ can be mimicked using the premises $A_1 \preceq A_2$ and $A_2 \preceq A_3$.

$$\frac{A_1 \preceq A_3}{t : A_1 \vdash t : A_3}\ \text{coerce} \qquad \Rightarrow \qquad \frac{\dfrac{A_1 \preceq A_2}{t : A_1 \vdash t : A_2}\ \text{coerce} \quad \dfrac{A_2 \preceq A_3}{y : A_2 \vdash y : A_3}\ \text{coerce}}{t : A_1 \vdash t : A_3}\ \text{cut}$$

(iii) As a counterexample to anti-symmetry, consider

$$\forall A.(A \to \text{Char}) \preceq (\forall A.A) \to \text{Char}, (\forall A.A) \to \text{Char} \preceq \forall A.(A \to \text{Char})$$

although

$$\forall A.\,(A \to \text{Char}) \ \neq\ (\forall A.\,A) \to \text{Char}.$$

(iv) The subsumption $A \to T \preceq A' \to T'$ means (see Definition 3.3.1):

$$\frac{\Pi}{\Gamma \vdash t : A \to T} \quad \Rightarrow \quad \frac{\Pi'}{\Gamma \vdash t : A' \to T'}$$

If one proves this using the assumptions $A' \preceq A$ and $T \preceq T'$, then the rule arrow can be derived.

The derivation proceeds as follows:

$$\dfrac{\dfrac{\dfrac{A' \preceq A}{x : A' \vdash x : A}\ \text{coerce} \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash t : A \to T}{\Gamma \vdash \lambda x.t\ x : A \to T}\ \eta}{\Gamma, x : A \vdash t\ x : T}\ \text{abs} \qquad \dfrac{T \preceq T'}{y : T \vdash y : T'}\ \text{coerce}}{\Gamma, x : A \vdash t\ x : T'}\ \text{cut}}{\Gamma, x : A' \vdash t\ x : T'}\ \text{cut}}{\dfrac{\Gamma \vdash \lambda x.t\ x : A' \to T'}{\Gamma \vdash t : A' \to T'}\ \eta}\ \text{abs}$$

□

To illustrate how a subsumption relation between two arrow types (i.e., $A \to B \preceq C \to D$) can be broken we used the η-rule (see Lemma 3.4.7).

In Haskell's main compiler (GHC), this rule does not generally hold, since $\lambda x.\, t\ x$ and t may behave differently in the presence of side effects. However, the η-rule can be enabled explicitly by using the `-XDeepSubsumption` language extension [27].

Here is an example of when $\lambda x.\, t\ x$ behaves differently from t, in Haskell:

```
f1, f2 :: Int -> Int
f1 = error "urk"
f2 = \x -> f1 x

main = do
  print (f1 `seq` True) -- calls error "urk"
  print (f2 `seq` True) -- returns True
```

Here, evaluating `f1 ‘seq‘ True` immediately triggers the error, whereas `f2 ‘seq‘ True` returns `True` because `f2` is a lambda abstraction and already in weak head normal form.

When the η-rule is not valid, the proofs showing how the subsumption relation between arrow types can be decomposed no longer hold (see Lemma 3.4.7).

3.5 Summary

In this chapter, we modified system F to obtain a version more suitable for programming languages, which we call system S. We introduced subsumption and showed how it can be incorporated as coercion rules within system S. We then proved that any derivation using coercions remains a valid proof in system S, if coercions are removed. Finally, we discussed the implied coercion rules, such as reflexivity, transitivity, non-antisymmetry, and contravariance for function types.

Chapter 4

Type Equations

In this chapter, a general system of type equations is introduced with the rules for manipulating them. Type equations are built up from the types used within these equations. We then show how to generate equations from terms, and how to collect these equations. To show this equation generation makes sense, we prove that if a term can be type-checked in system $F_{\eta+\text{annotation}}$ with a given type, then this type provides a solution that solves the generated type equations for that term.

4.1 Type equations

To define the general syntax of type equations, the syntax of types used in the system is specified. We then define the syntax of equations, and include a discussion of quantifiers. Finally, we describe how to reason about these equations.

4.1.1 Type equation syntax

Let Ω is a set of type constructors with arity $: \Omega \rightarrow [\{+,-\}]$, where the arity assigns to each type constructor a list of variances indicating how it behaves with respect to its arguments. Thus, arity$(F) = [v_1, \ldots, v_n]$, where each $v_i \in \{+,-\}$ indicates the variance of the i^{th} argument: + denotes that the argument is covariant, while − denotes that it is contravariant. This arity list, in particular, tells us how subsumption relations between argument types affect the subsumption of the entire type constructed with F.

The types of our system are defined inductively by:

$$
\begin{array}{rll}
T ::= & X & \text{(type variable)} \\
& F(T_1, \ldots, T_n), F \in \Omega & \text{(type constructor)} \\
& \forall X.\, T & \text{(universal type)}
\end{array}
$$

Figure 4.1.1. Type(Ω, arity)

We let $\forall X, Y.\, T$ as usual denote $\forall X.\, (\forall Y.\, T)$.

Based on the definition for Type(Ω, arity) (see Figure 4.1.1), we can define the type equations inductively:

$$
\begin{array}{rll}
\text{Equ} ::= & \text{Equ}, \text{Equ} & \text{(conjuction)} \\
& \underline{\exists} X.\text{Equ} & \text{(existentials)} \\
& \underline{\forall} X.\text{Equ} & \text{(foralls)} \\
& T = S & \text{(equality)} \\
& T \preceq S & \text{(subsumption)}
\end{array}
$$

Figure 4.1.2. Type equation, Eqn (Ω, arity)

In the above inductive definition, T and S are types, and X is a type variables.

We use the contravariant rule (see Lemma 3.4.7), in a general form, for type constructors $F \in \Omega$:

$$
\frac{\text{arity}(F) = [\nu_1, \ldots, \nu_n] \qquad T_1 \preceq_{\nu_1} T_1', \quad \ldots, \quad T_n \preceq_{\nu_n} T_n'}{F(T_1, \ldots, T_n) \preceq F(T_1', \ldots, T_n')}\ \text{cons}
$$

where $\preceq_+ = \preceq$, $\preceq_- = \succeq$.

This means that F is monotone with respect to variance. The assumption that type constructors are jointly monic with respect to variance, gives the following two way rule:

$$
\frac{\text{arity}(F) = [\nu_1, \ldots, \nu_n] \qquad T_1 \preceq_{\nu_1} T_1', \quad \ldots, \quad T_n \preceq_{\nu_n} T_n'}{F(T_1, \ldots, T_n) \preceq F(T_1', \ldots, T_n')}\ \text{cons}
$$

Example 4.1.3. The following examples show the application of the cons rule:

(1) The function type constructor $\rightarrow$ has arity $(\rightarrow) = [-, +]$, as it is contravariant in its first argument (the input type) and covariant in its second argument (the output type). This implies that:

$$
\frac{\text{arity}(\rightarrow) = [-, +] \qquad B \preceq A, A' \preceq B}{A \rightarrow A' \preceq B \rightarrow B'}\ \text{cons}
$$

which is the arrow rule (see Lemma 3.4.7).

(2) The list type constructor List has arity (List) = [+], meaning it is covariant in its single argument.

$$\frac{\text{arity(List)} = [+] \qquad A \preceq A'}{\text{List}(A) \preceq \text{List}(A')}\ \text{cons}$$

Example 4.1.4. Assume $\Omega = \{F, G\}$ with arities $\text{arity}(F) = [+, -, +]$ and $\text{arity}(G) = [-]$. Below are type equations, constructed using the rules in Figure 4.1.2.

(1) $X = Y$

(2) $F(X, Y, Z) = G(Y)$

(3) $\underline{\forall} X, Y, Z.\ (X = F(X, Y, Z))$

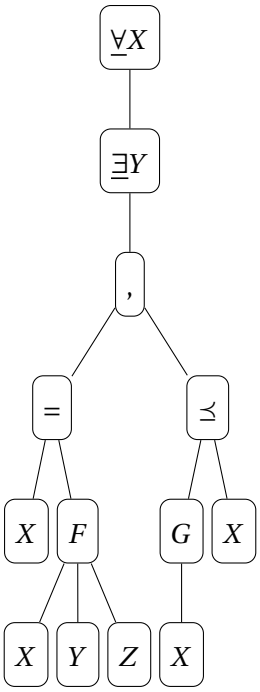
(4) $\underline{\exists} Y, Y', Y''.\ F(Y, Y', Y'') \preceq G(Y')$

(5) $\underline{\forall} X.\underline{\exists} Y.\ (X = F(X, Y, Z),\ G(X) \preceq X)$

An equation is closed, if all of its type variables are bound. In the examples above, equations (3) and (4) are closed.

We may represent the structure of equations diagrammatically as trees. The nodes correspond to conjunctions (‘,‘, which have two children), underlined quantifiers used in equations ($\underline{\forall}$ and $\underline{\exists}$, each with one child), relations (such as equality and subsumption, each with two children), type constructors (with the number of children determined by their arity), and the standard $\forall$ quantifier used in types (which has one child), while the leaves represent type variables.

The following is the syntax tree presentation for equation (5) in Example 4.1.4 :



4.1.2 Quantifiers

In the system of equations described above, three kinds of quantifiers are used: the basic universal quantifier $\forall$, which appears in type construction, and the underlined quantifiers $\underline{\forall}$ and $\underline{\exists}$, which are used in the formation of equations.

The standard rules for manipulating quantified equations, from predicate logic, holds in this system:

(i) $\underline{\forall}A.\, E(A),\ E'(A) \ \equiv\ \underline{\forall}A.\, E(A),\ \underline{\forall}A.\, E'(A)$

(ii) $\underline{\forall}A.\, E \ \equiv\ E \quad \text{if } A \notin \text{fv}(E)$

(iii) $\underline{\forall}A.\, \underline{\forall}B.\, E(A,B) \ \equiv\ \underline{\forall}B.\, \underline{\forall}A.\, E(A,B)$

(iv) $\underline{\exists}X.\, (E(X), F) \ \equiv\ (\underline{\exists}X.\, E(X)), F \quad \text{if } X \notin \text{fv}(F)$

(v) $\underline{\exists}X.\, E \ \equiv\ E \quad \text{if } X \notin \text{fv}(E)$

(vi) $\underline{\exists}A.\, \underline{\exists}B.\, E(A,B) \ \equiv\ \underline{\exists}B.\, \underline{\exists}A.\, E(A,B)$

Note that, when we write an expression like $E(A)$, we are indicating that the variable A may appear in E. Rule (iii), states that underlined quantifiers $\underline{\forall}$ can be reordered only if variable bindings are preserved. For example:

$$\underline{\forall}B.\, \underline{\forall}A.\, (B = \text{Char},\ A = B) \ \equiv\ \underline{\forall}A.\, \underline{\forall}B.\, (B = \text{Char},\ A = B)$$

but

$$\underline{\forall} B.\,(B = \text{Char},\ \underline{\forall} A.\,A = B) \ \not\equiv\ \underline{\forall} A.\,(B = \text{Char},\ \underline{\forall} B.\,A = B)$$

since the binding of B changes.

Similarly, rule (vi) applies to $\underline{\exists}$: reordering is valid only when variable bindings are preserved.

The $\underline{\forall}$ quantifiers *cannot* be moved across $\underline{\exists}$ quantifiers, since this may change the meaning of the equation: existential variables may depend on universally quantified variables.

Thus,

$$\underline{\forall} B.\,\underline{\exists} A.\,(E(B),\ E'(A,B)) \quad \not\equiv \quad \underline{\exists} A.\,\underline{\forall} B.\,(E(B),\ E'(A,B))$$

4.1.3 Manipulating type equations

We now describe how type equations can be manipulated.

Definition 4.1.5. The inference rules for manipulating equations are defined as follows:

(i) Rule for conjunction:

$$\frac{E, E'}{E}\ \text{conj}$$

(ii) Type constructors:

$$\frac{\text{arity}(F) = [\nu_1, \ldots, \nu_n] \qquad T_1 \preceq_{\nu_1} T_1', \quad \ldots, \quad T_n \preceq_{\nu_n} T_n'}{F(T_1, \ldots, T_n) \preceq F(T_1', \ldots, T_n')}\ \text{cons}$$

For type constructors, the rule is a general form of the contravariant rule for arrows, which was discussed in Lemma 3.4.7: the assumption of being monomorphic makes it a two way rule.

(iii) Pulling out $\forall$ on the left side of a subsumption relation:

$$\frac{\underline{\exists} X.(T \preceq T')}{(\forall X.T) \preceq T'}\ \text{spec}$$

Here, the notation $\underline{\exists}$ indicates that the variable X may be instantiated. The rule is justified by interpreting $\preceq$ as entailment $\vdash$; under this interpretation, a universally quantified type on the left can only be introduced by instantiating the bound variable [28].

(iv) Pulling out $\forall$ on the right side of a subsumption relation:

$$\frac{\underline{\forall} X.(T \preceq T')}{T \preceq (\forall X.T')} \text{ gen}$$

Here, the notation $\underline{\forall}$ indicates that the variable X is not allowed to be instantiated. The rule is justified, by interpreting $\preceq$ as entailment $\vdash$; under this interpretation, a universally quantified type on the right can only be introduced by moving the bound variable from the context (i.e., left of entailment) [28]. Thus, it is a variable in the context which cannot be substituted.

4.2 Generating equations

The syntax of the source language we are working with is shown in Figure 4.2.1. This syntax is based on system S. Figure 4.2.1 defines the syntax of terms (t) and types (T) in the language. Terms include variables, lambda abstractions, applications, type annotations, and let-bindings. Types consist of type variables, type constructors which can be applied to multiple types, and universally quantified types.

t ::=	x	(variable)
	λx.t	(abstraction)
	t t	(application)
	t :: T	(annotation)
	let $x = u$ in t	(let)
T ::=	X	(type variable)
	$F(T_1, \ldots, T_n), F \in \Omega$	(type constructor)
	$\forall X$. T	(universal type)

Figure 4.2.1. Syntax of the source language

In Figure 4.2.2, which shows the generating equations system, each judgment is annotated with type equations between types. During type inference, these equations are collected and later solved. Before the solving stage, the generated equations must be $\alpha-$renamed to prevent name conflicts between bound variables.

At each node in the proof tree, fresh type variables are introduced together with equations relating them to existing type variables. We record this as

$$\underline{\exists} X, \ldots, X_n.\ E_1, \ldots, E_m,$$

where the newly introduced variables X_i, $i \in 1, \ldots, n$, are bound on the left, and the equations E_j, $j \in 1, \ldots, n$, are listed on the right. The type assigned to the term at each step is called that term's principal type. This type is represented by a free type variable, Q, in the collected equations from that point onward. This ensures that the only free type variable when collecting equations for a term is that term's principal type.

Note that, to ensure we always obtain some type $T \preceq Q$, the first equation we collect is:

$$\frac{\overline{\vdash t : Q' \langle E_1 \rangle}}{\vdash t : Q \quad \langle \underline{\exists} Q'.Q' \preceq Q, E_1 \rangle}$$

where Q' is the principal type as it corresponds to a term, while Q is called the wrapper type and is the actual type of the term. After collecting this first equation we continue to apply the rules in Figure 4.2.2 to generate the equations of a term. By successively collecting the equations throughout the derivation, and then eliminating the existentially bound variables by substitution, one obtains the principal type of the entire term. In the proof of theorem 4.3.1, we rely on the basic rules for solving equations, namely that $T = T$ and $T \preceq T$ always hold. However, equation solving in general is more complex, and a detailed description of the solving procedure is given in Chapter 5.

$$\frac{}{\Gamma, x : A \vdash x : Q \quad \langle A \preceq Q \rangle}\ \text{var}$$

$$\frac{x : A, \Gamma \vdash t : B \quad \langle E_1 \rangle}{\Gamma \vdash \lambda x.t : Q \quad \langle \underline{\exists} A, B.Q = A \to B, E_1 \rangle}\ \text{abs}$$

$$\frac{\Gamma \vdash t : A \quad \langle E_1 \rangle \quad \Gamma \vdash u : B \quad \langle E_2 \rangle}{\Gamma \vdash t\ u : Q \quad \langle \underline{\exists} A, B.A = B \to Q, E_1, E_2 \rangle}\ \text{app}$$

$$\frac{\Gamma, x : A \vdash t : B \quad \langle E_1 \rangle \quad \Gamma_C \vdash u : C \quad \langle E_2 \rangle}{\Gamma \vdash \text{let}\ x = u\ \text{in}\ t : Q \quad \langle \underline{\exists} A, B, C.A = C, B = Q, E_1, E_2 \rangle}\ \text{let}$$

$$\frac{\Gamma \vdash t : B \quad \langle E_1 \rangle}{\Gamma \vdash (t :: A) : Q \quad \langle \underline{\exists} B.B = Q, A = Q, E_1 \rangle}\ \text{annotation}$$

Figure 4.2.2. Generating equations system

4.3 Correspondence to system F_η+annotation

Since type assignment in system S is undecidable [4], type inference for higher-rank types using the rules in Figure 4.2.2 is also undecidable. However, if a term with a known type can be type checked in system $F_{\eta\text{+annotation}}$, then that type must satisfy all of the generated type equations for that term.

Theorem 4.3.1. If $t : A$ can be proved in system $F_{\eta\text{+annotation}}$, the type equations generated for $\tilde{t}$ (see Definition 3.2.2) in system S using the rules in Fig 4.2.2 can be solved, with the type of t substituted with A.

Proof. To verify that Theorem 4.3.1 holds for every rule, we perform structural induction over the system $F_{\eta\text{+annotation}}$ type checking rules given in Figure 3.1.8.

The system $F_{\eta\text{+annotation}}$ type checking rules are placed in parallel with the generating equations system:

$$\frac{\Gamma \vdash t_1 : A_1 \quad \Gamma \vdash t_2 : A_2}{\Gamma \vdash t : A_0}$$

System $F_{\eta\text{+annotation}}$

$$\frac{\Gamma \vdash \tilde{t_1} : Q_1\langle E_1 \rangle \quad \Gamma \vdash \tilde{t_2} : Q_2\langle E_2 \rangle}{\Gamma \vdash \tilde{t} : Q\langle E, E_1, E_2 \rangle}$$

generating equations system

Therefore, at each step of the induction, we assume that $E_1[A_1/Q_1]$ and $E_2[A_2/Q_2]$ are valid, and we need to show that $E[A_0/Q, A_1/Q_1, A_2/Q_2]$ is also valid.

T-Var$_F$: If $y : A$ type checks in system $F_{\eta\text{+annotation}}$:

$$\frac{y : A \in \Gamma}{\Gamma \vdash y : A}\ \text{T-Var} \qquad\qquad \frac{}{\Gamma, y : A \vdash y : Q \quad \langle A \preceq Q \rangle}\ \text{var}$$

System $F_{\eta\text{+annotation}}$ generating equations system

we need to show that $\langle A \preceq Q \rangle [A/Q]$ holds:

$$\langle A \preceq Q \rangle [A/Q] \Rightarrow \langle A \preceq A \rangle$$

The equation $\langle A \preceq A \rangle$ is trivially satisfied, so this case serves as the base case in the inductive argument.

T-Annot$_F$: If system $F_{\eta\text{+annotation}}$ type checks A for the term $t :: A$:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash (t :: A) \ : A}\ \text{T-Annot} \qquad\qquad \frac{\Gamma \vdash t : B \ \langle E_1 \rangle}{\Gamma \vdash (t :: A) \ : Q \quad \langle \underline{\exists} B.B = Q, A = Q, E_1 \rangle}\ \text{annotation}$$

System $F_{\eta\text{+annotation}}$ generating equations system

inductively $E_1[A/B]$ is solvable, so we need to show $\langle \underline{\exists} B.B = Q, A = Q \rangle [A/B, A/Q]$ holds:

$$\langle \underline{\exists} B.B = Q, A = Q \rangle [A/Q, A/B] \Rightarrow \langle A = A, A = A \rangle$$

The equation $\langle A = A \rangle$ is trivially satisfied, so this case holds.

T-Abs$_F$: If system $F_{\eta\text{+annotation}}$ type checks $\lambda x : A.t : A \to A'$:

$$\frac{x : A, \Gamma \vdash t : A'}{\Gamma \vdash \lambda x : A.t : A \to A'}\ \text{T-Abs}_F \qquad\qquad \frac{x : A, \Gamma \vdash t : B \ \langle E_1 \rangle}{\begin{array}{c}\Gamma \vdash \lambda x.t : Q \\ \langle \underline{\exists} A, B.Q = A \to B, E_1 \rangle\end{array}}\ \text{abs}$$

System $F_{\eta\text{+annotation}}$ generating equations system

inductively $E_1[A'/B]$ is solvable, then we need to show $\langle \underline{\exists} A, B.Q = A \to B \rangle [A'/B, (A \to A')/Q]$ holds:

$$\langle \underline{\exists} A, B.Q = A \to B \rangle [A'/B, (A \to A')/Q] \Rightarrow \langle A \to A' = A \to A' \rangle$$

The equation $\langle A \to A' = A \to A' \rangle$ is trivially satisfied, so this case holds.

T-App$_F$: If $t\ u : Q'$ can get type checked in system F using the T-App$_F$ rule:

$$\frac{\Gamma \vdash t : A' \to A \quad \Gamma \vdash u : A'}{\Gamma \vdash t\ u : A}\ \text{T-App}_F$$

System $F_{\eta\text{+annotation}}$

$$\frac{\Gamma \vdash t : B\ \langle E_1 \rangle \quad \Gamma \vdash u : B'\ \langle E_2 \rangle}{\Gamma \vdash t\ u : Q}\ \text{app}$$
$$\langle \underline{\exists} B, B'.B = B' \to Q, E_1, E_2 \rangle$$

generating equations system

inductively $E_1[A' \to A/B], E_2[A'/B']$ is solvable, so we need to show $\langle \underline{\exists} B, B'.B = B' \to Q \rangle [A/Q, A' \to A/B, A'/B']$ holds:

$$\langle \underline{\exists} B, B'.B = B' \to Q \rangle [A/Q, A' \to A/B, A'/B'] \Rightarrow \langle A' \to A = A' \to A \rangle$$

The equation $\langle A' \to A = A' \to A \rangle$ is trivially satisfied, so this case holds.

T-TApp$_F$: If $t[A] : T[A/X]$ can get type-checked in system $F_{\eta\text{+annotation}}$, then :

$$\frac{\Gamma \vdash t : \forall X.T \quad \Gamma \vdash_{\text{type}} A}{\Gamma \vdash t[A] : T[A/X]}\ \text{T-TApp}_F$$

System $F_{\eta\text{+annotation}}$

$$\frac{\Gamma \vdash t : \forall X.T \quad \langle E_1 \rangle}{\Gamma \vdash t : T[A/X]}$$
$$\langle (\forall X.T) \preceq T[A/X], E_1 \rangle$$

generating equations system

inductively $E_1[\forall X.T/\forall X.T]$ is solvable, so we need to show $\langle (\forall X.T) \preceq T[A/X] \rangle$ holds:

$$\frac{\Gamma \vdash t : \forall X.T \quad \dfrac{\forall X.T \preceq T[A/X]}{x : \forall X.T \vdash x : T[A/X]}\ \text{coerce}}{\Gamma \vdash t : T[A/X]}\ \text{cut}$$

T-TAbs$_F$: If $\Lambda X.t : \forall X.A$ can be type-checked in System F:

$$\frac{\Gamma, X \vdash t : A}{\Gamma \vdash \Lambda X.t : \forall X.A}\ \text{T-TAbs}_F$$

System $F_{\eta\text{+annotation}}$

$$\frac{\Gamma, X \vdash t : A \qquad \langle E_1 \rangle}{\Gamma \vdash t : \forall X.A}$$
$$\langle A \preceq \forall X.A, E_1 \rangle$$

generating equations system

inductively $E_1[A/A]$ is solvable, so we need to show $\langle A \preceq \forall X.A \rangle$ holds, using definition 4.1.5:

$$\frac{\underline{\forall} X.(A \preceq A)}{A \preceq (\forall X.A)}\ \text{gen}$$

and $\underline{\forall} X.(A \preceq A)$ holds because $(A \preceq A)$ is valid. Consequently, we obtain $\underline{\forall} X.()$, which is trivially true.

□

Example 4.3.2. Consider the following term:

$$\text{id} : \forall X.X \to X \vdash (\text{id True},\ \text{id } `A`)$$

Collecting the equations using the generating equation rules:

$$\dfrac{\dfrac{\dfrac{}{\substack{\text{id} : \forall X.X \to X \vdash \text{id} : 4 \\ \langle \forall X.X \to X \preceq 4 \rangle}}\ \text{var} \quad \dfrac{}{\substack{\text{id} : \forall X.X \to X \vdash `A` : 5 \\ \langle \text{Bool} \preceq 5 \rangle}}\ \text{var}}{\substack{\text{id} : \forall X.X \to X \vdash \text{id True} : 2 \\ \langle \underline{\exists} 4, 5.4 = 5 \to 2 \rangle}}\ \text{app} \quad \dfrac{\dfrac{}{\substack{\text{id} : \forall X.X \to X \vdash \text{id} : 6 \\ \langle \forall X.X \to X \preceq 6 \rangle}}\ \text{var} \quad \dfrac{}{\substack{\text{id} : \forall X.X \to X \vdash \text{True} : 7 \\ \langle \text{Char} \preceq 7 \rangle}}\ \text{var}}{\substack{\text{id} : \forall X_3.X_3 \to X_3 \vdash \text{id } `A` : 3 \\ \langle \underline{\exists} 6, 7.6 = 7 \to 3 \rangle}}\ \text{app}}{\substack{\text{id} : \forall X.X \to X \vdash (\text{id True},\ \text{id } `A`) : 1 \\ \langle \underline{\exists} 2, 3.1 = (2,3) \rangle}}\ \text{tuple}$$

$$\dfrac{}{\substack{\text{id} : \forall X.X \to X \vdash (\text{id True},\ \text{id } `A`) : 0 \\ \langle \underline{\exists} 1.1 \preceq 0 \rangle}}$$

The collected equations are:

$$\Big\langle \underline{\exists} 1.\ 1 \preceq 0,$$
$$\quad \Big\langle \underline{\exists} 2, 3.\ 1 = (2,3),$$
$$\quad\quad \Big\langle \underline{\exists} 4, 5.\ 4 = 5 \to 2,\ \langle \forall X.\ X \to X \preceq 4, \quad \text{Bool} \preceq 5 \rangle,$$
$$\quad\quad \Big\langle \underline{\exists} 6, 7.\ 6 = 7 \to 3,\ \langle \forall X.\ X \to X \preceq 6, \quad \text{Char} \preceq 7 \rangle$$
$$\quad \Big\rangle$$
$$\Big\rangle$$

As mentioned before, we must α-rename the bound variables in this equation to avoid variable name clashes. Thus, we rename X to X_0 and X_1 in the corresponding quantified expressions.

$$\Big\langle \underline{\exists} 1.\ 1 \preceq 0,$$
$$\quad \Big\langle \underline{\exists} 2, 3.\ 1 = (2,3),$$
$$\quad\quad \Big\langle \underline{\exists} 4, 5.\ 4 = 5 \to 2,\ \langle \forall X_0.\ X_0 \to X_0 \preceq 4, \quad \text{Bool} \preceq 5 \rangle,$$
$$\quad\quad \Big\langle \underline{\exists} 6, 7.\ 6 = 7 \to 3,\ \langle \forall X_1.\ X_1 \to X_1 \preceq 6, \quad \text{Char} \preceq 7 \rangle$$
$$\quad \Big\rangle$$
$$\Big\rangle$$

4.4 Summary

In this chapter, we discussed how to manipulate type equations, which sets the foundations for defining the rules of type inference. We also established an important guarantee: the equations generated for any term are sound with respect to system $F_{\eta+\text{annotation}}$. That is, terms that can be type-checked in system $F_{\eta+\text{annotation}}$ have types that provide a solution to the equations they generate.

Chapter 5

Valid Quantified Substitutions

A solution for the generated equations from a term is given by a list of quantified substitutions which must be valid. In this chapter, we define how such lists of substitutions are constructed and what it means for them to be valid. To set this up, we define context with a hole and for such a hole we show how to track varity and determine which variables are visible. Finally, we show that a valid substitution, which solves the equations generated by a term, can be used to build a derivation in system S for that term. This is achieved by substituting each step of the equation generation to reconstitute the corresponding rule being used from system S.

5.1 Tracking varity

Varity describes whether a type or equation occurs in a covariant/positive or contravariant/negative quantification. We compute varity by applying the varity annotation rules to the structure of the equation, starting with a positive varity (+) at the top and propagating it through the constructors according to the rules defined in Figure 5.1.2 and Figure 5.1.3. Although varity is conceptually simple, it takes some effort to formalize it precisely.

Figure 5.1.2 shows how varity is propagated through type expressions, based on the type formation rules in Figure 4.1.1. The universal quantifier $\forall$ passes its varity unchanged to its body. For type constructors, each argument position has a fixed arity (covariant or contravariant). The varity at the constructor node is multiplied by these arities to determine the varity of each argument. That is, if the constructor has varity v and the i-th argument position has varity v_i, then the i-th argument is annotated with $v \cdot v_i$,

based on the Figure 5.1.1:

$\cdot$	$+$	$-$
$+$	$+$	$-$
$-$	$-$	$+$

Figure 5.1.1. Sign multiplication table

$$\frac{}{\underline{X_\nu}}\ \text{type variable}$$

$$\frac{(\forall X.T)_\nu}{T_\nu}\ \text{universal type}$$

$$\frac{(F(T_1,\ldots,T_n))_\nu \quad \text{arity}(F) = [\nu_1,\ldots,\nu_n]}{(T_1)_{\nu.\nu_1} \quad \ldots \quad (T_n)_{\nu.\nu_n}}\ \text{type constructor}$$

Figure 5.1.2. Varity annotation based on type formation rules - proof search direction

Figure 5.1.3 shows how varity is propagated through equations. For compound equations, such as conjunctions and quantifiers, the same varity is passed down to each sub-equation. For subsumption, the left-hand side is treated contravariantly (the varity is flipped), and the right-hand side is treated covariantly (the varity is preserved). Note that equality is not present because $A = B$ is treated as $A \preceq B, B \preceq A$.

$$\frac{\text{Equ}, \text{Equ'}_\nu}{\text{Equ}_\nu \quad \text{Equ'}_\nu}\ \text{conjunction}$$

$$\frac{(\underline{\exists} X.\text{Equ})_\nu}{\text{Equ}_\nu}\ \text{existentials}$$

$$\frac{(\underline{\forall} X.\text{Equ})_\nu}{\text{Equ}_\nu}\ \text{foralls}$$

$$\frac{(T \preceq S)_\nu}{(T)_{-.\nu} \quad (S)_{+.\nu}}\ \text{subsumption}$$

Figure 5.1.3. Varity annotation based on equation formation rules - proof search direction

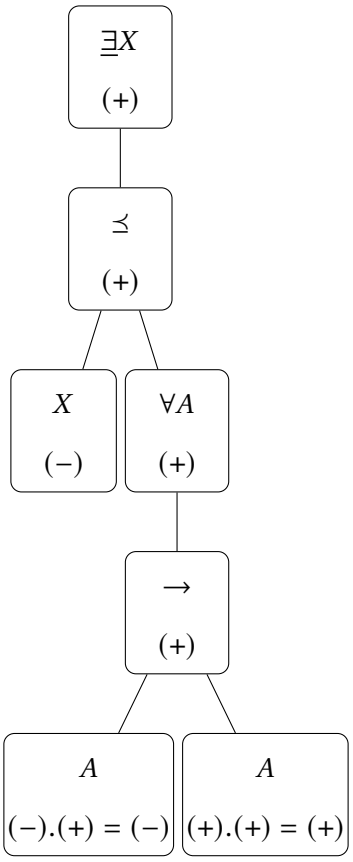
Example 5.1.4. If we apply the varity annotation rules to the following equation:

$$\underline{\exists} X.\ X \preceq \forall A.\ A \to A$$

the varities are determined as follows:

$$\dfrac{\dfrac{\dfrac{(\underline{\exists} X.\ X \preceq \forall A.\ A \to A)_+}{(X \preceq \forall A.\ A \to A)_+}\ \text{existentials}}{\dfrac{(X)_-}{}\qquad \dfrac{(\forall A.\ A \to A)_+}{\dfrac{(A \to A)_+ \quad \text{arity}(\to) = [-,+]}{\dfrac{(A)_-}{}\ \text{type variable}\qquad \dfrac{(A)_+}{}\ \text{type variable}}\ \text{type constructor}}\ \text{universal type}}\ \text{subsumption}}$$

The following is the syntax tree presentation of the equation above:



5.2 Context with holes

A type context with a hole is a type expression containing a single hole where one can insert a type. A type context with a hole is denoted as $(C[\![\_]\!] \quad \text{Type})$. When a type, T, is inserted into this position, we

write $(C[\![T]\!] \quad \text{Type})$.

Figure 5.2.1 presents the rules for constructing type contexts within type expressions. The hole type rule is the basic case, where the context is just the hole. The universal type rule extends a context by wrapping it with a universal quantifier $\forall X$. The type constructor rule places the context as an argument of the type constructor.

$$\frac{}{C[\![\_]\!] \quad \text{type}} \text{ hole type}$$

$$\frac{C[\![\_]\!] \quad \text{type}}{\forall X.C[\![\_]\!] \quad \text{type}} \text{ universal type}$$

$$\frac{\text{arity}(F) = [v_1, \ldots, v_n] \quad (C[\![\_]\!] \quad \text{type})}{F(\ldots, C[\![\_]\!], \ldots) \quad \text{type}} \text{ type constructor}$$

Figure 5.2.1. Context formation rules for types

Similarly, an equation context with a hole is an equation with a single hole where one can insert an equation. This hole is denoted as $(C[\![\_]\!] \quad \text{eqn})$. When an equation Equ is inserted into the hole, we write $(C[\![\text{Equ}]\!] \quad \text{eqn})$. The rules for constructing equation contexts are presented in Figure 5.2.2. The equation rule include the base case of an empty context, the conjunction rule adds another equation with a comma, the existentials and foralls rules wrap the context in an underlined quantifier. The lsubsumption rule places a type context on the left side of a subsumption, while the rsubsumption rule places it on the right, with both forming an equation context.

$$\frac{}{C[\![\_]\!] \quad \text{eqn}} \text{ equation}$$

$$\frac{C[\![\_\,]\!] \quad \text{eqn}}{\text{Equ},\ C[\![\_]\!] \quad \text{eqn}} \text{ conjunction}$$

$$\frac{C[\![\_\,]\!] \quad \text{eqn}}{\underline{\exists} X.C[\![\_]\!] \quad \text{eqn}} \text{ existentials}$$

$$\frac{C[\![\_\,]\!] \quad \text{eqn}}{\underline{\forall} X.C[\![\_]\!] \quad \text{eqn}} \text{ foralls}$$

$$\frac{C[\![\_\,]\!] \quad \text{type}}{C[\![\_\,]\!] \preceq T \quad \text{eqn}} \text{ lsubsumption}$$

$$\frac{C[\![\_]\!] \quad \text{type}}{T \preceq C[\![\_]\!] \quad \text{eqn}} \text{ rsubsumption}$$

Figure 5.2.2. Context formation rules for equations

Example 5.2.3. Consider the following type:

$$C[\![(\forall B.\, B)]\!] = (\forall A.\, A) \to (\forall B.\, B) \preceq (\forall C.\, C) \to (\forall D.\, D)$$

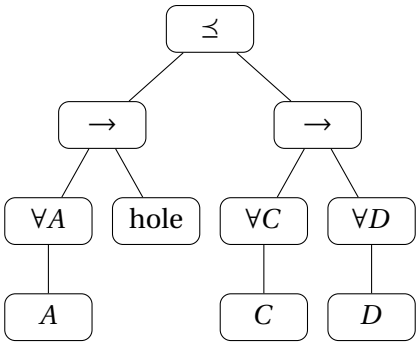
where $\forall B.\, B$ is in the hole of a type context. That is, we consider the type context:

$$(\forall A.\, A) \to [\![\ \_\ ]\!] \preceq (\forall C.\, C) \to (\forall D.\, D)$$

Then, applying the context formation rules yields the following derivation:

$$\dfrac{\dfrac{\dfrac{}{[\![\ \_\ ]\!]}\ \text{hole type}}{(\forall A.\, A) \to ([\![\ \_\ ]\!])}\ \text{type constructor}}{(\forall A.\, A) \to [\![\ \_\ ]\!] \preceq (\forall C.\, C) \to (\forall D.\, D)}\ \text{lsubsumption}$$

Following is the syntax tree presentation of $\forall A.\, A \to [\![\ \_\ ]\!] \preceq \forall C.\, C \to \forall D.\, D$:



The context formation rules can be used to track different aspects of type and equation structures. In the following sections, we use it to track visible variables (see Section 5.2.1) and varity (see Section 5.2.3). The following notation will be used to represent how visibility and varities are tracked within a type context:

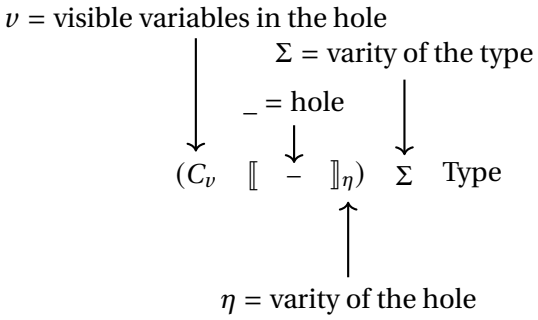


Figure 5.2.4. Notation for holes in types

The following notation shows how we track visibility and varity in equation contexts. Since the varity of equations is always +, we do not need to track it explicitly:

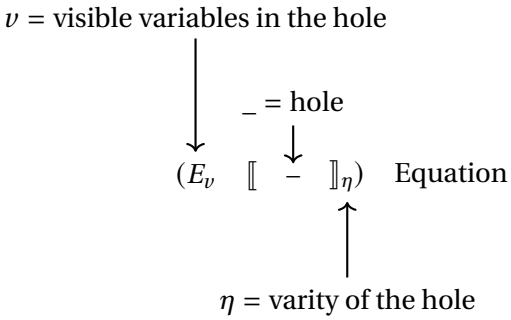


Figure 5.2.5. Notation for holes in equations

5.2.1 Visibility for holes

The type context may provide variables that are visible to the term in the hole, thus the term in the hole can use the variables that are visible to it. We indicate the visible variables by subscripting the context ($C_v[\![T]\!]_\eta$ Type) to indicate that the variables in the set v are visible in the hole, and the hole has varity η . The rules for constructing type contexts are presented in Figure 5.2.6.

Figure 5.2.6 shows the rules for constructing context types while tracking the visible variables to the hole. We define context types only for types whose varities are negative. The hole type$_{\text{vis}}$ rule handles the simplest case: a context with only a hole and no visible variables. The universal type$_{\text{vis}}$ rule extends the visible variable set by adding the bound variable X in $\forall X$. The type constructor$_{\text{vis}}$ rule shows how the context's visibility is updated to include universal variables from the arguments $T_1, \ldots, T_n$ of type constructor F. In particular, the univ$(F(T_1, \ldots T_n))$ is added to the set of visible variables in the context. Therefore, we require a set of rules that extract the universal quantifiers in types, which are defined in Section 5.2.2.

$$\frac{}{\emptyset[\![\_]\!]_- \quad \text{type}} \text{ hole type}_{\text{vis}}$$

$$\frac{C_v[\![\_]\!]_- \quad \text{type}}{\forall X.C_{v,X}[\![\_]\!]_- \quad \text{type}} \text{ universal type}_{\text{vis}}$$

$$\frac{C_v[\![\_]\!]_- \quad \text{type}}{F(T_1, \ldots, C_{v,\,\text{univ}(F(T_1,\ldots,T_n))}[\![\_]\!]_-, \ldots, T_n) \quad \text{type}} \text{ type constructor}_{\text{vis}}$$

Figure 5.2.6. Visibility based on type formation rules

Like type contexts, an equation context also provides visible variables to the equation filling the hole, which we indicate with the subscript notation ($C_v[\![\text{Equ}]\!]$ eqn). The rules for constructing equation contexts are presented in Figure 5.2.7. As before, these rules formalize how a context determines the visible variables to the hole. The equation$_{\text{vis}}$ rule handles the base case of an empty context. The conjunction$_{\text{vis}}$ rule shows how the context propagates through conjunctions, adding the universal quantified types in the conjuncted equation to the visible variable set. As with types, the rules for extracting variables in universal quantifications from an equation are defined in Section 5.2.2. The existential$_{\text{vis}}$ and forall$_{\text{vis}}$ rule extend the visible variable set by adding the newly bound variable. The

lsubsumption$_{\text{vis}}$ and rsubsumption$_{\text{vis}}$ rules define how a type context can form an equation context by appearing in a subsumption, adding variables that occur in universal quantifications on the left or right side, respectively.

$$\frac{}{\emptyset[\![\_]\!] \quad \text{eqn}}\ \text{equation}_{\text{vis}}$$

$$\frac{C_v[\![\_]\!] \quad \text{eqn}}{\text{Equ},\ C_{v,\text{univE(Equ)}}[\![\_]\!] \quad \text{eqn}}\ \text{conjunction}_{\text{vis}}$$

$$\frac{C_v[\![\_]\!] \quad \text{eqn}}{\underline{\exists} X.C_{v,X}[\![\_]\!] \quad \text{eqn}}\ \text{existentials}_{\text{vis}}$$

$$\frac{C_v[\![\_]\!] \quad \text{eqn}}{\underline{\forall} X.C_{v,X}[\![\_]\!] \quad \text{eqn}}\ \text{foralls}_{\text{vis}}$$

$$\frac{C_v[\![\_]\!]_- \quad \text{type}}{C_{v,\text{univ(T)}}[\![\_]\!] \preceq T \quad \text{eqn}}\ \text{lsubsumption}_{\text{vis}}$$

$$\frac{C_v[\![\_]\!]_- \quad \text{type}}{T \preceq C_{v,\text{univ(T)}}[\![\_]\!] \quad \text{eqn}}\ \text{rsubsumption}_{\text{vis}}$$

Figure 5.2.7. Visibility based on equation formation rules

Example 5.2.8. Consider the following type:

$$C[\![(\forall B.\, B)]\!] = (\forall A.\, A) \to (\forall B.\, B) \preceq (\forall C.\, C) \to (\forall D.\, D)$$

where$\forall B.\, B$ is in the hole of a type context. That is, we consider the type context:

$$(\forall A.\, A) \to [\![\ \_\ ]\!] \preceq (\forall C.\, C) \to (\forall D.\, D)$$

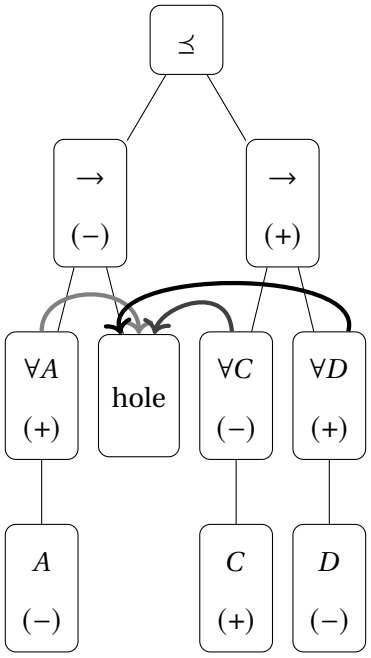
Applying the visibility rules then yields the following derivation:

$$\frac{\dfrac{\dfrac{}{\emptyset[\![\ \_\ ]\!]_-}\ \text{hole type}_{\text{vis}}}{(\forall A.\, A) \to (_A[\![\ \_\ ]\!]_-)}\ \text{type constructor}_{\text{vis}}}{(\forall A.\, A) \to {}_{A,C,D}[\![\ \_\ ]\!]_- \preceq (\forall C.\, C) \to (\forall D.\, D)}\ \text{lsubsumption}_{\text{vis}}$$

The corresponding syntax tree shown below illustrates which quantifiers are visible in the hole in the

equation:

$$(\forall A.\, A) \to [\![\ \_\ ]\!] \preceq (\forall C.\, C) \to (\forall D.\, D)$$



All universally quantified types are visible to a hole, which is acceptable because all universally quantified types can turn into underlined universally or existentially quantified types outside an equation. However, it is crucial that substitutions based on this visibility do not introduce cyclic dependencies.

5.2.2 Extract universal quantifications

The following figure defines how universal quantifications are extracted for types. Type variables do not contribute any quantifications and return the empty set when reached. For universal types, the universally quantified variable is added. For a type constructor, the universal quantifications are the union of the universal quantification of its arguments.

$$\frac{}{\mathrm{univ}(X) = \emptyset}\ \text{type variable}$$

$$\frac{\mathrm{univ}(T) = P}{\mathrm{univ}(\forall X.T) = \{X\} \cup P}\ \text{universal type}$$

$$\frac{\mathrm{univ}(T_1) = P_1 \quad \ldots \quad \mathrm{univ}(T_n) = P_n}{\mathrm{univ}(F(T_1, \ldots, T_n)) = P_1 \cup \cdots \cup P_n}\ \text{type constructor}$$

Figure 5.2.9. Universal quantifications in types

The following figure defines how universal quantifications are extracted from equations. Similar to types, we track where types are universally quantified. In a conjunction, the universal quantification are the union of those from both sides. For existential and universal quantifiers, we first add the quantified type to the variables, and then recursively inspect their bodies. For subsumptions, we extract the universal quantifiers from the types on both sides of the relation using univ, as defined in Figure 5.2.9.

$$\frac{\mathrm{univE}(\mathrm{Equ}) = P_1 \quad \mathrm{univE}(\mathrm{Equ}') = P_2}{\mathrm{univE}(\mathrm{Equ}, \mathrm{Equ}') = P_1 \cup P_2}\ \text{conjunction}$$

$$\frac{\mathrm{univE}(\mathrm{Equ}) = P}{\mathrm{univE}(\underline{\exists} X.\mathrm{Equ}) = \{X\} \cup P}\ \text{existential}$$

$$\frac{\mathrm{univE}(\mathrm{Equ}) = P}{\mathrm{univE}(\underline{\forall} X.\mathrm{Equ}) = \{X\} \cup P}\ \text{universal}$$

$$\frac{\mathrm{univ}(T) = P_1 \quad \mathrm{univ}(S) = P_2}{\mathrm{univE}(T \preceq S) = P_1 \cup P_2}\ \text{subsumption}$$

Figure 5.2.10. Universal quantifications in equations

5.2.3 Varity of context holes

The type context defined in Section 5.2 can be used to track how the type inserted into the hole gets a variance, written as $((C[\![\ \_\ ]\!]_v) \quad w$ type), which denotes a context into which a type with variance v can be inserted when the entire outer type has variance w. When a type T is inserted into the hole, we write $((C[\![\ T\ ]\!]_v) \quad w$ type) to denote the resulting type.

Figure 5.2.11 presents the rules for building type contexts that track variance. These rules are based on the type context formation rules defined in Figure 5.2.1, and the variance rules defined in Figure 5.1.2. The hole type$_{\text{varity}}$ rule is the base case, where the context is just a hole annotated with variance v. The universal type$_{\text{varity}}$ rule allows the context to appear under a universal quantifier $\forall X.\ C[\![\ \_\ ]\!]$, which does not affect the variance, so the hole retains variance v. The type constructor$_{\text{varity}}$ rule applies when the context appears in the ith argument of a type constructor $F(t_1, \ldots, t_n)$. If that position has declared arity v_i, i.e., arity$(F) = [v_1, \ldots, v_n]$, then the overall variance at the hole becomes $v_i \cdot v$, where $\cdot$ is defined in Figure 5.1.1.

$$\frac{}{[\![\ \_\ ]\!]_v \quad v \text{ type}} \text{ hole type}_{\text{varity}}$$

$$\frac{C[\![\ \_\ ]\!]_v \quad w \text{ type}}{\forall X.\ C[\![\ \_\ ]\!]_v \quad w \text{ type}} \text{ universal type}_{\text{varity}}$$

$$\frac{(C[\![\ \_\ ]\!]_v \quad w \text{ type}) \quad \text{arity}(F) = [v_1, \ldots, v_n]}{(F(T_1, \ldots, C[\![\ \_\ ]\!]_{(v_i \cdot v)}, \ldots, T_n) \quad w \text{ type})} \text{ type constructor}_{\text{varity}}$$

Figure 5.2.11. Varity propagation rules (type formations)

Example 5.2.12. Consider the following type:

$$C[\![\ \forall A.\, A \to A\ ]\!] = (\forall A.\, A \to A) \to (\forall B.\, B \to B)$$

where $(\forall A.\, A \to A)$ is in the hole of a type context. That is, we consider the type context:

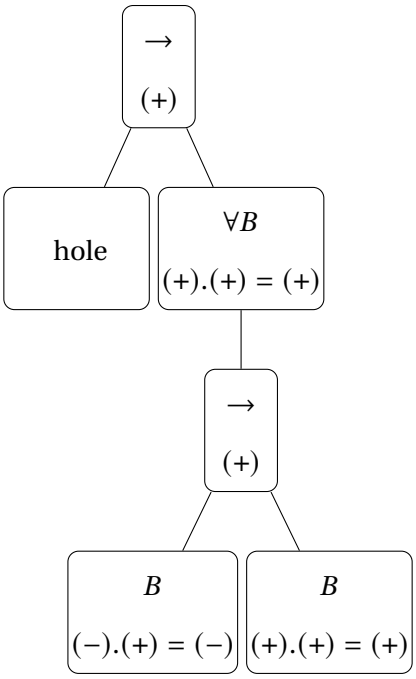
$$[\![\ \_\ ]\!] \to (\forall B.\, B \to B)$$

By applying the varity propagation rules, we can determine the variance at the hole as follows:

$$\frac{([\![\ \_\ ]\!]_+ \quad + \text{ type}) \quad \text{arity}(\to) = [-, +]}{([\![\ \_\ ]\!]_- \to (\forall B.\, B \to B) \quad + \text{ type})} \text{ type constructor}_{\text{varity}}$$

Here, the variance at the hole is initially +, and since it appears in the left (contravariant) position of the function type constructor $\to$, the variance at the hole becomes $-$.

The following is the syntax tree presentation of the type above:



The varity propagation rules for equation formation follow a structure similar to those for type contexts. We write $(E[\![\_]\!]_v$ Equation) to denote an equation context with a hole that has varity v. Note that, the outer varity of all equations is always +, therefore there is no need to track the outer varity of an equation. Figure 5.2.13 presents the rules for constructing equation contexts that track varity. The $\text{equation}_{\text{varity}}$ rule is the base case where the context is just a hole. The $\text{conjunction}_{\text{varity}}$ rule adds another equation using a comma. The $\text{existentials}_{\text{varity}}$ and $\text{foralls}_{\text{varity}}$ rules wrap the context with $\underline{\exists}$ or $\underline{\forall}$, respectively. The $\text{rsubsumption}_{\text{varity}}$ rule builds a subsumption relation with the hole in a contravariant position. The $\text{lsubsumption}_{\text{varity}}$ rule builds a subsumption relation with the hole in a covariant position.

$$\frac{}{(E[\![\ \_\ ]\!]_\nu \quad \text{eqn})}\ \text{equation}_{\text{varity}}$$

$$\frac{(E[\![\ \_\ ]\!]_\nu \quad \text{eqn})}{(\text{Equ},\ E[\![\ \_\ ]\!]_\nu \quad \text{eqn})}\ \text{conjunction}$$

$$\frac{(E[\![\ \_\ ]\!]_\nu \quad \text{eqn})}{(\underline{\exists} X.\ E[\![\ \_\ ]\!]_\nu \quad \text{eqn})}\ \text{existentials}_{\text{varity}}$$

$$\frac{(E[\![\ \_\ ]\!]_\nu \quad \text{eqn})}{(\underline{\forall} X.\ E[\![\ \_\ ]\!]_\nu \quad \text{eqn})}\ \text{foralls}_{\text{varity}}$$

$$\frac{(S[\![\ \_\ ]\!]_\nu \quad +\ \text{type}) \quad T\ \text{type}}{(S[\![\ \_\ ]\!]_{(-.\nu)} \preceq T \quad \text{eqn})}\ \text{rsubsumption}_{\text{varity}}$$

$$\frac{(S[\![\ \_\ ]\!]_\nu \quad +\ \text{type}) \quad T\ \text{type}}{(T \preceq S[\![\ \_\ ]\!]_\nu \quad \text{eqn})}\ \text{lsubsumption}_{\text{varity}}$$

Figure 5.2.13. Varity propagation rules (equation formations)

Example 5.2.14. Consider the equation:

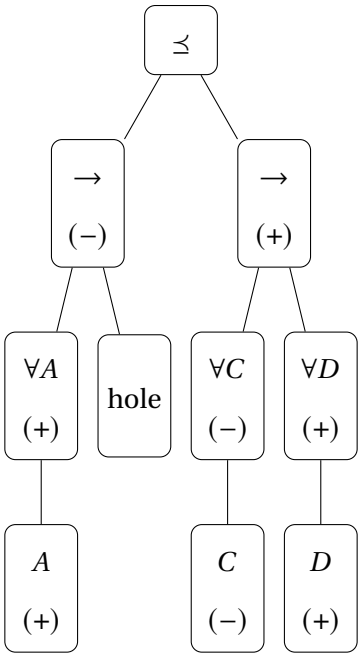
$$C[\![\forall B.\ B]\!] = (\forall A.\ A) \to (\forall B.\ B) \preceq (\forall C.\ C) \to (\forall D.\ D)$$

where $\forall B.\ B$ is in a hole, then applying the varity rules gives the following derivation:

$$\frac{\dfrac{\dfrac{}{([\![\ \_\ ]\!]_+ \quad +\ \text{type})}\ \text{hole type}_{\text{varity}}}{(\forall A.\ A \to [\![\ \_\ ]\!]_+ \quad +\ \text{type}) \quad \text{arity}(\to) = [-,+]}\ \text{type constructor}_{\text{varity}}}{((\forall A.\ A) \to [\![\ \_\ ]\!]_- \preceq (\forall C.\ C) \to (\forall D.\ D) \quad \text{eqn})}\ \text{lsubsumption}_{\text{varity}}$$

The corresponding syntax tree shown below illustrates the varity of the hole:

$$(\forall A.\ A) \to [\![\ \_\ ]\!]_- \preceq (\forall C.\ C) \to (\forall D.\ D)$$



The varity of the hole can determine how to pull out a universal quantification inside types, as discussed in Proposition 5.2.15. The following two-way rule is used in its proof.

$$\frac{W \preceq F(S_1, \ldots, S_n)}{\underline{\exists} T_1, \ldots, T_n. W = F(T_1, \ldots, T_n), W \preceq F(S_1, \ldots, S_n)} \text{con}_{\preceq}$$

which means a type constructor can only match another type with the same constructor.

Proposition 5.2.15. Universal quantifications nested in the subsumption relation can be pulled out, as follows:

(i) $$\frac{(W \preceq W'[\![\forall A.S]\!]_+) \quad \text{eqn}}{\underline{\forall} A.\, (W \preceq W'[\![S]\!]_+) \quad \text{eqn}}$$

(ii) $$\frac{(W[\![\forall A.S]\!]_- \preceq W') \quad \text{eqn}}{\underline{\forall} A.\, (W[\![S]\!]_- \preceq W') \quad \text{eqn}}$$

(iii) $$\frac{(W[\![\forall A.S]\!]_+ \preceq W') \quad \text{eqn}}{\underline{\exists} A.\, (W[\![S]\!]_+ \preceq W') \quad \text{eqn}}$$

(iv) $$\frac{(W \preceq W'[\![\forall A.S]\!]_-) \quad \text{eqn}}{\underline{\exists} A.\, (W \preceq W'[\![S]\!]_-) \quad \text{eqn}}$$

Proof. The proof is by a mutual structural induction:

(i) We proceed by structural induction on the type context W', which is formed using the rules in Figure 5.2.11.

hole type: In this case, the context is simply a hole:

$$(W'[\![\forall A.S]\!]_+ \quad \text{type}) = \forall A.S$$

Therefore, we need to prove:

$$\frac{((W \preceq \forall A.S) \quad \text{eqn})}{(\underline{\forall} A.(W \preceq S) \quad \text{eqn})}$$

which holds based on the gen rule defined in Definition 4.1.5 and serves as the base case of the structural induction.

universal type: In this case, the context consists of a universal type:

$$(W'[\![\forall A.S]\!]_+ \quad \text{type}) = (\forall Y.C[\![\forall A.S]\!]_+ \quad \text{type})$$

Therefore, we need to prove:

$$\frac{W \preceq \forall Y.C[\![\forall A.S]\!]_+}{\underline{\forall} A(W \preceq \forall Y.C[\![S]\!]_+)}$$

which is proved by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(W \preceq \forall Y.C[\![\forall A.S]\!]_+ \quad \text{eqn})}{(\underline{\forall} Y(W \preceq C[\![\forall A.S]\!]_+) \quad \text{eqn})}\,\text{gen}}{(\underline{\forall} Y(\underline{\forall} A(W \preceq C[\![S]\!]_+)) \quad \text{eqn})}\,\text{induction hypothesis (i)}}{(\underline{\forall} A(\underline{\forall} Y(W \preceq C[\![S]\!]_+)) \quad \text{eqn})}\,\underline{\forall}\text{-commute}}{(\underline{\forall} A(W \preceq \forall Y.C[\![S]\!]_+) \quad \text{eqn})}\,\text{gen}$$

type constructor: In this case, the context consists of a type constructor:

$$(W'[\![\forall A.S]\!]_+ \quad \text{type}) = (F(S_1, \ldots, C[\![\forall A.S]\!]_+, \ldots, S_n)\text{type})$$

Therefore, we need to prove:

$$\frac{(W \preceq F(S_1, \ldots, C[\![\forall A.S]\!]_+, \ldots, S_n) \quad \text{eqn})}{(\underline{\forall} A(W \preceq F(S_1, \ldots, C[\![S]\!]_+, \ldots, S_n)) \quad \text{eqn})}$$

which is proved by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(W \preceq F(S_1, \ldots, C[\![\forall A.S]\!]_+, \ldots, S_n) \quad \text{eqn})}{(\underline{\exists} T_1, \ldots T_n.W = F(T_1, \ldots, T_n)), (W \preceq F(S_1, \ldots, C[\![\forall A.S]\!]_+, \ldots, S_n) \quad \text{eqn})}\,\text{cons}_{\preceq}}{(\underline{\exists} T_1, \ldots T_n.W = F(T_1, \ldots, T_n)), (F(T_1, \ldots, T_i, \ldots, T_n) \preceq F(S_1, \ldots, C[\![\forall A.S]\!]_+, \ldots, S_n) \quad \text{eqn}) \quad \text{arity}(F) = [\nu_1, \ldots, +, \ldots, \nu_n]}\,\text{substitution}}{(\underline{\exists} T_1, \ldots T_n.W = F(T_1, \ldots, T_n)), (T_1 \preceq_{\nu_1} S_1, \ldots, T_i \preceq_{\nu_i} C[\![\forall A.S]\!]_+, \ldots, T_n \preceq_{\nu_n} S_n \quad \text{eqn})}\,\text{cons}}{(\underline{\exists} T_1, \ldots T_n.W = F(T_1, \ldots, T_n)), (T_1 \preceq_{\nu_1} S_1, \ldots, \underline{\forall} A(T_i \preceq_{\nu_i} C[\![S]\!]_+), \ldots, T_n \preceq_{\nu_n} S_n \quad \text{eqn})}\,\text{induction hypothesis (i)}}{(\underline{\exists} T_1, \ldots T_n.W = F(T_1, \ldots, T_n)), (\underline{\forall} A(T_1 \preceq_{\nu_n} S_1, \ldots, T_i \preceq_{\nu_i} C[\![S]\!]_+, \ldots, T_n \preceq_{\nu_n} S_n) \quad \text{eqn}) \quad \text{arity}(F) = [\nu_1, \ldots, +, \ldots, \nu_n]}\,\underline{\forall}\text{-rescope}}{(\underline{\exists} T_1, \ldots T_n.W = F(T_1, \ldots, T_n)), (\underline{\forall} A(F(T_1, \ldots, T_i, \ldots, T_n) \preceq F(S_1, \ldots, C[\![S]\!]_+, \ldots, S_n)) \quad \text{eqn})}\,\text{cons}}{\underline{\forall} A((\underline{\exists} T_1, \ldots T_n.W = F(T_1, \ldots, T_n)), (F(T_1, \ldots, T_i, \ldots, T_n) \preceq F(S_1, \ldots, C[\![S]\!]_+, \ldots, S_n)) \quad \text{eqn}))}\,\forall\text{-distr}}{\underline{\forall} A((\underline{\exists} T_1, \ldots T_n.W = F(T_1, \ldots, T_n)), (W \preceq F(S_1, \ldots, C[\![S]\!]_+, \ldots, S_n)) \quad \text{eqn}))}\,\text{resubstitute}}{(\underline{\forall} A(W \preceq F(S_1, \ldots, C[\![S]\!]_+, \ldots, S_n)) \quad \text{eqn})}\,\text{cons}_{\preceq}$$

(ii) We proceed by structural induction on the type context W, which is formed using the rules in Figure 5.2.11. This is a mutual case with (i), and the base case comes from (i).

hole type: This case does not arise because the type $\forall A.\, S$, appearing on the left of a subsumption, cannot have negative variance.

universal type: In this case, the context consists of a universal type:

$$(W[\![\forall A.S]\!]_- \quad \text{type}) = (\forall Y.C[\![\forall A.S]\!]_- \quad \text{type})$$

Therefore, we need to prove:

$$\frac{(\forall Y.C[\![\forall A.S]\!]_- \preceq W' \quad \text{eqn})}{(\underline{\forall} A(\forall Y.C[\![S]\!]_- \preceq W') \quad \text{eqn})}$$

which is proved by the following derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{((\forall Y.C[\![\forall A.S]\!])_- \preceq W' \quad \text{eqn})}{(\underline{\forall} Y(C[\![\forall A.S]\!]_-) \preceq W' \quad \text{eqn})}\,\text{induction hypothesis (ii)}}{(\underline{\forall} Y(\underline{\forall} A(C[\![S]\!]_-) \preceq W') \quad \text{eqn})}\,\text{induction hypothesis (ii)}}{(\underline{\forall} A(\underline{\forall} Y(C[\![S]\!]_- \preceq W')) \quad \text{eqn})}\,\underline{\forall}\text{-commute}}{(\underline{\forall} A(\forall Y.C[\![S]\!]_- \preceq W') \quad \text{eqn})}\,\text{induction hypothesis (ii)}$$

type constructor: In this case, the context consists of a type constructor:

$$(W[\![\forall A.S]\!]_- \quad \text{type}) = (F(S_1, \ldots, C[\![\forall A.S]\!]_-, \ldots, S_n) \quad \text{type})$$

Therefore, we need to prove:

$$\frac{F(S_1, \ldots, C[\![\forall A.S]\!]_-, \ldots, S_n) \preceq W'}{\underline{\forall} A(F(S_1, \ldots, C[\![S]\!]_-, \ldots, S_n) \preceq W')}$$

which is proved by the following derivation (assuming $W' = F(T_1, \ldots, T_n)$, otherwise the entire subsumption is invalid. This corresponds to using the $\text{cons}_{\preceq}$.):

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{W' = F(T_1, \ldots, T_n), (F(S_1, \ldots, C[\![\forall A.S]\!]_-, \ldots, S_n) \preceq W' \quad \text{eqn})}{W' = F(T_1, \ldots, T_n), (F(S_1, \ldots, C[\![\forall A.S]\!]_-, \ldots, S_n) \preceq F(T_1, \ldots, T_i, \ldots, T_n) \quad \text{eqn}) \quad \text{arity}(F) = [\nu_1, \ldots, -, \ldots, \nu_n]}\,\text{substitution}}{W' = F(T_1, \ldots, T_n), (S_1 \preceq_{\nu_1} T_1, \ldots, T_i \preceq_{\nu_i} C[\![\forall A.S]\!]_+, \ldots, S_n \preceq_{\nu_n} T_n \quad \text{eqn})}\,\text{cons}}{W' = F(T_1, \ldots, T_n), (S_1 \preceq_{\nu_1} T_1, \ldots, \underline{\forall} A(T_i \preceq_{\nu_i} C[\![S]\!]_+), \ldots, S_n \preceq_{\nu_n} T_n \quad \text{eqn})}\,\text{induction hypothesis (i)}}{W' = F(T_1, \ldots, T_n), (\underline{\forall} A(S_1 \preceq_{\nu_1} T_1, \ldots, T_i \preceq_{\nu_i} C[\![S]\!]_+, \ldots, S_n \preceq_{\nu_n} T_n) \quad \text{eqn}) \quad \text{arity}(F) = [\nu_1, \ldots, -, \ldots, \nu_n]}\,\underline{\forall}\text{-rescope}}{W' = F(T_1, \ldots, T_n), (\underline{\forall} A(F(S_1, \ldots, C[\![S]\!]_-, \ldots, S_n) \preceq F(T_1, \ldots, T_i, \ldots, T_n)) \quad \text{eqn})}\,\text{cons}}{W' = F(T_1, \ldots, T_n), (\underline{\forall} A(F(S_1, \ldots, C[\![S]\!]_-, \ldots, S_n) \preceq W) \quad \text{eqn})}\,\text{resubstitute}$$

(iii) We proceed by structural induction on the type context W, which is formed using the rules in Figure 5.2.11.

hole type: In this case, the context is simply a hole:

$$(W[\![\forall A.S]\!]_+ \quad \text{type}) = \forall A.S$$

Therefore, we need to prove:

$$\frac{((\forall A.S \preceq W') \quad \text{eqn})}{(\underline{\exists} A.(S \preceq W) \quad \text{eqn})}$$

which holds based on the spec rule defined in Definition 4.1.5 and serves as the base case of the structural induction.

universal type: In this case, the context consists of a universal type.

$$(W[\![\forall A.S]\!]_+ \quad \text{type}) = (\forall Y.C[\![\forall A.S]\!]_+ \quad \text{type})$$

Therefore, we need to prove:

$$\frac{(\forall Y.C[\![\forall A.S]\!]_+ \preceq W' \quad \text{eqn})}{(\underline{\exists} A(\forall Y.C[\![S]\!]_+ \preceq W') \quad \text{eqn})}$$

which is proved by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{((\forall Y.C[\![\forall A.S]\!])_+ \preceq W' \quad \text{eqn})}{(\underline{\exists} Y(C[\![\forall A.S]\!]_+) \preceq W' \quad \text{eqn})}\,\text{spec}}{(\underline{\exists} Y(\underline{\exists} A(C[\![S]\!]_+) \preceq W') \quad \text{eqn})}\,\text{induction hypothesis (iii)}}{(\underline{\exists} A(\underline{\exists} Y(C[\![S]\!]_+ \preceq W')) \quad \text{eqn})}\,\underline{\exists}\text{-commute}}{(\underline{\exists} A(\forall Y.C[\![S]\!]_+ \preceq W') \quad \text{eqn})}\,\text{spec}$$

type constructor: In this case, the context consists of a type constructor:

$$(W[\![\forall A.S]\!]_+ \quad \text{type}) = (F(S_1, \ldots, C[\![\forall A.S]\!]_+, \ldots, S_n) \quad \text{type})$$

Therefore, we need to prove:

$$\frac{F(S_1, \ldots, C[\![\forall A.S]\!]_+, \ldots, S_n) \preceq W'}{\underline{\exists} A(F(S_1, \ldots, C[\![S]\!]_+, \ldots, S_n) \preceq W')}$$

which is proved by the following derivation (assuming $W' = F(T_1, \ldots, T_n)$, otherwise the entire subsumption is invalid. This corresponds to using the $\text{cons}_{\preceq}$.):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{W' = F(T_1,\ldots,T_n), (F(S_1,\ldots,C[\![\forall A.S]\!]_+,\ldots,S_n) \preceq W' \quad \text{eqn})}{W' = F(T_1,\ldots,T_n), (F(S_1,\ldots,C[\![\forall A.S]\!]_+,\ldots,S_n) \preceq F(T_1,\ldots,T_i,\ldots,T_n) \quad \text{eqn}) \quad \text{arity}(F) = [\nu_1,\ldots,+,\ldots,\nu_n]}\ \text{substitution}}{W' = F(T_1,\ldots,T_n), (S_1 \preceq_{\nu_1} T_1,\ldots,C[\![\forall A.S]\!]_+ \preceq_{\nu_i} T_i,\ldots,S_n \preceq_{\nu_n} T_n \quad \text{eqn})}\ \text{cons}}{W' = F(T_1,\ldots,T_n), (T_1 \preceq_{\nu_1} S_1,\ldots,\underline{\exists} A(C[\![S]\!]_+ \preceq_{\nu_i} T_i),\ldots,T_n \preceq_{\nu_n} S_n \quad \text{eqn})}\ \text{induction hypothesis (iii)}}{W' = F(T_1,\ldots,T_n), (\underline{\exists} A(S_1 \preceq_{\nu_1} T_1,\ldots,C[\![S]\!]_+ \preceq_{\nu_i} T_i,\ldots,S_n \preceq_{\nu_n} T_n) \quad \text{eqn}) \quad \text{arity}(F) = [\nu_1,\ldots,+,\ldots,\nu_n]}\ \underline{\forall}\text{-rescope}}{W' = F(T_1,\ldots,T_n), (\underline{\exists} A(F(S_1,\ldots,C[\![S]\!]_+,\ldots,S_n) \preceq F(T_1,\ldots,T_i,\ldots,T_n)) \quad \text{eqn})}\ \text{cons}}{W' = F(T_1,\ldots,T_n), (\underline{\exists} A(F(S_1,\ldots,C[\![S]\!]_+,\ldots,S_n) \preceq W') \quad \text{eqn})}\ \text{resubstitute}$$

(iv) We proceed by structural induction on the type context W', which is formed using the rules in Figure 5.2.11. This is a mutual case with (iii), and the base case comes from (iii).

hole type: This case does not arise because the type $\forall A.\, S$, appearing on the right of a subsumption, cannot have negative variance.

universal type: In this case, the context consists of a universal type:

$$(W'[\![\forall A.S]\!]_- \quad \text{type}) = (\forall Y.C[\![\forall A.S]\!]_- \quad \text{type})$$

Therefore, we need to prove:

$$\dfrac{W \preceq \forall Y.C[\![\forall A.S]\!]_-}{\underline{\exists} A(W \preceq \forall Y.C[\![S]\!]_-)}$$

which is proved by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{(W \preceq \forall Y.C[\![\forall A.S]\!]_- \quad \text{eqn})}{(\underline{\exists} Y(W \preceq C[\![\forall A.S]\!]_- \quad \text{eqn})}\ \text{induction hypothesis (iv)}}{(\underline{\exists} Y(\underline{\exists} A(W \preceq C[\![S]\!]_-)) \quad \text{eqn})}\ \text{induction hypothesis (iv)}}{(\underline{\exists} A(\underline{\exists} Y(W \preceq C[\![S]\!]_-)) \quad \text{eqn})}\ \underline{\exists}\text{-commute}}{(\underline{\exists} A(W \preceq \forall Y.C[\![S]\!]_-) \quad \text{eqn})}\ \text{induction hypothesis (iv)}$$

type constructor: In this case, the context consists of a type constructor:

$$(W'[\![\forall A.S]\!]_- \quad \text{type}) = (F(S_1,\ldots,C[\![\forall A.S]\!]_-,\ldots,S_n) \quad \text{type})$$

Therefore, we need to prove:

$$\dfrac{(W \preceq F(S_1,\ldots,C[\![\forall A.S]\!]_-,\ldots,S_n) \quad \text{eqn})}{(\underline{\exists} A(W \preceq F(S_1,\ldots,C[\![S]\!]_-,\ldots,S_n)) \quad \text{eqn})}$$

which is proved by the following derivation (assuming $W = F(T_1,\ldots,T_n)$, otherwise the entire subsumption is invalid. This corresponds to using the $\text{cons}_{\preceq}$.):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{W = F(T_1,\ldots,T_n),\,(W \preceq F(S_1,\ldots,C[\![\forall A.S]\!]_-,\ldots,S_n)\quad \text{eqn})}{W = F(T_1,\ldots,T_n),\,(F(T_1,\ldots,T_i,\ldots,T_n) \preceq F(S_1,\ldots,C[\![\forall A.S]\!]_-,\ldots,S_n)\quad \text{eqn})\quad \text{arity}(F) = [\nu_1,\ldots,+,\ldots,\nu_n]}\ \text{substitution}}{W = F(T_1,\ldots,T_n),\,(T_1 \preceq_{\nu_1} S_1,\ldots,C[\![\forall A.S]\!]_+ \preceq_{\nu_i} T_i,\ldots,T_n \preceq_{\nu_n} S_n\quad \text{eqn})}\ \text{cons}}{W = F(T_1,\ldots,T_n),\,(T_1 \preceq_{\nu_1} S_1,\ldots,\underline{\exists}A(C[\![S]\!]_+ \preceq_{\nu_i} T_i),\ldots,T_n \preceq_{\nu_n} S_n\quad \text{eqn})}\ \text{induction hypothesis (iii)}}{W = F(T_1,\ldots,T_n),\,(\underline{\exists}A(T_1 \preceq_{\nu_n} S_1,\ldots,C[\![S]\!]_+ \preceq_{\nu_i} T_i,\ldots,T_n \preceq_{\nu_n} S_n)\quad \text{eqn})\quad \text{arity}(F) = [\nu_1,\ldots,-,\ldots,\nu_n]}\ \underline{\exists}\text{-rescope}}{W = F(T_1,\ldots,T_n),\,(\underline{\exists}A(F(T_1,\ldots,T_i,\ldots,T_n) \preceq F(S_1,\ldots,C[\![S]\!]_-,\ldots,S_n))\quad \text{eqn})}\ \text{cons}}{W = F(T_1,\ldots,T_n),\,(\underline{\exists}A(W \preceq F(S_1,\ldots,C[\![S]\!]_-,\ldots,S_n))\quad \text{eqn})}\ \text{resubstitute}$$

□

5.3 Quantified Substitution

When substituting types, non-quantified substitutions, replaces the variable directly:

$$X \to X \quad [D/X] \quad = \quad D \to D$$

However, when substituting with a universally quantified type, bound variable names can be α-renamed, and it is convenient to arrange that they remain distinct in each position where they appear. For example, consider:

$$X \to X \quad [\forall A.\, A \to A/X] \quad = \quad (\forall A_0.\, A_0 \to A_0) \to (\forall A_1.\, A_1 \to A_1)$$

In the above, the bound variable A in each $\forall$-quantified type is renamed (to A_0 and A_1) to ensure correct bindings and to avoid name clashes. One systematic way to achieve such α-renaming, in quantified substitution, is to assign indices from left to right to the occurrences of the substituted variable, and to reindex the bound variables by this index.

First, we define how to rename quantified variables in a type using an index:

Definition 5.3.1. $(T')_i$**:** For a type term T', the indexed version $(T')_i$ is defined inductively as follows:

1. If T' is a type variable X, then
$$(X)_i = X.$$
2. If T' is of the form $F(T_1,\ldots,T_n)$, then
$$\big(F(T_1,\ldots,T_n)\big)_i = F\big((T_1)_i,\ldots,(T_n)_i\big).$$
3. If T' is of the form $\forall X.S$, then
$$\big(\forall X.S\big)_i = \forall X_i.\big(S[X_i/X]\big)_i.$$

Next, we define how to traverse a type from left to right, applying indices systematically to all quantified variables using definition 5.3.1.

Definition 5.3.2. Quantified substitution for types: Let T and T' be type terms, and let A be a type variable. The quantified substitution $T[_iT'/A]_{i+\#_A(T)}$ replaces each occurrence of the variable A in T with the type term T'. To track the position of substitution in the term, we subscript the left bracket with i and the right bracket with $i + \#_A(T)$, where $\#_A(T)$ denotes the number of occurrences of A in T.

1. If T is a variable:

   a) If $X \neq A$, then the substitution does not affect X:

   $$X[_iT'/A]_i = X.$$

   b) If $X = A$, then the substitution replaces A with T' and shifts the index of the right bracket by one:

   $$A[_iT'/A]_{i+1} = (T')_i.$$

2. If T is of the form $F(T_1, \ldots, T_n)$, then the substitution proceeds component-wise. Let

   $$i_p = i + \sum_{q=1}^{p-1} \#_A(T_q),$$

   for $1 \leq p \leq n$. Let $j = i + \#_A(F(T_1, \ldots, T_n))$. Then we have:

   $$F(T_1, \ldots, T_n)[_iT'/A]_j = F\big(T_1[_{i_1}T'/A]_{i_1+\#_A(T_1)}, \ldots, T_n[_{i_n}T'/A]_{i_n+\#_A(T_n)}\big).$$

3. If T is of the form $\forall X.S$:

   a) If $X \neq A$, then the substitution does not affect the quantifier variable:

   $$(\forall X.S)[_iT'/A]_j = \forall X.\big(S[_iT'/A]_j\big).$$

   b) If $X = A$, then substitution does not affect the term because A is bound:

   $$(\forall A.S)[_iT'/A]_i = \forall A.S.$$

Next, we define how to traverse an equatin from left to right, applying indices systematically to all quantified variables using definition 5.3.1.

Definition 5.3.3. Quantified substitution for equations: Let Equ be equation and T' be a type, respectively, and let A be a type variable. The higher-order substitution $\mathrm{Equ}[{}_iT'/A]_{i+\#_A(\mathrm{Equ})}$ replaces each occurrence of the variable A in Equ with the type term T'. As with types, we track the substitution's position using the subscripts i and $i+\#_A(\mathrm{Equ})$.

1. If Equ is a conjunction $\mathrm{Equ}_1, \mathrm{Equ}_2$:

$$(\mathrm{Equ}_1, \mathrm{Equ}_2)[{}_iT'/A]_j = \big(\mathrm{Equ}_1[{}_iT'/A]_k,\ \mathrm{Equ}_2[{}_kT'/A]_j\big),$$

   where $k = i + \#_A(\mathrm{Equ}_1), \quad j = i + \#_A(\mathrm{Equ}_1) + \#_A(\mathrm{Equ}_2)$.

2. If Equ is of the form $\underline{\forall} X.\mathrm{Equ}'$:

   a) If $X \neq A$, then substitution proceeds under the quantifier:

$$(\underline{\forall} X.\mathrm{Equ}')[{}_iT'/A]_j = \underline{\forall} X.\big(\mathrm{Equ}'[{}_iT'/A]_j\big),$$

      where $j = i + \#_A(\mathrm{Equ}')$.

   b) If $X = A$, then substitution stops because A is bound:

$$(\underline{\forall} A.\mathrm{Equ}')[{}_iT'/A]_i = \underline{\forall} A.\mathrm{Equ}'.$$

3. If Equ is of the form $\underline{\exists} X.\mathrm{Equ}'$, the rules are the same as for $\underline{\forall} X.\mathrm{Equ}'$:

   a) If $X \neq A$:

$$(\underline{\exists} X.\mathrm{Equ}')[{}_iT'/A]_j = \underline{\exists} X.\big(\mathrm{Equ}'[{}_iT'/A]_j\big), \quad j = i + \#_A(\mathrm{Equ}').$$

   b) If $X = A$:

$$(\underline{\exists} A.\mathrm{Equ}')[{}_iT'/A]_i = \underline{\exists} A.\mathrm{Equ}'.$$

4. If Equ is of the form $T = S$ (equality):

$$(T = S)[{}_iT'/A]_j = \big(T[{}_iT'/A]_k = S[{}_kT'/A]_j\big),$$

   where $k = i + \#_A(T), \quad j = i + \#_A(T) + \#_A(S)$.

5. If Equ is of the form $T \preceq S$ (subsumption):

$$(T \preceq S)[{}_iT'/A]_j = \big(T[{}_iT'/A]_k \preceq S[{}_kT'/A]_j\big),$$

   where $k = i + \#_A(T), \quad j = i + \#_A(T) + \#_A(S)$.

Example 5.3.4. Quantified substitution examples

(1) Consider the type, $T = \forall A.(A \to B)$, and the substitution, $T[{}_0C/A]_{0+\#_A(T)}$, where C is fresh type variable. We compute, $\#_A(\forall A.(A \to B)) = \#_A(A \to B) = \#_A(A) + \#_A(B) = 1 + 0 = 1$, therefore, $\#_A(T) = 1$ and the substitution is $T[{}_0C/A]_1$. Since T is of the form $\forall A.S$ and $X = A$, the substitution stops because A is bound:

$$(\forall A.(A \to B))[{}_0C/A]_0 = \forall A.(A \to B).$$

(2) Next, consider the type, $T = \forall X.(A \to X)$, and the substitution, $T[{}_0C/A]_{0+\#_A(T)}$. In this case, $\#_A(\forall X.(A \to X)) = \#_A(A \to X) = 1 + 0 = 1$, therefore the quantified substitution is $T[{}_0C/A]_1$. Since $X \neq A$, substitution proceeds under the quantifier: $(\forall X.(A \to X))[{}_0C/A]_1 = \forall X.\big((A \to X)[{}_0C/A]_1\big)$, and for the inner type $A \to X[{}_0C/A]_1$ as it is a type constructor, on domain: $A[{}_0C/A]_1 = (C)_0 = C$ and on the codomain $X[{}_1C/A]_1 = X$ .Thus,

$$(\forall X.(A \to X))[{}_0C/A]_1 = \forall X.(C \to X).$$

5.3.1 Candidate solution

To give a solution for a set of equations, one must provide a list of non-circular quantified substitutions, which will then be verified for correctness. Consider the equation:

$$(\forall A.A \to A) \preceq (\forall B.B \to B) \to (\forall C.C \to C).$$

A candidate solution could be:

$$[\forall D.D \to D/A, \quad D_0/B, \quad C/D_1],$$

We apply the substitution list one by one using the following two inference rules:

$$\frac{C_v[\![\underline{\exists} X.E]\!] \quad \text{eqn} \quad v \vdash_{\text{type}} T'}{C_v[\![E[{}_iT'/X]_j]\!] \quad \text{eqn}} \ \exists\text{-Elim}$$

$$\frac{C_v[\![\forall A.T]\!] \quad \text{eqn} \quad v \vdash_{\text{type}} T'}{C_v[\![T[{}_iT'/A]_j]\!] \quad \text{eqn}} \ \forall\text{-Elim}$$

Figure 5.3.5. Elimination rules by substitution into quantified equations - proof search direction

Each substitution in the list is applied using one of these rules, depending on whether we are eliminating an existential variable or instantiating a universal quantifier. These rules ensure that the type being substituted is visible at the hole where the substitution occurs. Once all substitutions have been applied, the resulting equation can be checked for correctness.

Example 5.3.6. Consider the equation:

$$(\forall A.\, A \to A) \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C)$$

and the substitution list:

$$[\forall D.\, D \to D/A, \quad D_0/B, \quad C/D_1].$$

1. First substitution: $\forall D.\, D \to D/A$

   - We apply the $\forall$-Elim rule:

$$\frac{C_v[\![\forall A.\, (A \to A)]\!] \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C) \quad v \vdash_{\text{type}} \forall D.\, D \to D}{C_v[\![(A \to A)[_0(\forall D.\, D \to D)/A]_2]\!] \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C)}\ \forall\text{-Elim}$$

   - Compute the substitution:

$$A \to A[_0(\forall D.\, D \to D)/A]_2 = (\forall D.\, D \to D)_0 \to (\forall D.\, D \to D)_1$$

   - using the definition for $(T)_i$ as defined in Definition 5.3.1:

$$= (\forall D_0.\, D_0 \to D_0) \to (\forall D_1.\, D_1 \to D_1)$$

   - Resulting equation:

$$(\forall D_0.\, D_0 \to D_0) \to (\forall D_1.\, D_1 \to D_1) \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C)$$

2. Second substitution: D_0/B

   - Apply the $\forall$-Elim rule:

$$\frac{(\forall D_0.\, D_0 \to D_0) \to (\forall D_1.\, D_1 \to D_1) \preceq C_v[\![\forall B.\, (B \to B)]\!] \to (\forall C.\, C \to C) \quad v \vdash_{\text{type}} D_0}{(\forall D_0.\, D_0 \to D_0) \to (\forall D_1.\, D_1 \to D_1) \preceq C_v[\![(B \to B)[_0 D_0/B]_2]\!] \to (\forall C.\, C \to C)}\ \forall\text{-Elim}$$

- Compute the substitution:

$$(B \to B)[{}_0D_0/B]_1 = D_0 \to D_0$$

- Resulting equation:

$$(\forall D_0.\, D_0 \to D_0) \to (\forall D_1.\, D_1 \to D_1) \preceq (D_0 \to D_0) \to (\forall C.\, C \to C)$$

3. Third substitution: C/D_1

- Apply the ∀-Elim rule:

$$\frac{(\forall D_0.\, D_0 \to D_0) \to C_\nu[\![\forall D_1.\, (D_1 \to D_1)]\!] \preceq (D_0 \to D_0) \to (\forall C.\, C \to C) \quad \nu \vdash_{\text{type}} C}{(\forall D_0.\, D_0 \to D_0) \to C_\nu[\![(D_1 \to D_1)[{}_0C/D_1]_2]\!] \preceq (D_0 \to D_0) \to (\forall C.\, C \to C)}\ \forall\text{-Elim}$$

- Compute the substitution:

$$(D_1 \to D_1)[{}_0C/D_1]_2 = C \to C$$

- Final equation:

$$(\forall D_0.\, D_0 \to D_0) \to (C \to C) \preceq (D_0 \to D_0) \to (\forall C.\, C \to C)$$

In the absence of quantifications, a non-circular substitution list can be transformed into a parallel substitution by replacing each substitution term with all later substitutions (see Section 2.4) [21]. However, if a list of substitutions contains quantified substitutions, a substitution may introduce new universally quantified variables, and these new variables might be used in a later substitution. Therefore, the order of the substitutions matter and in general they cannot be reduced to a parallel substitution.

5.3.2 Valid solution

To verify that a candidate solution is valid, we ensure that the equation obtained after applying the substitution list can be verified using the rules in Figure 5.3.7. The rules are explained in detail below.

$\underline{\forall}$-Elim$_+$: Eliminates a universal quantifier appearing in a positive (covariant) position by introducing it into the scope of the judgment using $\underline{\forall}A$, thus making the variable A available for future instantiation within the subsumption. (see Proposition 5.2.15)

match$_\preceq$: Reduces a subsumption between two constructed types according to the variance specified by the constructor's arity, to subsumptions between the arguments.

Elim$_\preceq$: Recognizes that a variable is trivially equal to itself and leaves the rest of the equation unchanged.

$\underline{\forall}$-Elim, $\underline{\exists}$-Elim: Eliminate quantifiers when their bound variables do not occur free in the body of the equation.

$$\frac{E[\![\forall A.S]\!]_+ \quad \text{equ}}{\underline{\forall} A.(E[\![S]\!]_+ \quad \text{equ})}\ \underline{\forall}\text{-Elim}_+$$

$$\frac{\text{arity}(F) = \{v_1, \ldots, v_n\},\ F(T_1, \ldots, T_n) \preceq F(T_1', \ldots, T_n')}{T_1 \preceq_{v_1} T_1', \ldots, T_n \preceq_{v_n} T_n'}\ \text{match}_\preceq$$

$$\frac{A \preceq A,\ E}{E}\ \text{Elim}_\preceq \qquad \frac{\underline{\forall} A.E \quad A \notin \text{fv}(E)}{E}\ \underline{\forall}\text{-Elim} \qquad \frac{\underline{\exists} A.E \quad A \notin \text{fv}(E)}{E}\ \underline{\exists}\text{-Elim}$$

Figure 5.3.7. Verifying solution rules - proof search direction

Example 5.3.8. In Example 5.3.6, we applied the following substitution list:

$$[\forall D.\, D \to D/A, \quad D_0/B, \quad C/D_1].$$

on the equation:

$$(\forall A.\, A \to A) \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C)$$

which resulted in:

$$(\forall D_0.\, D_0 \to D_0) \to (C \to C) \preceq (D_0 \to D_0) \to (\forall C.\, C \to C)$$

Now we want to verify this equation using the rules defined in Figure 5.3.7:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\forall D_0.\, D_0 \to D_0) \to (C \to C) \preceq (D_0 \to D_0) \to (\forall C.\, C \to C)}{D_0 \to D_0 \preceq \forall D_0.\, D_0 \to D_0}\ \text{match}_\preceq}{\underline{\forall} D_0.\, D_0 \to D_0 \preceq D_0 \to D_0}\ \underline{\forall}\text{-Elim}_+}{\underline{\forall} D_0.\, D_0 \preceq D_0}\ \text{match}_\preceq}{\underline{\forall} D_0.}\ \text{Elim}_\preceq}{}\ \underline{\forall}\text{-Elim} \qquad \dfrac{\dfrac{\dfrac{\dfrac{C \to C \preceq \forall C.\, C \to C}{\underline{\forall} C.\, C \to C \preceq C \to C}\ \underline{\forall}\text{-Elim}_+}{\underline{\forall} C.\, C \preceq C}\ \text{match}_\preceq}{\underline{\forall} C.}\ \text{Elim}_\preceq}{}\ \underline{\forall}\text{-Elim}$$

Therefore, this substitution list is valid and can be verified based on the rules in Figure 5.3.7.

5.4 From valid solutions to system S derivations

We want to prove that the solutions obtained for the generated equations of a term can be used to construct a derivation in the type-checking rules of System S.

Theorem 5.4.1. If the generated type equations for a closed term t has a solution σ, then t can be typed in system S as $\Gamma[\sigma] \vdash t : Q[\sigma]$.

Proof. To verify that Theorem 5.4.1 holds for every rule, we perform structural induction over the generating equation system rules given in Figure 4.2.2.
At each step, the generating equations rules are placed in parallel with the system S type checking rules:

$$\frac{\Gamma \vdash t_1 : Q_1\langle E_1\rangle \quad \Gamma \vdash t_2 : Q_2\langle E_2\rangle}{\Gamma \vdash t : Q\langle E\rangle}$$

generating equation rule

$$\frac{\Gamma\ [\sigma] \vdash t_1 : Q_1\ [\sigma] \quad \Gamma\ [\sigma] \vdash t_2 : Q_2\ [\sigma]}{\Gamma\ [\sigma] \vdash t : Q\ [\sigma]}$$

system S

We assume that $E_1[\sigma]$ and $E_2[\sigma]$ are valid, and that there is a valid proof of $\Gamma\ [\sigma] \vdash t_1 : Q_1\ [\sigma]$ and $\Gamma\ [\sigma] \vdash t_2 : Q_2\ [\sigma]$ in the type-checking rules of System S. We then need to show that if $E[\sigma]$ holds, this substitution justifies a valid use of the corresponding rule in the type-checking rules of System S for $\Gamma\ [\sigma] \vdash t : Q\ [\sigma]$.

var: If the equation $\langle A \preceq Q\rangle[\sigma]$ holds, then we must show that the following application of the T-Var rule is valid:(see Definition 5.3.1)

$$\frac{}{y : A \vdash y : Q \quad \langle A \preceq Q\rangle}\ \text{var}$$

generating equation rule

$$\frac{(A \preceq Q)[\sigma]}{y : A[\sigma] \vdash y : Q[\sigma]}\ \text{T-Var}$$

system S

$(A \preceq Q)[\sigma]$ holds, therefore T-Var is valid, and the base case holds.

annotation: By assumption $E_1[\sigma]$ holds, and $\Gamma\ [\sigma] \vdash t : A\ [\sigma]$ is valid in system S. We need to show that if the equation $\langle \underline{\exists} B.B = Q, A = Q, E_1\rangle[\sigma]$ holds, then the following application of the T-Annot rule is valid:

$$\frac{\Gamma \vdash t : B \ \langle E_1 \rangle}{\Gamma \vdash (t :: A) \ : Q \quad \langle \underline{\exists} B.B = Q, A = Q, E_1 \rangle} \text{ annotation}$$

generating equation rule

$$\frac{\Gamma \ [\sigma] \vdash t : A \ [\sigma]}{\Gamma \ [\sigma] \vdash (t :: A) \ : Q \ [\sigma]} \text{ T-Annot}$$

system S

In order for σ to solve the equation $(\underline{\exists} B.B = Q, A = Q, E_1)$, it must contain the substitution $A/Q, A/B$. This ensures that the corresponding rule in System S takes the form

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash (t :: A) : A} \text{ T-Annot}$$

and therefore this case holds.

abs: By assumption $E_1[\sigma]$ holds, and $x : A, \Gamma_B[\sigma] \vdash t : B[\sigma]$ is valid in system S. We need to show that if the equation $\langle \underline{\exists} A, B.Q = A \to B, E_1 \rangle \ [\sigma]$ holds, then the following application of the T-Abs rule is valid:

$$\frac{x : A, \Gamma_B \vdash t : B \ \langle E_1 \rangle}{\Gamma \vdash \lambda x.t : Q \quad \langle \underline{\exists} A, B.Q = A \to B, E_1 \rangle} \text{ abs}$$

generating equation rule

$$\frac{x : A, \Gamma_B \ [\sigma] \vdash t : B \ [\sigma]}{\Gamma \vdash \lambda x.t : Q[\sigma]} \text{ T-Abs}$$

system S

In order for σ to solve the equation $Q = A \to B$, it must contain the substitution $(A \to B/Q)$. This ensures that the corresponding rule in System S takes the form

$$\frac{x : A[\sigma], \Gamma_B \vdash t : B[\sigma]}{\Gamma \vdash \lambda x.t : A[\sigma] \to B[\sigma]} \text{ T-Abs}$$

and therefore this case holds.

app: By assumption $E_1[\sigma], E_2[\sigma]$ hold, and $\Gamma_A[\sigma] \vdash t : A[\sigma], \Gamma_B[\sigma] \vdash u : B[\sigma]$ are valid in system S. We need to show that if the equation $\langle \underline{\exists} A, B.A = B \to Q, E_1, E_2 \rangle [\sigma]$ holds, then the following application of the T-App rule is valid:

$$\frac{\Gamma \vdash t : A \ \langle E_1 \rangle \quad \Gamma \vdash u : B \ \langle E_2 \rangle}{\Gamma \vdash t \ u : Q \quad \langle \underline{\exists} A, B.A = B \to Q, E_1, E_2 \rangle} \text{ app}$$

generating equation rule

$$\frac{\Gamma[\sigma_1] \vdash t : A[\sigma_1] \quad \Gamma[\sigma_2] \vdash u : B[\sigma_2]}{\Gamma[\sigma_0, \sigma_1, \sigma_2] \vdash t \ u : Q[\sigma_0, \sigma_1, \sigma_2]} \text{ T-App}$$

system S

For σ to solve the equation $A = B \to Q$, we require σ to contain $(B' \to Q'/A, B'/B, Q'/Q)$. This guarantees that the corresponding rule in System S is

$$\frac{\Gamma \vdash t : (B' \to Q')[\sigma] \quad \Gamma \vdash u : B'[\sigma]}{\Gamma \vdash t\ u : Q'[\sigma]}\ \text{T-App}$$

and thus the case holds.

let: The expression let $x = u$ in t is equivalent to $(\lambda x.\, t)\ u$. Since the cases for abstraction and application have already been established, this case follows directly.

□

Example 5.4.2. We consider the term discussed in Example 4.3.2:

$$\text{id} : \forall X.X \to X \vdash (\text{id True},\ \text{id } `A`)$$

The equations collected for this term, as described in Example 4.3.2, are as follows:

$$\begin{aligned}
&\langle \underline{\exists} 1.\ 1 \preceq 0, \\
&\quad \langle \underline{\exists} 2, 3.\ 1 = (2, 3), \\
&\qquad \langle \underline{\exists} 4, 5.\ 4 = 5 \to 2,\ \langle \forall X_4.\ X_4 \to X_4 \preceq 4, \quad \text{Bool} \preceq 5 \rangle, \\
&\qquad \langle \underline{\exists} 6, 7.\ 6 = 7 \to 3,\ \langle \forall X_6.\ X_6 \to X_6 \preceq 6, \quad \text{Char} \preceq 7 \rangle \\
&\quad \rangle \\
&\rangle
\end{aligned}$$

We assume that every universally quantified type in Γ is renamed with an index determined by its corresponding principal variable, which is unique at each step of the inference. This ensures consistent handling of quantifiers throughout the inference process and takes care of the α-renaming step prior to the solving equation stage.

a valid substitution list which can be verified for this is:

$$[5 \to 2/4, 7 \to 3/6, (2,3)/1, \text{Bool}/5, \text{Char}/7, \text{Bool}/X_4, \text{Char}/X_6, \text{Bool}/2, \text{Char}/3, (\text{Bool},\text{Char})/0]$$

We want to show that this substitution list if applied to the principle types at each step give a valid derivation in system S :

$$\dfrac{\dfrac{\dfrac{}{\mathrm{id} : \forall X_4.X_4 \to X_4 \vdash \mathrm{id} : 4}\ \text{var} \quad \dfrac{}{\mathrm{id} : \forall X_5.X_5 \to X_5 \vdash \mathrm{True} : 5}\ \text{var}}{\dfrac{\langle \forall X_4.X_4 \to X_4 \preceq 4 \rangle \qquad \langle \mathrm{Bool} \preceq 5 \rangle}{\mathrm{id} : \forall X_2.X_2 \to X_2 \vdash \mathrm{id\ True} : 2}\ \text{app}} \quad \dfrac{\dfrac{}{\mathrm{id} : \forall X_6.X_6 \to X_6 \vdash \mathrm{id} : 6}\ \text{var} \quad \dfrac{}{\mathrm{id} : \forall X_7.X_7 \to X_7 \vdash \text{`}A\text{`} : 7}\ \text{var}}{\dfrac{\langle \forall X_6.X_6 \to X_6 \preceq 6 \rangle \qquad \langle 7 = \mathrm{Char} \rangle}{\mathrm{id} : \forall X_3.X_3 \to X_3 \vdash \mathrm{id}\ \text{`}A\text{`} : 3}\ \text{app}}}{\dfrac{\langle \underline{\exists} 4, 5.4 = 5 \to 2 \rangle \qquad \langle \underline{\exists} 6, 7.6 = 7 \to 3 \rangle}{\mathrm{id} : \forall X_1.X_1 \to X_1 \vdash (\mathrm{id\ True},\ \mathrm{id}\ \text{`}A\text{`}) : 1}\ \text{tuple}}}{\dfrac{\langle \underline{\exists} 2, 3.1 = (2, 3) \rangle}{\mathrm{id} : \forall X_0.X_0 \to X_0 \vdash (\mathrm{id\ True},\ \mathrm{id}\ \text{`}A\text{`}) : 0}}$$

$$\langle \underline{\exists} 1.1 \preceq 0 \rangle$$

generating equations

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall X_4.X_4 \to X_4 \preceq \mathrm{Bool} \to \mathrm{Bool}}\ \text{var}}{\mathrm{id} : \forall X_4.X_4 \to X_4 \vdash \mathrm{id} : \mathrm{Bool} \to \mathrm{Bool}} \quad \dfrac{\dfrac{}{\mathrm{Bool} \preceq \mathrm{Bool}}\ \text{var}}{\mathrm{id} : \forall X_5.X_5 \to X_5 \vdash \mathrm{True} : \mathrm{Bool}}}{\mathrm{id} : \forall X_2.X_2 \to X_2 \vdash \mathrm{id\ True} : \mathrm{Bool}}\ \text{app} \quad \dfrac{\dfrac{\dfrac{}{\forall X_6.X_6 \to X_6 \preceq \mathrm{Char} \to \mathrm{Char}}\ \text{var}}{\mathrm{id} : \forall X_6.X_6 \to X_6 \vdash \mathrm{id} : \mathrm{Char} \to \mathrm{Char}} \quad \dfrac{\dfrac{}{\mathrm{Char} \preceq \mathrm{Char}}\ \text{var}}{\mathrm{id} : \forall X_7.X_7 \to X_7 \vdash \text{`}A\text{`} : \mathrm{Char}}}{\mathrm{id} : \forall X_3.X_3 \to X_3 \vdash \mathrm{id}\ \text{`}A\text{`} : \mathrm{Char}}\ \text{app}}{\mathrm{id} : \forall X_1.X_1 \to X_1 \vdash (\mathrm{id\ True},\ \mathrm{id}\ \text{`}A\text{`}) : (\mathrm{Bool}, \mathrm{Char})}\ \text{tuple}}{\mathrm{id} : \forall X_0.X_0 \to X_0 \vdash (\mathrm{id\ True},\ \mathrm{id}\ \text{`}A\text{`}) : (\mathrm{Bool}, \mathrm{Char})}$$

system S derivation

5.5 Summary

In this chapter, we proved that a valid list of substitutions for the equations generated by a term yields a type for which there exists a valid judgement in the System S. This establishes a key connection between the process of generating equations and System S: any valid substitution that satisfies the equations generated for a term, using the verifying rules defined in Figure 5.3.2, yields a derivation in system S.

Chapter 6

Type Inference

Given a set of generated equations from a term, we are looking for a type inference algorithm that uses the equations to generate a valid solution and thus a type of the term. The chapter, begins by introducing a small-step type inference system and proving that it is type inference sound, meaning that the solution to each simplified equation at any step also serves as a solution to the corresponding original equation prior to simplification. Furthermore, we prove that the small-step system, when it produces a solution, produces a valid one.

Building on the structure of the small-step system, we define a big-step type inference system, in which several small-step rules are applied in sequence. In order to show the relation to the Hindley–Milner system, we introduce a big-step system restricted to handling equalities, using the big-step type inference system.

Since the big-step system is derived from the small-step system, its type inference soundness follows immediately. The rules presented in the big-step inference are non-deterministic, therefore we define a deterministic algorithm for the type inference. Finally, as a sanity check, we prove that the type inference algorithm we present is capable of typing all terms that can be typed by CaMPL's type inference algorithm, which is equivalent to Hindley-Milner type system.

6.1 Small-step type inference system

The rules in Figure 6.1.2 define the small-step system for reasoning about type inequalities. In this system, each rule application constitutes a single small step. For example, at each step only one occurrence of a

type variable X is replaced, rather than all occurrences at once. Each step is annotated with a list of universal quantifiers, Q, recording the variables that cannot be substituted, and a solution $[\sigma]$, given as a list of substitutions, which is gradually accumulated as the rules are applied and, if the proof succeeds, can be used to solve the corresponding equation. In the following, these rules are explained in detail.

$\underline{\forall}$-Elim$_+$: This rule refers to the rules for verifying solutions defined in Figure 5.3.7. It applies when a universal quantifier appears in a positive quantification. It removes the quantifier $\forall A.S$ and continues the subsumption using the body S. To ensure that the quantified variable A remains fixed (i.e., cannot be instantiated), the entire subsumption is placed under the scope of an underlined quantifier $\underline{\forall}A$, and A is added to the universal types.

$$\frac{C[\![\forall A.S]\!]_+ \quad \text{eqn} \quad \langle A, Q.\sigma\rangle}{\underline{\forall}A.(C[\![S]\!]_+ \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\forall}\text{-Elim}_+$$

$\underline{\forall}$-Elim$_-$: This rule applies when a universal quantifier appears in a negative quantification. It removes the quantifier $\forall A.S$ and proceeds with the body S in the subsumption. To allow the variable A to be instantiated during this process, the entire subsumption is placed under the scope of an existential quantifier $\underline{\exists}A$. The context E' indicates that we might use quantified variables outside the inner context C with which to instantiate A.

$$\frac{(E'[\![C[\![\forall A.S]\!]_-]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(E'[\![\underline{\exists}A.C[\![S]\!]_-]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\forall}\text{-Elim}_-$$

$\underline{\exists}$-Elim$_{\preceq_+}$: This rule applies when a variable X appears in a positive (covariant) quantification within a larger equation E, and we know that $T \preceq X$. In this case, we replace the positive occurrence of X in E with T, yielding the new equation $E[\![T]\!]_+$.

$$\frac{(C[\![\underline{\exists}X.(T \preceq X, E[\![X]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists}X.(T \preceq X, E[\![T]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq_+}$$

$\underline{\exists}$-Elim$_{-\preceq}$: This rule applies when a variable X appears in a negative (contravariant) quantification within a larger equation E, and we know that $X \preceq T$. In this case, we replace the negative occurrences of X in E with T, yielding the new equation $E[\![T]\!]_-$.

$$\frac{(C[\![\underline{\exists} X.(X \preceq T, E[\![X]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists} X.(X \preceq T, E[\![T]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{-\preceq}$$

$\underline{\exists}$-Elim$_{+\preceq}$: This rule applies when a variable X appears in a positive (covariant) position inside a larger equation E, and we already have $X \preceq T$. Then we also add $T \preceq X$ to force $X = T$.

$$\frac{(C[\![\underline{\exists} X.(X \preceq T, E[\![X]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists} X.(T \preceq X, X \preceq T, E[\![T]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{+\preceq}$$

$\underline{\exists}$-Elim$_{\preceq-}$: : This rule applies when a variable X appears in a negative (contravariant) position inside a larger equation E, and we already have $T \preceq X$. Then we also add $X \preceq T$ to force $X = T$.

$$\frac{(C[\![\underline{\exists} X.(T \preceq X, E[\![X]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists} X.(T \preceq X, X \preceq T, E[\![T]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq-}$$

match$_{\preceq}$: This rule handles subsumption between two type expressions of the form $F(T_1, \ldots, T_n)$, where F is a type constructor of arity n. The rule applies if both types have the same outer constructor F, and when the arity of F is $[v_1, \ldots, v_n]$, where each $v_i \in \{+, -\}$, subsumption is then checked componentwise, taking the variance into account:

$$T_i \preceq_{v_i} T_i'$$

Here, $\preceq_+$ denotes standard subsumption $T_i \preceq T_i'$, and $\preceq_-$ denotes contravariant subsumption $T_i' \preceq T_i$. This ensures that the overall subsumption respects the variance declared for each parameter position of F.

$$\frac{\text{arity}(F) = [v_1, \ldots, v_n] \quad (C[\![F(T_1, \ldots, T_n) \preceq F(T_1', \ldots, T_n')]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![T_1 \preceq_{v_1} T_1', \ldots, T_n \preceq_{v_n} T_n')]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{match}_{\preceq}$$

If the main subsumption (i.e., $F(T_1, \ldots, T_n) \preceq F(T_1', \ldots, T_n')$) is bounded with quantifiers like $\underline{\forall}$ or $\underline{\exists}$, and the variable introduced by the quantifier does not appear in some of the decomposed subsumption (i.e., $T_i \preceq_{v_i} T_i'$), then the quantifier only needs to apply to the parts where the variable actually occurs.

However, it is important that if the bound variable appears in more than one decomposed subsumption, then the quantifier must be applied to all of those relations together.

Example 6.1.1. This example shows that a bound variable must be quantified over all subsumptions where it appears. We apply $\text{match}_{\preceq}$ to the equation

$$(\underline{\forall} X.\, B \preceq X),\ A \preceq C,$$

as follows:

$$\dfrac{\dfrac{\text{arity}(F) = [-,+] \quad (\underline{\forall} X.X \to A \preceq B \to C) \quad X \notin A,B,C}{(\underline{\forall} X.B \preceq X, A \preceq C)}\ \text{match}_{\preceq}}{(\underline{\forall} X.B \preceq X), A \preceq C}\ \underline{\forall}\text{-rescope}$$

Then in $\underline{\forall} X.\ (B \preceq X,\ A \preceq C)$, as X does not occur in A, B, or C, the quantifier $\underline{\forall}$ can be applied only on $B \preceq X$, so we get

$$(\underline{\forall} X.\, B \preceq X),\ A \preceq C$$

$\underline{\exists}$-Elim$_{\preceq\succeq}$: This rule says that if we have $T \preceq X$ and $X \preceq T'$, and $X \notin \text{fv}(E)$, then we can conclude $T \preceq T'$.

$$\dfrac{(C[\![\underline{\exists} X.\ (T \preceq X,\ X \preceq T',\ E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E) \quad \langle Q.T/X, \sigma\rangle}{(C[\![T \preceq T', E)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq\succeq}$$

In this rule, the induced substitution $[T/X]$ is added to the list of substitutions that form the solution.

Elim$_{\preceq}$: A type always subsumes itself. This rule removes the trivial constraint $A \preceq A$, as it is always valid. The rest of the constraint set E remains unchanged.

$$\dfrac{(C[\![A \preceq A,\ E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{Elim}_{\preceq}$$

$\underline{\exists}$-Elim$_{\succeq}$: : If X only appears on the right side of $T \preceq X$ and not in E, we can remove X.

$$\dfrac{(C[\![\underline{\exists} X.(T \preceq X, E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E) \quad \langle Q.T/X, \sigma\rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\succeq}$$

In this rule, the induced substitution $[T/X]$ is added to the list of substitutions that form the solution.

$\underline{\exists}$**-Elim$_\preceq$:** : If X only appears on the left side of $X \preceq T$ and not in E, we can remove X.

$$\frac{(C[\![\underline{\exists}X.(X \preceq T, E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E) \quad \langle Q.T/X, \sigma\rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_\preceq$$

In this rule, the induced substitution $[T/X]$ is added to the list of substitutions that form the solution.

$\underline{\forall}$**-Elim,** $\underline{\exists}$**-Elim** : If a quantified variable does not occur free in an equation, then the quantifier can be removed.

$$\frac{C[\![\forall A.S]\!]_+ \quad \text{eqn} \quad \langle A, Q.\sigma\rangle}{\underline{\forall}A.(C[\![S]\!]_+ \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\forall}\text{-Elim}_+ \qquad \frac{(E'[\![C[\![\forall A.S]\!]_-]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(E'[\![\underline{\exists}A.C[\![S]\!]_-]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\forall}\text{-Elim}_-$$

$$\frac{(C[\![\underline{\exists}X.(T \preceq X, E[\![X]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists}X.(T \preceq X, E[\![T]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq_+} \qquad \frac{(C[\![\underline{\exists}X.(X \preceq T, E[\![X]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists}X.(X \preceq T, E[\![T]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{-\preceq}$$

$$\frac{(C[\![\underline{\exists}X.(X \preceq T, E[\![X]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists}X.(T \preceq X, X \preceq T, E[\![T]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{+\preceq}$$

$$\frac{(C[\![\underline{\exists}X.(T \preceq X, E[\![X]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists}X.(T \preceq X, X \preceq T, E[\![T]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq-}$$

$$\frac{\text{arity}(F) = [v_1, \ldots, v_n] \quad (C[\![F(T_1, \ldots, T_n) \preceq F(T'_1, \ldots, T'_n)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![T_1 \preceq_{v_1} T'_1, \ldots, T_n \preceq_{v_n} T'_n]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{match}_\preceq$$

$$\frac{(C[\![\underline{\exists}X.(\ T \preceq X,\ X \preceq T',\ E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E) \quad \langle Q.T/X, \sigma\rangle}{(C[\![T \preceq T', E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq\succeq}$$

$$\frac{(C[\![\underline{\exists}X.(T \preceq X, E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E) \quad \langle Q.T/X, \sigma\rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_\succeq$$

$$\frac{(C[\![\underline{\exists}X.(X \preceq T, E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E) \quad \langle Q.T/X, \sigma\rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_\preceq$$

$$\frac{(C[\![A \preceq A,\ E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{Elim}_\preceq$$

$$\frac{A \notin \text{fv}(E) \quad C[\![\underline{\forall}A.E]\!] \quad \langle Q.\sigma\rangle}{C[\![E]\!] \quad \langle Q.\sigma\rangle}\ \underline{\forall}\text{-Elim} \qquad \frac{A \notin \text{fv}(E) \quad C[\![\underline{\exists}A.E]\!] \quad \langle Q.\sigma\rangle}{C[\![E]\!] \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}$$

Figure 6.1.2. Small-step inference rules - proof search direction

Definition 6.1.3. Type inference soundness. A general rule in a type inference system looks like:

$$\frac{E \quad \langle Q.T/X, \sigma \rangle}{E' \quad \langle Q.\sigma \rangle}$$

A type inference system is type inference sound if it can be proved that, for every rule, whenever σ can solve the antecedent, $[T/X, \sigma]$ can solve the conclusion of the rule (as we are in the proof search direction, the antecedent is at the bottom and the conclusion is at the top of each rule).

Theorem 6.1.4. The small-step inference system is type inference sound.

Proof. We must prove that, for every rule, a solution to the antecedent of that rule is also a solution to the conclusion of that rule:

$\underline{\forall}$-Elim$_+$: In the following inference rule:

$$\frac{C[\![\forall A.S]\!]_+ \quad \text{eqn} \quad \langle A, Q.\sigma \rangle}{\underline{\forall} A.(C[\![S]\!]_+ \quad \text{eqn}) \quad \langle Q.\sigma \rangle} \; \underline{\forall}\text{-Elim}_+$$

If there exists a solution σ that solves the antecedent of the inference rule, σ may include variables bound by a $\forall$ quantifier, in particular A. However, in the conclusion, A is assumed to have the largest possible scope, which ensures that σ remains a valid solution for the conclusion as well.

$\underline{\forall}$-Elim$_-$: In the following inference rule:

$$\frac{(E'[\![C[\![\forall A.S]\!]_-]\!] \quad \text{eqn}) \quad \langle Q.\sigma \rangle}{(E'[\![\underline{\exists} A.C[\![S]\!]_-]\!] \quad \text{eqn}) \quad \langle Q.\sigma \rangle} \; \underline{\forall}\text{-Elim}_-$$

If there exists a solution σ that solves the antecedent of the inference rule, σ may include variables that are available in the outer context E'. However, in the conclusion, these variables are also available in the context E', which ensures that σ remains a valid solution for the conclusion as well.

$\underline{\exists}$-Elim$_{\preceq_+}$: If there exists a solution σ to the antecedent of this inference rule, σ is also a solution to the conclusion:

$$\frac{(C[\![\underline{\exists} X.(T \preceq X, E[\![X]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma \rangle}{(C[\![\underline{\exists} X.(T \preceq X, E[\![T]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma \rangle} \; \underline{\exists}\text{-Elim}_{\preceq_+}$$

In the solution σ, the existential variable X is replaced by some type S. Thus, we need to show that:

$$\frac{T \preceq S,\ E[\![T]\!]_+[{}_iS/X]_{i+\#_X E}}{T \preceq S,\ E[\![S]\!]_+[{}_iS/X]_{i+\#_X E}}$$

The top results from applying the substitution S for X in the antecedent. The bottom results from applying the same substitution in the conclusion. Since $T \preceq S$ appears on both sides, it is enough to show:

$$\frac{T \preceq S,\ E[\![T]\!]_+[{}_iS/X]_{i+\#_X E}}{E[\![S]\!]_+[{}_iS/X]_{i+\#_X E}}$$

Since the substitution $[{}_iS/X]_{i+\#_X E}$ is the same on both sides, it suffices to show:

$$\frac{T \preceq S,\ E[\![T]\!]_+}{E[\![S]\!]_+}$$

which is proved in Lemma 6.1.6.

Therefore, if the rest of the substitutions in σ solve the antecedent, they will also solve the conclusion, since we are solving $E[{}_iS/X]_{i+\#_X E}$ in both cases.

$\exists$-Elim$_{-\preceq}$: This rule can be justified using the same reasoning as for the $\exists$-Elim$_{\preceq_+}$ rule.

$$\frac{(C[\![\exists X.(X \preceq T, E[\![X]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\exists X.(X \preceq T, E[\![T]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \exists\text{-Elim}_{-\preceq}$$

$\exists$-Elim$_{+\preceq}$: If there exists a solution σ to the antecedent of this inference rule, σ is also a solution to the conclusion:

$$\frac{(C[\![\exists X.(X \preceq T, E[\![X]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\exists X.(T \preceq X, X \preceq T, E[\![T]\!]_+)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \exists\text{-Elim}_{+\preceq}$$

In the solution σ, the existential variable X is replaced by some type S. We need to show that:

$$\frac{T \preceq S, S \preceq T,\ E[\![T]\!]_+[{}_iS/X]_{i+\#_X E}}{S \preceq T,\ E[\![S]\!]_+[{}_iS/X]_{i+\#_X E}}$$

The top results from applying the substitution S for X in the antecedent. The bottom results from applying the same substitution in the conclusion. Since $S \preceq T$ appears on both sides, it is enough to show:

$$\frac{T \preceq S,\ E[\![T]\!]_+[{}_iS/X]_{i+\#_X E}}{E[\![S]\!]_+[{}_iS/X]_{i+\#_X E}}$$

Since the substitution $[_i S/X]_{i+\#_X E}$ is the same on both sides, it suffices to show:

$$\frac{T \preceq S,\ E[\![T]\!]_+}{E[\![S]\!]_+}$$

which is proved in Lemma 6.1.6.

Therefore, if the rest of the substitutions in σ solve the antecedent, they will also solve the conclusion, since we are solving $E[_i S/X]_{i+\#_X E}$ in both cases.

$\underline{\exists}$-Elim$_{\preceq_}$: This rule can be justified using the same reasoning as for the $\underline{\exists}$-Elim$_{+\preceq}$ rule.

$$\frac{(C[\![\underline{\exists} X.(T \preceq X, E[\![X]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\underline{\exists} X.(T \preceq X, X \preceq T, E[\![T]\!]_-)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq\_}$$

match$_\preceq$: In the following rule:

$$\frac{\text{arity}(F) = [\nu_1, \ldots, \nu_n] \quad (C[\![F(T_1, \ldots, T_n) \preceq F(T'_1, \ldots, T'_n)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![T_1 \preceq_{\nu_1} T'_1, \ldots, T_n \preceq_{\nu_n} T'_n]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{match}_\preceq$$

If there is a list of substitutions σ that solves the antecedent of this rule, then by part 2 of Definition 5.3.2 (quantified substitution for constructor types), each substitution is applied componentwise to the arguments of the constructor. Therefore, if a list of substitutions works for the antecedent, it must also work for the arguments of the constructor, which form the conclusion.

For each substitution in σ, namely $[_i S/A]_{i+\#_A(F(T_1,\ldots,T_n)\preceq F(T'_1,\ldots,T'_n))}$):

Let

$$i_p = i + \sum_{q=1}^{p-1} \#_A(T_q),\ i'_p = i + \#_A(F(T_1, \ldots, T_n) + \sum_{q=1}^{p-1} \#_A(T'_q)$$

the following holds:

$$\frac{\text{arity}(F) = [\nu_1, \ldots, \nu_n] \quad F(T_1, \ldots, T_n) \preceq F(T'_1, \ldots, T'_n)[_i S/A]_{i+\#_A(F(T_1,\ldots,T_n)\preceq F(T'_1,\ldots,T'_n))}}{\begin{array}{l} T_1[_{i_1} S/A]_{i_1+\#_A(T_1)} \preceq_{\nu_1} T'_1[_{i'_1} S/A]_{i'_1+\#_A(T'_1)}, \ldots, \\ T_n[_{i_n} S/A]_{i_n+\#_A(T_n)} \preceq_{\nu_n} T'_n[_{i'_n} S/A]_{i'_n+\#_A(T'_n)} \end{array}}\ \text{match}_\preceq$$

$\underline{\exists}$-Elim$_{\preceq\succeq}$: In the following rule:

$$\frac{(C[\![\underline{\exists} X.\ (T \preceq X,\ X \preceq T',\ E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E \quad \langle Q.T/X, \sigma\rangle}{(C[\![T \preceq T', E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq\succeq}$$

If $(T \preceq T', E)\ [\sigma]$ holds, then we need to prove that $(T \preceq X,\ X \preceq T',\ E)\ [T/X, \sigma]$ also holds.

$$\frac{(T \preceq X,\ X \preceq T',\ E)\ [T/X, \sigma]}{(T \preceq T, T \preceq T', E)\ [\sigma]}$$

Then, $(T \preceq T', E)\ [\sigma]$ holds by assumption and $T \preceq T[\sigma]$ trivially holds.

Elim$_\preceq$: In the following rule:

$$\frac{(C[\![A \preceq A,\ E]\!] \quad \text{eqn}) \quad \langle Q.\sigma \rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma \rangle}\ \text{Elim}_\preceq$$

If a solution σ solves the antecedent E, then it also solves the conclusion $(A \preceq A,\ \ E)$, since the added equation $A \preceq A$ is trivially valid for any substitution σ. Therefore, σ that solves the antecedent also solves the conclusion.

$\underline{\exists}$-Elim$_\succeq$: In the following rule:

$$\frac{(C[\![\underline{\exists} X.(T \preceq X, E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E) \quad \langle Q.T/X, \sigma \rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma \rangle}\ \underline{\exists}\text{-Elim}_\succeq$$

If $E\ [\sigma]$ holds, then we need to prove that $(T \preceq X,\ E)\ [T/X, \sigma]$ also holds.

$$\frac{(T \preceq X,\ E)\ [T/X, \sigma]}{(T \preceq T, E)\ [\sigma]}$$

Then, $E\ [\sigma]$ holds by assumption and $T \preceq T[\sigma]$ trivially holds.

$\underline{\exists}$-Elim$_\preceq$: In the following rule:

$$\frac{(C[\![\underline{\exists} X.(X \preceq T, E)]\!] \quad \text{eqn}) \quad X \notin \text{fv}(E) \quad \langle Q.T/X, \sigma \rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma \rangle}\ \underline{\exists}\text{-Elim}_\preceq$$

If a solution σ solves the antecedent E, then the solution $[T/X, \sigma]$ can solve $X \preceq T, E$, in the conclusion.

□

Lemma 6.1.5.

$$\frac{S \preceq T \quad (C[\![\_]\!]_\nu \text{ type})}{(C[\![S]\!] \ \preceq_\nu \ C[\![T]\!])\ \text{eqn}}$$

Proof. We proceed by structural induction on the context C, which is formed using the rules in Figure 5.2.11.

hole type: In this case, the context is simply a hole:

$$C[\![\_]\!]_\nu = [\![\_]\!]_\nu$$

Therefore, we need to prove:

$$\frac{S \preceq T}{S \preceq T}$$

which holds and serves as the base case of the structural induction.

universal type: By the induction hypothesis, we assume that from $S \preceq T$ and $(C[\![\_]\!]_\nu \text{ type})$,

$$(C[\![S]\!] \preceq_\nu C[\![T]\!]) \quad \text{eqn}$$

is derivable.

The universal quantifier $\forall X.$ is simply added on top of both sides. Since it does not affect the varity ν of the context, the result follows directly from the rule.

$$\frac{\dfrac{S \preceq T \quad (C[\![\_]\!]_\nu \text{ type})}{(C[\![S]\!] \preceq_\nu C[\![T]\!]) \quad \text{eqn}}\ \text{induction hypothesis}}{(\forall X.\, C[\![S]\!] \preceq_\nu \forall X.\, C[\![T]\!]) \quad \text{eqn}}\ \text{universal type}$$

Thus, the lemma holds in this case.

type constructor: We need to prove the following derivation is valid.

$$\frac{\dfrac{S \preceq T \quad (C[\![\_]\!]_\nu \text{ type})}{\text{arity}(F) = [\nu_1, \ldots, \nu_n] \quad (C[\![S]\!] \preceq_\nu C[\![T]\!]) \quad \text{eqn}}\ \text{induction hypothesis}}{(F(T_1, \ldots, C[\![S]\!], \ldots, T_n) \preceq_{\nu.\nu_i} F(T_1, \ldots, C[\![T]\!], \ldots, T_n)) \quad \text{eqn}}\ \text{cons-context}$$

Using how the arity of a type constructor affects the subsumption of the arguments and the entire type:

$$\frac{\text{arity}(F) = [\nu_1, \ldots, \nu_n] \quad S \preceq T \quad T_i \text{ Type}}{F(T_1, \ldots, S, \ldots, T_n) \preceq_{\nu_i} F(T_1, \ldots, T, \ldots, T_n)}\ \text{match}_\preceq$$

is derivable. The cons-context rule generalizes $\text{match}_\preceq$ by allowing the types (i.e., S) to appear inside a type context $C[\![\_]\!]$ that has a variance ν. In this case, the variance at the constructor argument position becomes $\nu \cdot \nu_i$, where ν_i is the variance assigned to that position by the

constructor's arity. As a result, $v \cdot v_i$ determines how the overall subsumption between the two type constructors is affected.

$$\frac{\text{arity}(F) = [v_1, \ldots, v_n] \quad (C[\![S]\!] \preceq_v C[\![T]\!]) \quad (T_i \text{ Type})}{(F(T_1, \ldots, C[\![S]\!], \ldots, T_n) \preceq_{v \cdot v_i} F(T_1, \ldots, C[\![T]\!], \ldots, T_n)) \quad \text{eqn}} \text{ cons-context}$$

Therefore, the cons-context can be proved, and the lemma holds in this case.

Thus, in all cases, the lemma holds by structural induction on the context. □

Lemma 6.1.6. The following holds:

$$\frac{T \preceq S, \; E[\![T]\!]_+}{E[\![S]\!]_+}$$

Proof. To prove this, we proceed by structural induction on E using the rules in Figure 5.2.13.

empty equation: The base case is when E is empty, for which

$$\frac{T \preceq S, \; E[\![T]\!]_+}{E[\![S]\!]_+}$$

holds trivially.

lsubsumption: Suppose $E[\![T]\!]_+ = (W \preceq W'[\![T]\!]_+)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{T \preceq S \quad (E[\![T]\!]_+) \text{ eqn}}{T \preceq S \quad (W \preceq W'[\![T]\!]_+) \text{ eqn}} \; \mathrm{E}[\![T]\!]_+ := (W \preceq W'[\![T]\!]_+)}{T \preceq S \quad (W'[\![T]\!]_+) \text{ type} \quad (W \preceq W'[\![T]\!]_+) \text{ eqn}} \; \text{lsubsumption}_{\text{varity}}}{(W \preceq W'[\![T]\!]_+) \text{ eqn} \quad (W'[\![T]\!] \preceq W'[\![S]\!]) \text{ eqn}} \; \text{Lemma 6.1.5 } (v = +)}{(W \preceq W'[\![S]\!]_+) \text{ eqn}} \; \text{lsubsumption}_{\text{varity}}}{(E[\![S]\!]_+) \text{ eqn}} \; \mathrm{E}[\![S]\!]_+ := (W \preceq W'[\![S]\!]_+)$$

rsubsumption: Suppose $E[\![T]\!]_+ = (W[\![T]\!]_+ \preceq W')$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{T \preceq S \quad (E[\![T]\!]_+) \text{ eqn}}{T \preceq S \quad (W[\![T]\!]_+ \preceq W') \text{ eqn}} \; \mathrm{E}[\![T]\!]_+ := (W[\![T]\!]_+ \preceq W')}{T \preceq S \quad (W[\![T]\!]_-) \text{ type} \quad (W[\![T]\!]_+ \preceq W') \text{ eqn}} \; \text{rsubsumption}_{\text{varity}}}{(W[\![S]\!] \preceq W[\![T]\!]) \text{ eqn} \quad (W[\![T]\!]_+ \preceq W') \text{ eqn}} \; \text{Lemma 6.1.5 } (v = -)}{(W[\![S]\!]_+ \preceq W') \text{ eqn}} \; \text{rsubsumption}_{\text{varity}}}{(E[\![S]\!]_+) \text{ eqn}} \; \mathrm{E}[\![S]\!]_+ := (W[\![S]\!]_+ \preceq W')$$

conjunction: Suppose $E[\![T]\!]_+ = E'', E'[\![T]\!]_+$

By induction hypothesis we know:

$$\frac{T \preceq S,\ E'[\![T]\!]_+}{E'[\![S]\!]_+}$$

holds. Therefore:

$$\dfrac{\dfrac{\dfrac{T \preceq S \quad (E[\![T]\!]_+)\ \text{eqn}}{T \preceq S \quad (E'', E'[\![T]\!]_+)\ \text{eqn}}\ E[\![T]\!]_+ := E'', E'[\![T]\!]_+ \text{ induction hypothesis}}{(E'', E'[\![S]\!]_+)\ \text{eqn}}}{(E[\![S]\!]_+)\ \text{eqn}}\ \text{conjunction}$$

forall: Suppose $E[\![T]\!]_+ = \underline{\forall} X.E'[\![T]\!]_+$

By induction hypothesis we know:

$$\frac{T \preceq S,\ E'[\![T]\!]_+}{E'[\![S]\!]_+}$$

holds. Therefore:

$$\dfrac{\dfrac{\dfrac{T \preceq S \quad (E[\![T]\!]_+)\ \text{eqn}}{T \preceq S \quad (\underline{\forall} X.E'[\![T]\!]_+)\ \text{eqn}}\ E[\![T]\!]_+ := \underline{\forall} X.E'[\![T]\!]_+ \text{ induction hypothesis}}{(\underline{\forall} X.E'[\![S]\!]_+)\ \text{eqn}}}{(E[\![S]\!]_+)\ \text{eqn}}\ \text{forall}$$

existentials: Suppose $E[\![T]\!]_+ = \underline{\exists} X.E'[\![T]\!]_+$

By induction hypothesis we know

$$\frac{T \preceq S,\ E'[\![T]\!]_+}{E'[\![S]\!]_+}$$

holds. Therefore:

$$\dfrac{\dfrac{\dfrac{T \preceq S \quad (E[\![T]\!]_+)\ \text{eqn}}{T \preceq S \quad (\underline{\exists} X.E'[\![T]\!]_+)\ \text{eqn}}\ E[\![T]\!]_+ := \underline{\exists} X.E'[\![T]\!]_+ \text{ induction hypothesis}}{(\underline{\exists} X.E'[\![S]\!]_+)\ \text{eqn}}}{(E[\![S]\!]_+)\ \text{eqn}}\ \text{existentials}$$

□

Lemma 6.1.7. The following holds:

$$\frac{T \preceq S,\ E[\![S]\!]_-}{E[\![T]\!]_-}$$

This lemma can be proved in the same way as Lemma 6.1.6 is proved.

Theorem 6.1.8. The substitution list obtained from a proof in the small-step inference system is always valid.

Proof. The small-step inference system contains the verification rules of Figure 5.3.7 with additional rules that produce substitutions. Thus, the verification of a solution is always performed using the same rules. This means, if the small-step inference system produces a solution, this solution is valid because it has been verified using the rules of Figure 5.3.7. □

Remark 6.1.9. If an inference system is type inference sound, this does not mean that every solution to the original equations is necessarily a solution to the simplified equations. For example, consider the following:

$$\frac{\underline{\exists} X.\ (\forall A.\ A) \preceq X,\ X \to X \preceq X \to X \quad \langle\rangle}{\underline{\exists} X.\ (\forall A.\ A) \preceq X,\ X \to X \preceq X \to (\forall A.\ A) \quad \langle\rangle}\ \underline{\exists}\text{-Elim}_{\preceq_+}$$

Then, $\sigma = [\text{Char}/X]$ will be a solution to the conclusion of the above example, but it is definitely not a solution to the antecedent. Applying $\sigma = [\text{Char}/X]$ to the antecedent:

$$\frac{\underline{\exists} X.\ (\forall A.\ A) \preceq \text{Char},\ \text{Char} \to \text{Char} \preceq \text{Char} \to (\forall A.\ A) \quad \langle\rangle}{(\forall A.\ A) \preceq \text{Char},\ \text{Char} \preceq \text{Char},\ \text{Char} \preceq (\forall A.\ A) \quad \langle\rangle}\ \exists\text{-Unused}$$

Since $\text{Char} \preceq (\forall A.\ A)$ does not hold, the solution that works for the conclusion does not work for the antecedent.

Remark 6.1.10. Note that $\underline{\forall}$-Elim$_-$ converts negatively quantified type variables into existential ones. While $\underline{\forall}$-Elim$_+$ pulls universal quantifiers outward, bringing their positive quantified bound variables into scope.

In some cases, both types of rules may be applicable. However, applying $\underline{\forall}$-Elim$_+$ first is usually preferable, as it introduces variables that can be used in the instantiation of existential variables.

If $\underline{\forall}$-Elim$_-$ are applied before bringing the foralls into scope, type inference may fail due to lack of access to universaly quantified type variables needed for instantiation.

Consider the equation:

$$\forall A_1.\forall B_1.A_1 \to (B_1 \to B_1) \preceq (\forall A_2.A_2) \to (\forall B_2.B_2 \to B_2)$$

If we apply the $\underline{\forall}$-Elim$_-$ rule directly, without checking whether the $\underline{\forall}$-Elim$_+$ rule is applicable, the derivation proceeds as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall A_1.\,\forall B_1.\,A_1 \to (B_1 \to B_1) \preceq (\forall A_2.\,A_2) \to (\forall B_2.\,B_2 \to B_2) \quad \langle .A_1/A_2\rangle}{\underline{\exists} A_1.\,(\forall B_1.\,A_1 \to (B_1 \to B_1) \preceq (\forall A_2.\,A_2) \to (\forall B_2.\,B_2 \to B_2)) \quad \langle .A_1/A_2\rangle}\,\underline{\forall}\text{-Elim}_-}{\underline{\exists} A_1.\,\underline{\exists} B_1.\,(A_1 \to (B_1 \to B_1) \preceq (\forall A_2.\,A_2) \to (\forall B_2.\,B_2 \to B_2)) \quad \langle .A_1/A_2\rangle}\,\underline{\forall}\text{-Elim}_-}{\dfrac{\dfrac{\underline{\exists} A_1.\,(\forall A_2.\,A_2 \preceq A_1)\langle .A_1/A_2\rangle}{\underline{\exists} A_1.\,\underline{\exists} A_2.\,(A_2 \preceq A_1) \quad \langle .A_1/A_2\rangle}\,\underline{\forall}\text{-Elim}_-}{\underline{\exists} A_1.\quad \langle\rangle}\,\underline{\exists}\text{-Elim}_\preceq \qquad \dfrac{\dfrac{\underline{\exists} B_1.\,(B_1 \to B_1 \preceq \forall B_2.\,B_2 \to B_2) \quad \langle\rangle}{\underline{\exists} B_1.\,\underline{\forall} B_2.\,(B_1 \to B_1 \preceq B_2 \to B_2) \quad \langle\rangle}\,\underline{\forall}\text{-Elim}_+}{\underline{\exists} B_1.\,\underline{\forall} B_2.\,(B_2 \preceq B_1,\ B_1 \preceq B_2) \quad \langle\rangle}\,\text{match}_\preceq}\,\underline{\exists}\text{-Unused} \qquad \text{Fails}}\,\text{match}_\preceq$$

This derivation fails because the subsumption becomes guarded by a universal quantifier $\underline{\forall} B_2$, which prevents substitution for B_2. As a result, the system is unable to resolve the judgment, even though a correct derivation exists. The correct approach is to apply the $\underline{\forall}$-Elim$_+$ rule first, as shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall A_1.\,\forall B_1.\,A_1 \to (B_1 \to B_1) \preceq (\forall A_2.\,A_2) \to (\forall B_2.\,B_2 \to B_2) \quad \langle B_2.A_1/A_2, B_2/B_1\rangle}{\underline{\forall} B_2.\,(\forall A_1.\,\forall B_1.\,A_1 \to (B_1 \to B_1) \preceq (\forall A_2.\,A_2) \to (B_2 \to B_2)) \quad \langle .A_1/A_2, B_2/B_1\rangle}\,\underline{\forall}\text{-Elim}_+}{\underline{\forall} B_2.\,\underline{\exists} A_1.\,(\forall B_1.\,A_1 \to (B_1 \to B_1) \preceq (\forall A_2.\,A_2) \to (B_2 \to B_2)) \quad \langle .A_1/A_2, B_2/B_1\rangle}\,\underline{\forall}\text{-Elim}_-}{\underline{\forall} B_2.\,\underline{\exists} A_1.\,\underline{\exists} B_1.\,(A_1 \to (B_1 \to B_1) \preceq (\forall A_2.\,A_2) \to (B_2 \to B_2)) \quad \langle .A_1/A_2, B_2/B_1\rangle}\,\underline{\forall}\text{-Elim}_-}{\dfrac{\dfrac{\underline{\exists} A_1.\,(\forall A_2.\,A_2 \preceq A_1) \quad \langle .A_1/A_2\rangle}{\underline{\exists} A_1.\,\underline{\exists} A_2.\,(A_2 \preceq A_1) \quad \langle .A_1/A_2\rangle}\,\underline{\forall}\text{-Elim}_-}{\underline{\exists} A_1.\quad \langle\rangle}\,\underline{\exists}\text{-Elim}_\preceq \qquad \dfrac{\dfrac{\underline{\forall} B_2.\,\underline{\exists} B_1.\,(B_1 \to B_1 \preceq B_2 \to B_2) \quad \langle .B_2/B_1\rangle}{\underline{\forall} B_2.\,\underline{\exists} B_1.\,(B_2 \preceq B_1,\ B_1 \preceq B_2) \quad \langle .B_2/B_1\rangle}\,\text{match}_\preceq}{\dfrac{\underline{\forall} B_2.\,(B_2 \preceq B_2) \quad \langle\rangle}{\underline{\forall} B_2.\quad \langle\rangle}\,\text{Elim}_\preceq}\,\underline{\forall}\text{-Elim}}\,\underline{\exists}\text{-Unused} \qquad \underline{\exists}\text{-Elim}_{\preceq\succeq}}\,\text{match}_\preceq$$

This example demonstrates the importance of pulling out positively quantified $\forall$s before turning negatively quantified variables into $\underline{\exists}$.

6.2 Big-step type inference system

Each rule of the big-step inference system can be derived by applying multiple steps of the small-step inference system. We now introduce the rules of the big-step inference system (see Figure 6.2.1) and discuss how each rule is derived from the small-step inference rules:

$\underline{\forall}$-Elim$_+$: This rule comes from the small-step inference system. It removes a universal quantifier when the type appears in a positive position by putting the variable in scope.

$\underline{\forall}$-Elim$_{-}$: This rule comes from the small-step inference system. It replaces a universally quantified type in a negative position with an existential variable to continue solving the equation.

$\underline{\exists}$-Elim$_{\succeq}$: This rule results from applying the $\underline{\exists}$-Elim$_{\preceq_+}$ rule (defined in Figure 6.1.2) repeatedly until all positive occurrences of the variable X in E have been substituted. We denote this substitution sequence by $E([_iT/X_+]_{i+\#_{X_+}E})$, which ensures that all positively occurring Xs in E are replaced by T.

Once this is done, we apply the $\underline{\exists}$-Elim$_{\preceq_-}$ rule to start substituting the remaining negative occurrences of X. This step may also require multiple applications. After all occurrences of X, both positive and negative, have been substituted, we apply the $\underline{\exists}$-Elim$_{\preceq\succeq}$ rule from the small-step inference system to remove the existential quantifier and complete the substitution.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(C[\![\underline{\exists}X.\,(T \preceq X,\ E)]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}{(C[\![\underline{\exists}X.\,(T \preceq X,\ E([_iT/X_+]_{i+\#_{X_+}E}))]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq_+}{}^{*}}{(C[\![\underline{\exists}X.\,(T \preceq X,\ X \preceq T,\ E([_iT/X_+]_{i+\#_{X_+}E}))]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}\ \underline{\exists}\text{-Elim}_{-\preceq}}{(C[\![\underline{\exists}X.\,(T \preceq X,\ X \preceq T,\ E([_iT/X_+]_{i+\#_{X_+}E}[_iT/X_-]_{i+\#_{X_-}E}))]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq-}{}^{*}}{(C[\![(T \preceq T,\ E([_iT/X_+]_{i+\#_{X_+}E}[_iT/X_-]_{i+\#_{X_-}E}))]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq\succeq}}{(C[\![T \preceq T,\ E([_iT/X]_{i+\#_XE})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}$$

As $E([_iT/X_+]_{i+\#_{X_+}E}[_iT/X_-]_{i+\#_{X_-}E})$ is equivalent to $E([_iT/X]_{i+\#_XE})$, this entire process justifies the following derived rule:

$$\dfrac{(C[\![\underline{\exists}X.\,(T \preceq X,\ E)]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}{(C[\![(T \preceq X,\ E)([_iT/X]_{i+1+\#_XE})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\succeq}$$

which adds the induced substitution ,$[T/X]$, to the substitution list.

$\underline{\exists}$-Elim$_{\preceq}$: This rule can be justified using the same reasoning as $\underline{\exists}$-Elim$_{\succeq}$, and when applying it, the substitution list will be updated and $[T/X]$ will be added to it.

guess$_{\preceq}$: While applying $\underline{\exists}$-Elim$_{\preceq_+}$, defined in Figure 6.1.2, allows choosing any T_k, this rule ensures that the chosen T_k is subsumed by all others, making it a valid substitution. The derivation using the small-step inference system is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{l}(\exists k \in \{1,\ldots,n\}.\ \forall i \in \{1,\ldots,n\}.\ T_k \preceq T_i)\\ (C[\![\underline{\exists} X.\ X \preceq T_1,\ \ldots,\ X \preceq T_n,\ E]\!] \quad \text{eqn}) \quad \langle Q.T_k/X, \sigma\rangle\end{array}}{(C[\![(\underline{\exists} X.\ T_1 \preceq T_k,\ \ldots,\ T_k \preceq X,\ \ldots,\ T_n \preceq T_k,\ E)\ ([_iT_k/X_+]_{i+\#_{X_+}E})]\!] \quad \text{eqn}) \quad \langle Q.T_k/X, \sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq_+}{}^*}{(C[\![(\underline{\exists} X.\ T_1 \preceq T_k,\ \ldots,\ T_k \preceq X, X \preceq T_k,\ \ldots,\ T_n \preceq T_k,\ E)\ ([_iT_k/X_+]_{i+\#_{X_+}E})]\!] \quad \text{eqn}) \quad \langle Q.T_k/X, \sigma\rangle}\ \underline{\exists}\text{-Elim}_{-\preceq}}{(C[\![(\underline{\exists} X.\ T_1 \preceq T_k,\ \ldots,\ T_k \preceq X, X \preceq T_k,\ \ldots,\ T_n \preceq T_k,\ E)\ ([_iT_k/X_+]_{i+\#_{X_+}E}[_iT_k/X_-]_{i+\#_{X_-}E})]\!] \quad \text{eqn}) \quad \langle Q.T_k/X, \sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq -}{}^*}{(C[\![(T_1 \preceq T_k,\ \ldots,\ T_k \preceq T_k,\ \ldots,\ T_n \preceq T_k,\ E)\ ([_iT_k/X_+]_{i+\#_{X_+}E}[_iT_k/X_-]_{i+\#_{X_-}E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq \succeq}}{(C[\![(T_1 \preceq T_k,\ \ldots,\ T_n \preceq T_k,\ E)\ ([_iT_k/X]_{i+\#_X E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}$$

This justifies the following rule:

$$\dfrac{\begin{array}{l}(\exists k \in \{1,\ldots,n\}.\ \forall i \in \{1,\ldots,n\}.\ T_k \preceq T_i)\\ (C[\![\underline{\exists} X.\ X \preceq T_1,\ \ldots,\ X \preceq T_n,\ E]\!] \quad \text{eqn}) \quad \langle Q.T_k/X, \sigma\rangle\end{array}}{(C[\![(X \preceq T_1,\ \ldots,\ X \preceq T_n,\ E)\ ([_iT_k/X]_{i+n+\#_X E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{guess}_{\preceq}$$

Applying this rule adds $[T_k/X]$ to the substitution list.

guess$_{\succeq}$: This rule can be justified using the same reasoning as guess$_{\preceq}$, and when applying it, the substitution list will be updated and $[T_k/X]$ will be added to it.

$\underline{\forall}$-Elim: This rule comes from the small-step inference system. The same holds for $\underline{\exists}$-Elim, Elim$_{\preceq}$, and match$_{\preceq}$.

$$\frac{C[\![\forall A.S]\!]_+ \quad \text{eqn} \quad \langle A, Q.\sigma\rangle}{\underline{\forall} A.(C[\![S]\!]_+ \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\forall}\text{-Elim}_+ \qquad \frac{(E'[\![C[\![\forall A.S]\!]_-]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(E'[\![\underline{\exists} A.C[\![S]\!]_-]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\forall}\text{-Elim}_-$$

$$\frac{(C[\![\underline{\exists} X.\,(T \preceq X,\ E)]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}{(C[\![(T \preceq X,\ E)([_iT/X]_{i+1+\#_X E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\succeq}$$

$$\frac{(C[\![\underline{\exists} X.\,(X \preceq T,\ E)]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}{(C[\![(X \preceq T,\ E)([_iT/X]_{i+1+\#_X E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{\preceq}$$

$$\frac{\begin{array}{l}(\exists k \in \{1,\ldots,n\}.\ \forall i \in \{1,\ldots,n\}.\ T_i \preceq T_k)\\ (C[\![\underline{\exists} X.\ T_1 \preceq X,\ \ldots,\ T_n \preceq X,\ E]\!] \quad \text{eqn}) \quad \langle Q.T_k/X, \sigma\rangle\end{array}}{(C[\![(T_1 \preceq X,\ \ldots,\ T_n \preceq X,\ E)\ ([_iT_k/X]_{i+n+\#_X E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{guess}_{\succeq}$$

$$\frac{\begin{array}{l}(\exists k \in \{1,\ldots,n\}.\ \forall i \in \{1,\ldots,n\}.\ T_k \preceq T_i)\\ (C[\![\underline{\exists} X.\ X \preceq T_1,\ \ldots,\ X \preceq T_n,\ E]\!] \quad \text{eqn}) \quad \langle Q.T_k/X, \sigma\rangle\end{array}}{(C[\![(X \preceq T_1,\ \ldots,\ X \preceq T_n,\ E)\ ([_iT_k/X]_{i+n+\#_X E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{guess}_{\preceq}$$

$$\frac{\text{arity}(F) = [\nu_1,\ldots,\nu_n] \quad (C[\![F(T_1,\ldots,T_n) \preceq F(T'_1,\ldots,T'_n)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![T_1 \preceq_{\nu_1} T'_1,\ldots,T_n \preceq_{\nu_n} T'_n]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{match}_{\preceq}$$

$$\frac{(C[\![A \preceq A,\ E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![E]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{Elim}_{\preceq}$$

$$\frac{A \notin \text{fv}(E) \quad C[\![\underline{\forall} A.E]\!] \quad \langle Q.\sigma\rangle}{C[\![E]\!] \quad \langle Q.\sigma\rangle}\ \underline{\forall}\text{-Elim} \qquad \frac{A \notin \text{fv}(E) \quad C[\![\underline{\exists} A.E]\!] \quad \langle Q.\sigma\rangle}{C[\![E]\!] \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}$$

Figure 6.2.1. Big-step inference rules - proof search direction

Theorem 6.2.2. The big step type inference system is type inference sound.

Proof. The small-step inference rules are type inference sound (see Theorem 6.1.4). Since the big-step inference rules can be derived from the small-step inference rules, big-step inference is also type inference sound. □

Theorem 6.2.3. The substitution list obtained from a proof in the big-step inference system is always valid.

Proof. The verification rules in Figure 5.3.7 are part of the big-step inference system. In the big-step inference system some rules are responsible for generating substitutions, but the verification itself is always performed using the same rules as defined in Figure 5.3.7. This means, if the big-step infer-

ence system produces a solution, this solution is always valid because it is verified using the rules of Figure 5.3.7. □

Example 6.2.4. We aim to perform type inference for the following subsumption:

$$\forall A.\, A \to A \;\preceq\; (\forall B.\, B \to B) \to (\forall C.\, C \to C)$$

The corresponding derivation is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall A.\, A \to A \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C) \quad \langle C.\forall B.B \to B/A, B_0/B, C/B_1\rangle}{(\underline{\forall} C.(\forall A.\, A \to A) \preceq (\forall B.\, B \to B) \to (C \to C))\langle .\forall B.B \to B/A, B_0/B, C/B_1\rangle}\,\underline{\forall}\text{-Elim}_+}{(\underline{\forall} C.\underline{\exists} A.(A \to A \preceq (\forall B.\, B \to B) \to (C \to C))\langle .\forall B.B \to B/A, B_0/B, C/B_1\rangle}\,\underline{\forall}\text{-Elim}_-}{(\underline{\forall} C.\underline{\exists} A.((\forall B.\, B \to B) \preceq A, A \preceq (C \to C)) \quad \langle .\forall B.B \to B/A, B_0/B, C/B_1\rangle}\,\text{match}_\preceq}{(\underline{\forall} C.((\forall B.\, B \to B) \preceq (\forall B_0.\, B_0 \to B_0), (\forall B_1.\, B_1 \to B_1) \preceq (C \to C)) \quad \langle .B_0/B, C/B_1\rangle}\,\underline{\exists}\text{-Elim}_\preceq}{}\,\underline{\forall}\text{-reassoc}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\forall B.\, B \to B) \preceq (\forall B_0.\, B_0 \to B_0) \quad \langle .B_0/B\rangle}{\underline{\forall} B_0.\, (\forall B.\, B \to B) \preceq (\, B_0 \to B_0) \quad \langle .B_0/B\rangle}\,\underline{\forall}\text{-Elim}_+}{\underline{\forall} B_0.\underline{\exists} B.\, (B \to B) \preceq (\, B_0 \to B_0) \quad \langle .B_0/B\rangle}\,\underline{\forall}\text{-Elim}_-}{\underline{\forall} B_0.\underline{\exists} B.\, (B_0 \preceq B), (B \preceq B_0) \quad \langle .B_0/B\rangle}\,\text{match}_\preceq}{\underline{\forall} B_0.\, (B_0 \preceq B_0), (B_0 \preceq B_0) \quad \langle\rangle}\,\underline{\exists}\text{-Elim}_\preceq}{\underline{\forall} B_0.\, (B_0 \preceq B_0) \quad \langle\rangle}\,\text{Elim}_\preceq}{\underline{\forall} B_0 \quad \langle\rangle}\,\underline{\forall}\text{-Elim}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{(\underline{\forall} C.(\forall B_1.\, B_1 \to B_1) \preceq (C \to C)) \quad \langle .C/B_1\rangle}{\underline{\forall} C.\underline{\exists} B_1.\, (B_1 \to B_1) \preceq (\, C \to C) \quad \langle .C/B_1\rangle}\,\underline{\forall}\text{-Elim}_-}{\underline{\forall} C.\underline{\exists} B_1.\, (C \preceq B_1), (B_1 \preceq C) \quad \langle .C/B_1\rangle}\,\text{match}_\preceq}{\underline{\forall} C.\, (C \preceq C), (C \preceq C) \quad \langle\rangle}\,\underline{\exists}\text{-Elim}_\preceq}{\underline{\forall} C.\, (C \preceq C) \quad \langle\rangle}\,\text{Elim}_\preceq}{\underline{\forall} C \quad \langle\rangle}\,\underline{\forall}\text{-Elim}$$

Hence, the solution obtained by applying this type inference will be $[\forall B.B \to B/A, B_0/B, C/B_1]$

To verify this solution, we apply the substitution rules in Figure 5.3.5 to the equation using the substitution list:

1. First substitution: $\forall B.\, B \to B/A$

   - We apply the ∀-Elim rule (see Figure 5.3.5):

$$\dfrac{C_v[\![\forall A.\, (A \to A)]\!] \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C) \quad v \vdash_{\text{type}} (\forall B.\, B \to B)}{C_v[\![(A \to A)[_0(\forall B.\, B \to B)/A]_2]\!] \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C)}\,\forall\text{-Elim}$$

   - Compute the substitution:

$$A \to A[_0(\forall B.\, B \to B)/A]_2 = (\forall B.\, B \to B)_0 \to (\forall B.\, B \to B)_1$$

- Using the definition for $(T)_i$ as defined in Definition 7.2:

$$= (\forall B_0.\, B_0 \to B_0) \to (\forall B_1.\, B_1 \to B_1)$$

- Resulting equation:

$$(\forall B_0.\, B_0 \to B_0) \to (\forall B_1.\, B_1 \to B_1) \preceq (\forall B.\, B \to B) \to (\forall C.\, C \to C)$$

2. Second substitution: B_0/B

- Apply the ∀-Elim rule:

$$\frac{(\forall B_0.\, B_0 \to B_0) \to (\forall B_1.\, B_1 \to B_1) \preceq C_\nu[\![\forall B.\, (B \to B)]\!] \to (\forall C.\, C \to C) \quad \nu \vdash_{\text{type}} B_0}{(\forall B_0.\, B_0 \to B_0) \to (\forall B_1.\, B_1 \to B_1) \preceq C_\nu[\![(B \to B)[{}_0B_0/B]_2]\!] \to (\forall C.\, C \to C)}\ \forall\text{-Elim}$$

- Compute the substitution:

$$(B \to B)[{}_0B_0/B]_1 = B_0 \to B_0$$

- Resulting equation:

$$(\forall B_0.\, B_0 \to B_0) \to (\forall B_1.\, B_1 \to B_1) \preceq (B_0 \to B_0) \to (\forall C.\, C \to C)$$

3. Third substitution: C/B_1

- Apply the ∀-Elim rule:

$$\frac{(\forall B_0.\, B_0 \to B_0) \to C_\nu[\![\forall B_1.\, (B_1 \to B_1)]\!] \preceq (B_0 \to B_0) \to (\forall C.\, C \to C) \quad \nu \vdash_{\text{type}} C}{(\forall B_0.\, B_0 \to B_0) \to C_\nu[\![(B_1 \to B_1)[{}_0C/B_1]_2]\!] \preceq (B_0 \to B_0) \to (\forall C.\, C \to C)}\ \forall\text{-Elim}$$

- Compute the substitution:

$$(B_1 \to B_1)[{}_0C/B_1]_2 = C \to C$$

- Final equation:

$$(\forall B_0.\, B_0 \to B_0) \to (C \to C) \preceq (B_0 \to B_0) \to (\forall C.\, C \to C)$$

This equation can be verified using the rules defined in Figure 5.3.7, as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\forall B_0.\ B_0 \to B_0) \to (C \to C) \preceq (B_0 \to B_0) \to (\forall C.\ C \to C)}{B_0 \to B_0 \preceq \forall B_0.\ B_0 \to B_0}\ \text{match}_{\preceq}}{\underline{\forall} B_0.\ B_0 \to B_0 \preceq B_0 \to B_0}\ \underline{\forall}\text{-Elim}_{+}}{\underline{\forall} B_0.\ B_0 \preceq B_0}\ \text{match}_{\preceq}}{\underline{\forall} B_0.}\ \text{Elim}_{\preceq}}{}\ \underline{\forall}\text{-Elim} \qquad \dfrac{\dfrac{\dfrac{C \to C \preceq \forall C.\ C \to C}{\underline{\forall} C.\ C \to C \preceq C \to C}\ \underline{\forall}\text{-Elim}_{+}}{\underline{\forall} C.\ C \preceq C}\ \text{match}_{\preceq}}{\underline{\forall} C.}\ \text{Elim}_{\preceq}}{}\ \underline{\forall}\text{-Elim}$$

6.3 The big-step system for equalities

To show the relation to Hindley-Milner type system, we use the big-step inference system rules, which handle inequalities, to define the big-step system for equalities, which handles type equalities by interpreting an equality $T = S$ as the pair of inequalities $T \preceq S$ and $S \preceq T$. Each rule of the big-steps system for equalities is obtained by applying multiple steps of the big-step inference rules.

$\underline{\exists}$-Elim$_{=}$: This rule can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{(C[\![\underline{\exists} X.\,(X = T,\ E(X))]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}{(C[\![\underline{\exists} X.\,(X \preceq T,\ T \preceq X,\ E(X))]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}\ \underline{\exists}\text{-Elim}_{\succeq}}{(C[\![(X \preceq T,\ T \preceq X,\ E(X))([_iT/X]_{i+2+\#_X E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}}{(C[\![(T \preceq T,\ T \preceq T,\ E(X))([_iT/X]_{i+\#_X E})]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{Elim}_{\preceq}}{E(X)([_iT/X]_{i+\#_X E})}$$

match$_{=}$: This rule can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(C[\![F(T_1,\ldots,T_n) = F(T_1',\ldots,T_n')]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![F(T_1,\ldots,T_n) \preceq F(T_1',\ldots,T_n') \quad F(T_1',\ldots,T_n') \preceq F(T_1,\ldots,T_n)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle \quad \text{arity}(F) = [\nu_1,\ldots,\nu_n]}\ \text{match}_{\preceq}}{(C[\![T_1 \preceq_{\nu_1} T_1',\ldots,T_n \preceq_{\nu_n} T_n' \quad F(T_1',\ldots,T_n') \preceq F(T_1,\ldots,T_n)]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle \quad \text{arity}(F) = [\nu_1,\ldots,\nu_n]}\ \text{match}_{\preceq}}{(C[\![T_1 \preceq_{\nu_1} T_1',\ldots,T_n \preceq_{\nu_n} T_n' \quad T_1' \preceq_{\nu_1} T_1,\ldots,T_n' \preceq_{\nu_n} T_n]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}}{(C[\![(T_1 = T_1',\ldots,T_n = T_n')]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle \quad \text{arity}(F) = [\nu_1,\ldots,\nu_n]}}{(C[\![F(T_1,\ldots,T_n) = F(T_1',\ldots,T_n')]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}$$

Hence, the equality between two constructed types is derived using match$_{\preceq}$ rule.

$\forall_{=}$: The rule $\forall_{=}$ is based on the idea that two types are equal if they are subtypes of each other. Thus, $\forall A.\ S = T$ holds when both $\forall A.\ S \preceq T$ and $T \preceq \forall A.\ S$ are true.

$$\dfrac{(C[\![\forall A.S = T]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\forall A.S \preceq T, T \preceq \forall A.S]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}$$

$$\frac{(C[\![\underline{\exists} X.\,(X = T,\ E)]\!] \quad \text{eqn}) \quad \langle Q.T/X, \sigma\rangle}{(C[\![E[{}_iT/X]_{i+\#_X E}]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \underline{\exists}\text{-Elim}_{=}$$

$$\frac{(C[\![\quad \text{eqn}) F(T_1,\ldots,T_n) = F(T_1',\ldots,T_n')]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![T_1 = T_1' \quad \ldots \quad T_n = T_n']\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \text{match}_{=}$$

$$\frac{(C[\![\ \forall A.S = T]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}{(C[\![\forall A.S \preceq T \quad T \preceq \forall A.S]\!] \quad \text{eqn}) \quad \langle Q.\sigma\rangle}\ \forall_{=}$$

Figure 6.3.1. Big-step system for equalities - proof search direction

6.4 Type inference algorithm

In this section, we present a partial algorithm for type inference. The rules in Figure 6.2.1 are non-deterministic. The algorithm determines precisely which rules should be applied at each step.

The type inference algorithm has the following steps:

1. If an equality formula contains a universal quantifier, we rewrite the equality as two inequalities using the $\forall_=$ rule from Figure 6.3.1

2. Remove all the positive universal quantifiers from the type inequalities (see $\underline{\forall}$-Elim$_+$ in Figure 6.2.1). This results in an equation of the form:

$$\underline{\forall} X_1,\ldots,X_n.T_1 \preceq T_1',\ldots,T_n = T_n'$$

   where each $T_i \preceq T_i'$ only has negative universal quantifications.

3. Remove all the negative universal quantifiers from the type inequalities (see $\underline{\forall}$-Elim$_-$ in Figure 6.2.1). This results in a formula of the form:

$$\underline{\forall} X_1,\ldots,X_n.\underline{\exists} Y_1,\ldots,Y_n.T_1 \preceq T_1',\ldots,T_n = T_n'$$

   where each $T_i \preceq T_i'$ only has negative universal quantifications. Note that the order of existential quantifications is not fixed. (The eventual dependencies will require reordering in a compatible manner, to ensure non-circular substitution)

4. Given an equality or inequality between two terms, we match equalities using match= and inequalities using match$\preceq$. Repeatedly applying the matching rules to reduce the inequalities and equalities to the case in which one argument is a variable. The result is a formula of the form:

$$\underline{\forall} X_1, \ldots, X_n . \underline{\exists} Y_1, \ldots, Y_m . Z_1 \preceq T_1', T_2 \preceq Z_2, \ldots, Z_n = T_n'$$

where $Z_i \in X_1, \ldots, X_n, Y_1, \ldots, Y_m$.

5. Eliminate existentials using $\underline{\exists}$-Elim$_{\succeq}$, $\underline{\exists}$-Elim$_{\preceq}$, guess$_{\preceq}$, guess$_{\succeq}$, or $\underline{\exists}$-Elim, depending on whether the variable occurs in a single inequality, multiple inequalities, or an equality. This step may result in terms which can be matched. Keep repeating until there are no more opportunities for elimination. This, if successful, results in an equation of the form:

$$\underline{\forall} X_1, \ldots, X_n . \underline{\exists} Y_1, \ldots, Y_m . T \preceq Q$$

6. As the final step rebind the variables to obtain a type. For example:

$$\underline{\exists} A . A \to A \preceq Q \quad \text{becomes} \quad (\forall A . A \to A) \preceq Q$$

$$\underline{\forall} X . \underline{\exists} Y . X \to Y \preceq Q \quad \text{becomes} \quad (\forall X . X) \to (\forall Y . Y) \preceq Q .$$

6.5 Comparing type inference for S and basic CaMPL's type inference

We must prove that the type inference algorithm defined in section 6.4 properly extends the basic CaMPL inference system. In particular, we want to ensure that any term which is typable in CaMPL's type system remains typable with respect to the algorithm described above, so that the system we have introduced is stronger.

The rules in Figure 2.3.2 show the equation inference rules for a small sequential subset of the CaMPL language, and the rules in Figure 4.2.2 show the generating equation rules for system S.

Theorem 6.5.1. Let Γ be a context of closed types, and let t be a term. If t is typable in CaMPL's inference system, suppose Q and Q' are the principal types of t assigned in CaMPL's inference and inference rules for system S, respectively. Then $Q \preceq Q'$ holds.

Proof. To verify that Theorem 6.5.1 holds for all terms in the system, we proceed by structural induction on the terms, where the structure of terms corresponds to the typing rules of the system.

var: when t is a variable, say x, we use the variable rule in generating equations for both system S and CaMPL's:

$$\frac{}{\Gamma, x : A \vdash x : Q' \quad \langle A \preceq Q' \rangle}\ \text{var} \qquad \frac{}{\Gamma, x : A \vdash x : Q \quad \langle A = Q \rangle}\ \text{var}$$

system S CaMPL

In system S, the rule introduces a subsumption $A \preceq Q'$. In CaMPL's, the type Q is set equal to A. Therefore:

$$A = Q \quad \text{(from CaMPL)} \Rightarrow Q \preceq A \quad \text{(since = implies } \preceq\text{)}$$

$$A \preceq Q' \quad \text{(from system S)}$$

$$Q \preceq A,\ A \preceq Q' \Rightarrow Q \preceq Q' \quad \text{(transitivity)}$$

Therefore $Q \preceq Q'$ holds.

Hence, Theorem 6.5.1 holds in this case, which is the base case of the induction.

abs: When the term is an abstraction $\lambda x.\, t$, we apply the abstraction rule:

$$\frac{x : A,\ \Gamma \vdash t : B' \quad \langle E_1 \rangle}{\Gamma \vdash \lambda x.\, t : Q' \quad \langle \exists A, B'.\ Q' = A \to B',\ E_1 \rangle}\ \text{abs} \qquad \frac{x : A,\ \Gamma \vdash t : B \quad \langle E_1 \rangle}{\Gamma \vdash \lambda x.\, t : Q \quad \langle \exists A, B.\ Q = A \to B,\ E_1 \rangle}\ \text{abs}$$

system S CaMPL

By the inductive hypothesis, $B \preceq B'$, therefore:

$$\frac{\underline{\exists} A, B'.\ Q' = A \to B', B \preceq B'}{\underline{\exists} A.\ Q' = A \to B}\ \underline{\exists}\text{-Elim}_{\succeq}$$

$$Q = A \to B \quad \text{(from CaMPL)} \Rightarrow Q' = Q$$

$$Q' = Q \Rightarrow Q \preceq Q' \quad \text{(since = implies } \preceq\text{)}$$

app: When the term is an application $f\ t$, we apply the application rule:

$$\frac{\Gamma \vdash f : A' \quad \langle E_1 \rangle \qquad \Gamma \vdash t : B' \quad \langle E_2 \rangle}{\Gamma \vdash f\ t : Q' \quad \langle \exists A', B'.\ A' = B' \to Q',\ E_1, E_2 \rangle}\ \text{app} \qquad \frac{\Gamma \vdash f : A \quad \langle E_1 \rangle \qquad \Gamma \vdash t : B \quad \langle E_2 \rangle}{\Gamma \vdash f\ t : Q \quad \langle \exists A, B.\ A = B \to Q,\ E_1, E_2 \rangle}\ \text{app}$$

system S CaMPL

By inductive hypothesis $A \preceq A'$, $B \preceq B'$, therefore:

$$\dfrac{\dfrac{\underline{\exists} A', B'.\ A' = B' \to Q', B \preceq B', A \preceq A'}{\underline{\exists} A'.\ A' = B \to Q', A \preceq A'}\ \underline{\exists}\text{-Elim}_{\succeq}}{A = B \to Q}\ \underline{\exists}\text{-Elim}_{\succeq}$$

$$A = B \to Q \quad (\text{from CaMPL}) \Rightarrow Q' = Q$$

$$Q' = Q \Rightarrow Q \preceq Q' \quad (\text{since} = \text{implies} \preceq)$$

Let: When the term is a let-binding let $x = u$ in t, $B \preceq B'$, $C \preceq C'$

$$\frac{\Gamma, x : A \vdash t : B' \quad \langle E_1 \rangle \qquad \Gamma \vdash u : C' \quad \langle E_2 \rangle}{\begin{array}{c}\Gamma \vdash \text{let } x = u \text{ in } t : Q \\ \langle \exists A, B', C'.\ A = C',\ B' = Q',\ E_1,\ E_2 \rangle\end{array}}\ \text{let} \qquad \frac{\Gamma, x : A \vdash t : B \quad \langle E_1 \rangle \qquad \Gamma \vdash u : C \quad \langle E_2 \rangle}{\begin{array}{c}\Gamma \vdash \text{let } x = u \text{ in } t : Q \\ \langle \exists A, B, C.\ A = C,\ B = Q,\ E_1,\ E_2 \rangle\end{array}}\ \text{let}$$

system S CaMPL

By inductive hypothesis $B \preceq B'$, $C \preceq C'$, therefore:

$$\dfrac{\dfrac{\underline{\exists} A, B', C'.\ A = C', B' = Q', B \preceq B', C \preceq C'}{\underline{\exists} A, C'.\ A = C', B = Q', C \preceq C'}\ \underline{\exists}\text{-Elim}_{\succeq}}{\underline{\exists} A.A = C, B = Q'}\ \underline{\exists}\text{-Elim}_{\succeq}$$

$$B = Q \quad (\text{from CaMPL}) \Rightarrow Q' = Q$$

$$Q' = Q \Rightarrow Q \preceq Q' \quad (\text{since} = \text{implies} \preceq)$$

which confirms Theorem 6.5.1.

build-∀: This rule handles universally quantified types in system S, and such types are not typable in CaMPL's, therefore there is no corresponding rule in CaMPL to build-∀ rule in system S.

□

Example 6.5.2. We consider the term discussed in Example 4.3.2:

$$\mathrm{id} : \forall X.X \to X \vdash (\mathrm{id}\ \mathrm{True},\ \mathrm{id}\ `A`)$$

The equations collected for this term, as described in Example 4.3.2, are as follows:

$$\begin{array}{l}\Big\langle \underline{\exists} 1.\ 1 \preceq 0, \\ \quad \Big\langle \underline{\exists} 2, 3.\ 1 = (2,3), \\ \qquad \Big\langle \underline{\exists} 4, 5.\ 4 = 5 \to 2,\ \langle \forall X_0.\ X_0 \to X_0 \preceq 4, \quad \mathrm{Bool} \preceq 5 \rangle\,, \\ \qquad \Big\langle \underline{\exists} 6, 7.\ 6 = 7 \to 3,\ \langle \forall X_1.\ X_1 \to X_1 \preceq 6, \quad \mathrm{Char} \preceq 7 \rangle \\ \quad \Big\rangle \\ \Big\rangle \quad \langle .5 \to 2/4 \rangle \end{array}$$

The process for solving this equation, first by applying the rules in Figure 6.3.1 and then using the rules in Figure 6.2.1, proceeds as follows:

$$\frac{\begin{array}{l}\left\langle \exists\!\!=\!1.\ 1 \preceq 0,\right.\\ \quad \left\langle \exists\!\!=\!2,3.\ 1 = (2,3),\right.\\ \qquad \left\langle \exists\!\!=\!4,5.\ 4 = 5 \to 2,\ \langle \forall X_0.\ X_0 \to X_0 \preceq 4,\quad \mathrm{Bool} \preceq 5\rangle\right.,\\ \qquad \left\langle \exists\!\!=\!6,7.\ 6 = 7 \to 3,\ \langle \forall X_1.\ X_1 \to X_1 \preceq 6,\quad \mathrm{Char} \preceq 7\rangle\right.\\ \quad \rangle\\ \rangle\end{array}}{\langle .5 \to 2/4, 7 \to 3/6, (2,3)/1, \mathrm{Bool}/5, \mathrm{Char}/7, \mathrm{Bool}/X_0, \mathrm{Char}/X_1, \mathrm{Bool}/2, \mathrm{Char}/3\rangle}\ \forall\text{-Elim}_{\_}$$

$$\frac{\begin{array}{l}\left\langle \exists\!\!=\!1.\ 1 \preceq 0,\right.\\ \quad \left\langle \exists\!\!=\!2,3.\ 1 = (2,3),\right.\\ \qquad \left\langle \exists\!\!=\!4,5,X_0.\ 4 = 5 \to 2,\ \langle\ X_0 \to X_0 \preceq 4,\quad \mathrm{Bool} \preceq 5\rangle\right.,\\ \qquad \left\langle \exists\!\!=\!6,7.\ 6 = 7 \to 3,\ \langle \forall X_1.\ X_1 \to X_1 \preceq 6,\quad \mathrm{Char} \preceq 7\rangle\right.\\ \quad \rangle\\ \rangle\end{array}}{\langle .5 \to 2/4, 7 \to 3/6, (2,3)/1, \mathrm{Bool}/5, \mathrm{Char}/7, \mathrm{Bool}/X_0, \mathrm{Char}/X_1, \mathrm{Bool}/2, \mathrm{Char}/3\rangle}\ \forall\text{-Elim}_{\_}$$

$$\frac{\begin{array}{l}\left\langle \exists\!\!=\!1.\ 1 \preceq 0,\right.\\ \quad \left\langle \exists\!\!=\!2,3.\ 1 = (2,3),\right.\\ \qquad \left\langle \exists\!\!=\!4,5,X_0.\ 4 = 5 \to 2,\ \langle\ X_0 \to X_0 \preceq 4,\quad \mathrm{Bool} \preceq 5\rangle\right.,\\ \qquad \left\langle \exists\!\!=\!6,7,X_1.\ 6 = 7 \to 3,\ \langle X_1 \to X_1 \preceq 6,\quad \mathrm{Char} \preceq 7\rangle\right.\\ \quad \rangle\\ \rangle\end{array}}{\langle .5 \to 2/4, 7 \to 3/6, (2,3)/1, \mathrm{Bool}/5, \mathrm{Char}/7, \mathrm{Bool}/X_0, \mathrm{Char}/X_1, \mathrm{Bool}/2, \mathrm{Char}/3\rangle}\ \exists\!\!=\!\text{-Elim}_{=}$$

$$\Big\langle \underline{\exists} 1.\ 1 \preceq 0,$$
$$\quad \Big\langle \underline{\exists} 2, 3.\ 1 = (2, 3),$$
$$\quad\quad \Big\langle \underline{\exists} 5, X_0.\ X_0 \longrightarrow X_0 \preceq 5 \longrightarrow 2, \quad \mathrm{Bool} \preceq 5 \Big\rangle\,,$$
$$\quad\quad \Big\langle \underline{\exists} 6, 7, X_1.\ 6 = 7 \longrightarrow 3,\ \langle\ X_1 \longrightarrow X_1 \preceq 6, \quad \mathrm{Char} \preceq 7 \rangle$$
$$\quad \Big\rangle$$
$$\Big\rangle$$

$$\langle .7 \longrightarrow 3/6, (2, 3)/1, \mathrm{Bool}/5, \mathrm{Char}/7, \mathrm{Bool}/X_0, \mathrm{Char}/X_1, \mathrm{Bool}/2, \mathrm{Char}/3 \rangle$$

$\underline{\exists}$-Elim$_{=}$

$$\Big\langle \underline{\exists} 1.\ 1 \preceq 0,$$
$$\quad \Big\langle \underline{\exists} 2, 3.\ 1 = (2, 3),$$
$$\quad\quad \Big\langle \underline{\exists} 5, X_0.\ X_0 \longrightarrow X_0 \preceq 5 \longrightarrow 2, \quad \mathrm{Bool} \preceq 5 \Big\rangle\,,$$
$$\quad\quad \Big\langle \underline{\exists} 7, X_1.\ X_1 \longrightarrow X_1 \preceq 7 \longrightarrow 3, \quad \mathrm{Char} \preceq 7 \Big\rangle$$
$$\quad \Big\rangle$$
$$\Big\rangle$$

$$\langle .(2, 3)/1, \mathrm{Bool}/5, \mathrm{Char}/7, \mathrm{Bool}/X_0, \mathrm{Char}/X_1, \mathrm{Bool}/2, \mathrm{Char}/3 \rangle$$

$\underline{\exists}$-Elim$_{=}$

$$\Big\langle \underline{\exists} 2, 3.\ (2, 3) \preceq 0,$$
$$\quad \Big\langle \underline{\exists} 5, X_0.\ X_0 \longrightarrow X_0 \preceq 5 \longrightarrow 2, \quad \mathrm{Bool} \preceq 5 \Big\rangle\,,$$
$$\quad \Big\langle \underline{\exists} 7, X_1.\ X_1 \longrightarrow X_1 \preceq 7 \longrightarrow 3, \quad \mathrm{Char} \preceq 7 \Big\rangle$$
$$\Big\rangle$$

$$\langle .\mathrm{Bool}/5, \mathrm{Char}/7, \mathrm{Bool}/X_0, \mathrm{Char}/X_1, \mathrm{Bool}/2, \mathrm{Char}/3 \rangle$$

$\underline{\exists}$-Elim$_{\succeq}$

$$\Big\langle \underline{\exists} 2, 3.\ (2, 3) \preceq 0,$$
$$\quad \Big\langle \underline{\exists} X_0.\ X_0 \longrightarrow X_0 \preceq \mathrm{Bool} \longrightarrow 2 \Big\rangle\,,$$
$$\quad \Big\langle \underline{\exists} 7, X_1.\ X_1 \longrightarrow X_1 \preceq 7 \longrightarrow 3, \quad \mathrm{Char} \preceq 7 \Big\rangle$$
$$\Big\rangle$$

$$\langle .\mathrm{Char}/7, \mathrm{Bool}/X_0, \mathrm{Char}/X_1, \mathrm{Bool}/2, \mathrm{Char}/3 \rangle$$

$\underline{\exists}$-Elim$_{\succeq}$

$$\Big\langle \underline{\exists} 2, 3,\ (2,3) \preceq 0,$$

$$\Big\langle \underline{\exists} X_0.\ (X_0 \rightarrow X_0 \preceq \text{Bool} \rightarrow 2),\quad \underline{\exists} X_1.\ (X_1 \rightarrow X_1 \preceq \text{Char} \rightarrow 3)\Big\rangle$$

$$\Big\rangle$$

$$\langle .\text{Bool}/X_0, \text{Char}/X_1, \text{Bool}/2, \text{Char}/3\rangle$$

$$\overline{}\ \text{match}_{\preceq}$$

$$\Big\langle \underline{\exists} 2, 3.\ (2,3) \preceq 0,$$

$$\Big\langle \underline{\exists} X_0.\ (\text{Bool} \preceq X_0, X_0 \preceq 2),\quad \underline{\exists} X_1.\ (X_1 \rightarrow X_1 \preceq \text{Char} \rightarrow 3)\Big\rangle$$

$$\Big\rangle$$

$$\langle .\text{Bool}/X_0, \text{Char}/X_1, \text{Bool}/2, \text{Char}/3\rangle$$

$$\overline{}\ \underline{\exists}\text{-Elim}_{\succeq}$$

$$\Big\langle \underline{\exists} 2, 3.\ (2,3) \preceq 0,$$

$$\Big\langle \text{Bool} \preceq \text{Bool}, \text{Bool} \preceq 2),\quad \underline{\exists} X_1.\ (X_1 \rightarrow X_1 \preceq \text{Char} \rightarrow 3)\Big\rangle$$

$$\Big\rangle$$

$$\langle .\text{Char}/X_1, \text{Bool}/2, \text{Char}/3\rangle$$

$$\overline{}\ \text{Elim}_{\preceq}$$

$$\Big\langle \underline{\exists} 2, 3.\ (2,3) \preceq 0,$$

$$\Big\langle (\text{Bool} \preceq 2),\quad \underline{\exists} X_1.\ (X_1 \rightarrow X_1 \preceq \text{Char} \rightarrow 3)\Big\rangle$$

$$\Big\rangle \quad \langle .\text{Char}/X_1, \text{Bool}/2, \text{Char}/3\rangle$$

$$\overline{}\ \text{match}_{\preceq}$$

$$\Big\langle \underline{\exists} 2, 3.\ (2,3) \preceq 0,$$

$$\Big\langle (\text{Bool} \preceq 2),\quad \underline{\exists} X_1.\ (\text{Char} \preceq X_1, X_1 \preceq 3)\Big\rangle$$

$$\Big\rangle \quad \langle .\text{Char}/X_1, \text{Bool}/2, \text{Char}/3\rangle$$

$$\overline{}\ \underline{\exists}\text{-Elim}_{\succeq}$$

$$\Big\langle \underline{\exists} 2, 3,\ (2,3) \preceq 0,$$

$$\langle (\text{Bool} \preceq 2),\ (\text{Char} \preceq \text{Char}, \text{Char} \preceq 3)\rangle$$

$$\Big\rangle \quad \langle .\text{Bool}/2, \text{Char}/3\rangle$$

$$\overline{}\ \text{Elim}_{\preceq}$$

$$\dfrac{\Big\langle \underline{\exists} 2, 3.\ (2,3) \preceq 0, (\text{Bool} \preceq 2),\ (\text{Char} \preceq 3)\Big\rangle \quad \langle .\text{Bool}/2, \text{Char}/3\rangle}{\dfrac{\Big\langle \underline{\exists} 3,\ (\text{Bool}, 3) \preceq 0,\ (\text{Char} \preceq 3)\Big\rangle \quad \langle .\text{Char}/3\rangle}{\dfrac{\langle (\text{Bool}, \text{Char}) \preceq 0\rangle \quad \langle\rangle}{}}\ \underline{\exists}\text{-Elim}_{\succeq}}\ \underline{\exists}\text{-Elim}_{\succeq}$$

We get the list of substitutions out which is:

$$[5 \to 2/4, 7 \to 3/6, (2,3)/1, \text{Bool}/5, \text{Char}/7, \text{Bool}/X_0, \text{Char}/X_1, \text{Bool}/2, \text{Char}/3, (\text{Bool}, \text{Char})/0]$$

Therefore, the final type of the term (id True, id 'A') is (Bool, Char).

6.6 Summary

In this chapter, we introduced a type inference algorithm that produces substitutions which are guaranteed to be valid under the verification rules introduced in the previous chapter. Furthermore, we proved that the inference system we introduced is at least as powerful as CaMPL's inference system, ensuring that any type derivable in CaMPL's system remains derivable within ours.

Chapter 7

Conclusion

This thesis provides a novel algorithm for type inference for higher rank types, where universal quantifiers may appear nested within function arguments or return types. The proposed algorithm separates the inference process into two steps: first, it collects type equations for a given term, and then it attempts to solve these equations. Since type inference for this system is undecidable, our inference algorithm may fail to find a solution on some inputs, even when a valid solution exists. Nonetheless, the algorithm presented in this thesis is at least as powerful as existing algorithms in its ability to infer types.

This thesis presents a formal development of the type inference system, on which the algorithm is based. The thesis begins by presenting System F and modifies it to define "System S", which is compatible with programming. In designing a higher-rank type system based on System S, we observe that some terms can be assigned more than one type. To formalize this, we define a subsumption relation, $T \preceq S$, where a type T subsumes a type S if every term typable as T is also typable as S. These subsumption relations can be added to system S as coercions.

The thesis generalizes subsumptions to introduce type equations. It presents the rules for generating such type equations from a given term. It shows that if a term can be type-checked in System F with a particular type, then that type serves as a solution to the equations generated for the term. The thesis also discusses candidate solutions to type equations and formulates verification rules to determine their validity.

The type inference algorithm in this thesis is introduced in several stages. First, a small-step inference system is defined to simplify type equations, and its soundness is proven. This small-step inference

system is then used to construct a non-deterministic big-step inference system. This big-step inference is ultimately refined into a deterministic, yet necessarily partial, algorithm for type inference.
The algorithm presented in this thesis is slightly stronger than Haskell's: everything that can be type-checked in Haskell can also be type-checked by this algorithm, and some additional minor cases that Haskell cannot handle are also supported. The algorithm is simpler, as the bidirectional system has been absorbed into a uniform process of generating equations.

Bibliography

[1] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM-SIGACT Symposium on Principles of Programming Languages*, 1982.

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

[3] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shield. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1 – 82, 2007.

[4] Joe B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Log.*, 98:111–156, 1999.

[5] Jinxu Zhao, Bruno C. D. S. Oliveira, and Tom Schrijvers. A mechanical formalization of higher-ranked polymorphic type inference. *Proceedings of the ACM on Programming Languages*, 3:1 – 29, 2019.

[6] Masuka Yeasin. Linear functors and their fixed points. MSc thesis, University of Calgary, 2012.

[7] Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nord. J. Comput.*, 6:343–361, 1999.

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

[9] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. A quick look at impredicativity. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[10] The essence of ML type inference. In *Advanced Topics in Types and Programming Languages*. The MIT Press, 12 2004.

[11] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[12] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys (CSUR)*, 54(5):1–38, 2021.

[13] Simon Peyton Jones and Mark Shields. Lexically scoped type variables. Microsoft Research, January 2002.

[14] Prashant Kumar. Implementation of message passing language. MSc thesis, University of Calgary, 2018.

[15] Robin Cockett. CPSC 521: Basic programming for computable functions: BPCF (the typed lambda calculus with fixed points) – course notes. 2024.

[16] Aaron Stump, Garrin Kimmell, Hans Zantema, and Ruba El Haj Omar. A rewriting view of simple typing. *Logical Methods in Computer Science*, Volume 9, Issue 1, February 2013.

[17] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[18] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, USA, 1986.

[19] Stephen A. Edwards. The Hindley-Milner type system – lecture slides, columbia university, spring 2023. https://www.cs.columbia.edu/~sedwards/classes/2023/6998-spring-tlc/hindleymilner.pdf, 2023.

[20] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[21] Robin Cockett. CPSC 449: Notes on matching and unification (for discussion in labs!). 2021.

[22] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[23] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition edition, 2016.

[24] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, USA, 1989.

[25] J. B. Wells. Typability is undecidable for F+Eta. Technical report, Boston University, USA, 1996.

[26] Didier Rémy and Julien Cretin. From amber to coercion constraints. In Martin Abadi, Philippa Gardner, Andrew D. Gordon, and Radu Mardare, editors, *Essays for the Luca Cardelli Fest*, number MSR-TR-2014-104 in TechReport. Microsoft Research, September 2014.

[27] Simon Peyton Jones. Simplify subsumption discussion. https://github.com/ghc-proposals/ghc-proposals/pull/287, 2020. GHC Proposals, Pull Request #287.

[28] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory (2nd ed.)*. Cambridge University Press, USA, 2000.