Decidability and Algorithmic Analysis of Dependent Object Types (DOT)

by

Zhong Sheng (Jason) Hu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Abstract

Dependent Object Types, or *DOT*, is a family of calculi developed to study the Scala programming language. These calculi have *path dependent types* as a feature, and potentially intersection types, union types and recursive types. So far, the study of *DOT* calculi mostly focuses on the soundness proof, which does not directly contribute to development of compilers. This thesis presents a detailed investigation of decidability and algorithmic properties of the family of *DOT* calculi.

In decidability analysis, the undecidability of subtyping of several calculi is formally established, including the $D_{<:}$ and $D_{\wedge}$ calculi. Prior to this investigation, the undecidability of subtyping of all *DOT* calculi including $D_{<:}$ was open. Decidability analysis puts emphasis on a particular form of subtyping rules, called *normal form*. It turns out that a normal form definition is not only as expressive, but also more suggestive than the original definition. A conceptual device, called *small-step analysis*, is introduced to assist converting a usual definition of subtyping to its normal form definition. Moreover, decidability analysis gives direct contributions to the algorithmic analysis, by revealing two decidable fragments of $D_{<:}$ in declarative form, called the *kernels*. Decidability analysis also suggests a novel subtyping algorithm framework, *stare-at subtyping*. Stare-at subtyping and an existing algorithm are shown to be sound and complete w.r.t. their corresponding kernels.

In algorithmic analysis, stare-at subtyping is extended to other calculi, with more features than $D_{<:}$, including $D_{\wedge}$, $\mu DART$ and $jDOT$. In $\mu DART$ and $jDOT$, bi-directional type assignment algorithms are developed. The algorithms developed in this thesis are all shown to be sound with respect to their target calculi and terminating.

During the development of the algorithms, analysis shows a number of ways in which the Wadlerfest *DOT* calculus does not directly correspond to the Scala language, while substantially increases the difficulties of algorithmic design. $jDOT$, therefore, is developed as an alternative formalization of Scala.

Acknowledgements

In my opinion, a thesis is not only a piece of technical work but also a cumulative result of social events preceding it. At the time of concluding this thesis, it is the best moment to look back in my life and appreciate everyone and everything that make this thesis possible.

I would like to thank my supervisor Professor Ondřej Lhoták for knowledge I have learned and his continuous support since my application for the Master's program and during my research.

I feel very grateful for the lab environment. My understanding of DOT had been significantly improved thanks to conversations with Marianna Rapoport, Ifaz Kabir and surely my supervisor. The thesis inherits work from Abel Nieto's project. The discussions with him benefited greatly my work on undecidability proof of $D_{<:}$, for which I am equally grateful.

I would like to thank my readers Professors Prabhakar Ragde and Brad Lushman for their careful review of my thesis.

I would also like to thank Professors Yong Wang, Bo Hu, Xiaofeng Wu and Hui Feng for their support for my Master's application after years of my graduation from Bachelor's, and Professors Arie Gurfinkel and Richard Trefler for their support for my PhD application.

Last, I must thank my parents and my grandparents for giving me abundant freedom since my childhood. I am grateful to Pierre Grondin. It would probably have been much more difficult to settle in Canada without his help.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The thesis presents the results of an investigation of decidability and algorithmic analysis of the family of Dependent Object Types (DOT) calculi. This thesis improves the understanding of these calculi by analyzing the decidability of some of these calculi, and proposes new algorithmic typing methodologies to type check programs in the members of this family of calculi.

1.1 Motivation

Scala is a programming language which combines object orientation and functional programming with dependent types. The formalizations of Scala had been a long lasting problem. Dependent Object Types, or *DOT* [Rompf and Amin, 2016, Amin et al., 2016, Rapoport et al., 2017], has been generally considered as a representative model of Scala. Since *DOT*'s first appearance, the focus has been on the actual formulation and the soundness proof. Since 2016, there have been various successful formalizations with their soundness proofs [Rompf and Amin, 2016, Amin et al., 2016, Rapoport et al., 2017]. These proofs are established by progress-and-preservation [Wright and Felleisen, 1992].

Meanwhile, the Scala language was released in 2004, and Dotty, which is the compiler for the third generation of Scala, is under active development and has been bootstrapped. Its long history implies the language is implemented without a thorough formal analysis. In paricular, the compiler has the following three problems.

1. It is not formally specified.

2. It is not guaranteed to be correctly implemented.

3. It is not guaranteed to terminate.

Hence, it is hard for new comers to quickly understand the Dotty compiler, as it is developed based on some informal semantics.

What is more, the *DOT* calculi are formulated declaratively. A declarative definition of a calculus favors the mathematical nature of the language, instead of, e.g. how to type check programs in this language. This is unfortunate, because that means the definition gives no direct hints on how to implement a type checker. Though the calculus has been proven sound, strictly speaking, the same conclusion cannot be directly drawn for the implementation. The lack of general understanding of the calculus allows interpretations of language features to vary largely among compiler writers and in time, and makes the compiler even harder to understand.

This situation raises two research questions targeted in this thesis.

Question 1. *Is (sub)typing of DOT decidable?*

Question 2. *How should DOT programs be type checked?*

Let me elaborate on these two questions.

1.1.1 Decidability

The first question asks what we can expect the compiler to solve. If the type system of a language is decidable, then we can expect a complete algorithm to be implemented in the compiler to answer *all* problems on terms and types. For example, the type system in Standard ML [Milner et al., 1997] is based on Hindley-Milner type system [Milner, 1978, Hindley, 1969], and its (correctly implemented) compiler admits a program if and only if a type can be assigned to it. Fun [Cardelli and Wegner, 1985] is a programming language which can always decide whether one type is a subtype of another.

On the other hand, if a type system is undecidable, then it is impossible to implement a complete type checking algorithm. Whether the type system is decidable, therefore, is a fundamental property of the language. This problem turns out to be far less clear when it comes to *DOT*.

To actually answer this question, a rigorous *decidability analysis* is performed. Decidability analysis studies *all* interactions between the language features and their associated rules. This analysis is purely theoretical and challenging; therefore the problem serves theorists' interest. On the other side, if the language is shown undecidable, then the focus of the implementation should not be a complete type checking algorithm like the ones of Standard ML and Fun. This gives realistic significance to the compiler writers.

1.1.2 Type checking

The second question is related to the first one but has its own freedom. The decidability analysis answers whether a complete (sub)typing algorithm should be expected, while this question asks for direct instructions on how to implement a type checker and a subtyper. Though a compiler can readily be implemented without a rigorous analysis, just like the existing Scala and Dotty compilers, it is hard to quantify what the compilers should do and what the compilers actually do. Therefore the solution is to formally specify a set of inference rules which must respect the declarative definition of the language and terminate. This resolves all three problems of the current implementations posed above.

Nieto [2017] had an initiative to resolve this situation. His work adapted some methods from a well-known calculus $F_{<:}$ to express a partial typing algorithm for $D_{<:}$, the simplest member of the DOT family. $D_{<:}$ has only one feature of DOT, called the path dependent types. This thesis continues this direction and pushes it further by providing subtyping and typing algorithms for calculi with more features than $D_{<:}$.

1.2 Methodologies

As I will show in Section 2.5, there are three major features in the DOT calculus: path dependent types, intersection types and recursive types. In order to obtain substantial understanding of this calculus, the best way to approach it is to to consider the features each at a time. Since $D_{<:}$ has only path dependent types, I first focus on $D_{<:}$ to ensure the decidability and algorithmic analysis are done extensively and properly for the simplest problem. Only after this, I will add other features to $D_{<:}$, one by one, to study their interactions with path dependent types. At the very end, the techniques learned in all previous work are merged and adapted to obtain a final solution.

1.3 Organization

This thesis has strong linear dependency between chapters, so later chapters may appear hard to digest if earlier chapters are not read. The chapters are organized in the order of this linear dependency.

Chapter 2 defines the conventions used consistently throughout this thesis and describes the basic technical preliminaries.

Chapter 3 introduces the calculus $D_{<:}$, the simplest calculus in the *DOT* family. This chapter focuses on decidability analysis of several calculi, including $D_{<:}$.

Chapter 4 introduces algorithmic typing of $D_{<:}$, including Nieto's work and a novel improvement that is inspired by the decidability analysis in Chapter 3.

Chapter 5 starts to investigate the interactions between features. This chapter defines $D_{\wedge}$ (pronounced *Dee intersect*), which is a calculus extending $D_{<:}$ with intersection types. This chapter performs decidability analysis of $D_{\wedge}$, and extends the result in Chapter 4 with intersection types.

Chapter 6 investigates the interactions in a different direction. This chapter defines $\mu DART$, which is a calculus extending $D_{<:}$ with recursive types. Recursive types appear to put significant difficulties in decidability analysis, and therefore this chapter only considers algorithmic (sub)typing operationally.

Chapter 7 revises the study in Chapter 5 and Chapter 6, and reconsiders the definition of a version of *DOT*. In this chapter, *jDOT* is proposed, which is a more controlled version of *DOT*. *jDOT* has path dependent types, intersection types, and recursive types. This chapter describes algorithmic (sub)typing of this calculus, as well as examples of programs in this calculus.

Chapter 8 discusses some future work.

Chapter 9 concludes this thesis.

1.4 Contributions

In this thesis, I made the following contributions.

1. I established complete and formal proofs of undecidability of subtyping of the following calculi: $F_{<:}$, $F_{<:}^{-}$, $F_{<:}$ with bottom ($\bot$), $F_{<:>}^{-}$, $D_{<<:}$, $D_{<:}$ without transitivity, $D_{<:}$ and $D_{\wedge}$.

2. I established a complete and formal proof of undecidability of type assignment in $D_{<:}$.

3. I give strong reasons to focus study on a particular form of subtyping rules, called the *normal form*, and propose a conceptual device, *small-step analysis*, to convert non-normal form subtyping definitions to normal form.

4. I propose a novel subtyping decision framework, called *stare-at subtyping*, which substantially improves decidable fragments over previous work, while remaining easy to understand. Stare-at subtyping has been developed for the following calculi: $D_{<:}$, $D_{\wedge}$, $\mu DART$ and $jDOT$.

5. I discover decidable fragments of $D_{<:}$ and $D_{\wedge}$ in declarative form, called kernel $D_{<:}$, strong kernel $D_{<:}$, and strong kernel $D_{\wedge}$. I establish the proofs showing an existing subtyping algorithm, step subtyping, is sound and complete for kernel $D_{<:}$, and stare-at subtyping is sound and complete for strong kernel $D_{<:}$ and strong kernel $D_{\wedge}$.

6. I analyze Wadlerfest DOT and reveal a number of problems in its definition. From this, I propose another alternative of DOT, called $jDOT$, which is simpler than DOT, but makes better use of the language constructs, and becomes more expressive than DOT in its encoding. I give an example to demonstrate this claim.

7. I develop a bi-directional type assignment algorithm for $jDOT$.

There are many calculi involved in this thesis. To help better understand the thesis, I have summarized the result of decidability analysis in Table 1.1. In the table, the first group denotes four important features that make a difference in decidability analysis. Full function subtyping denotes a form of subtyping which allows subtyping comparison between parameter types. This will become clear when the concrete calculi are discussed. The Name column gives the names used in this thesis for calculi with these particular combinations of features. The remaining two columns show whether each calculus is decidable and if so, what is the subtyping algorithm. The decidability of $\mu DART$, DOT and $jDOT$ are unknown because they do not have either decidability or undecidability proof, though I conjecture that that their subtyping relations are all undecidable. This conjecture justifies the lack of completeness proofs of stare-at subtyping for these calculi.

Figure 1.1 summarizes the expressive power of some calculi with declarative form discussed in this thesis. The expressive power of the source calculus of an arrow is stronger than the one of the target.

features				name	decidable?	algorithm
bad bounds	intersections	μ types	full function subtyping			
yes	no	no	yes	$D_{<:}$	no	N/A
no	no	no	no	(strong) kernel $D_{<:}$	yes	step / stare-at subtyping
no	no	no	yes	$D_{<:}$ without transitivity	no	N/A
yes	yes	no	yes	$D_{\wedge}$	no	N/A
no	yes	no	no	strong kernel $D_{\wedge}$	yes	stare-at subtyping
yes	no	yes	yes	$\mu DART$	unknown	N/A
yes	yes	yes	yes	DOT	unknown	N/A
yes	yes	yes	yes	$jDOT$	unknown	N/A

Table 1.1: Decidability of subtyping with features

The technical work is done in a combination of Agda 2.5.4.2 and Coq 8.8.2 and can be found in https://gitlab.com/JasonHuZS/AlgDotCalculus. One can obtain detailed correspondences between formalization and theorems in this thesis following the link. In the formalization, decidability analysis is done in Agda while algorithmic analysis is done in Coq.

$$\begin{tikzcd}
 & & \text{Kernel } D_{<:} & \\
 & & \text{Strong kernel } D_{<:} \arrow[u] & \\
\text{Strong kernel } D_{\wedge} \arrow[urr] & & D_{<:} \arrow[u] & \\
 & D_{\wedge} \arrow[ul] \arrow[ur] & & \mu DART \arrow[ul] \\
 & & jDOT \arrow[ul] \arrow[ur] & \\
 & & DOT \arrow[u] &
\end{tikzcd}$$

Figure 1.1: Expressive power of calculi

Chapter 2

Background

In this chapter, I will introduce the technical background and the existing work. I will begin with an old calculus, $F_{<:}$, and present how its undecidability was determined. I will then describe the Scala language and its theoretical model, DOT.

2.1 Conventions

Before beginning the technical discussion, I list several conventions used throughout this thesis.

Convention 1. *For conclusions, I use the following conventions.*

1. *I use **Theorem** and **Lemma** to denote that the conclusions have been formally verified by proof assistants, or had external references.*
2. *I use **Corollary** and **Proposition** to denote that the conclusions have informal proofs. These proofs exist because they do not contribute to the eventual conclusions.*

Convention 2. *Conventions for symbols are listed in Table 2.1.*

Convention 3. *Throughout this thesis, I will use Barendregt's variable convention [Barendregt, 1984] which draws equivalences among types/terms up to α conversion, and assume α-conversion occurs automatically whenever necessary. This implies that I need to assume an infinite supply of names.*

symbols	meaning
Γ, Δ	typing environments / contexts
S, T, U	types
x, y, z, w	term variables
X, Y, Z	type variables
t, u	terms
v	values
V	sets of variables

Table 2.1: Symbol conventions

Additionally, to emphasize some free occurrences of a variable, I sometimes write the free variable in the subscript. For example, T_x means that x may occur free in T. If T_y appears later, it means all corresponding x's in T are replaced by y.

Convention 4. *I use semicolons (;) to denote context concatenation, instead of commas (,).*

Convention 5. *I use $\Gamma \vdash t : T$ to denote a typing judgment, meaning "in the context Γ, the term t has type T". I use $\Gamma \vdash S <: U$ to denote a subtyping judgment, meaning "in the context Γ, S is a subtype of U".*

When a concrete calculus is discussed, the name of the calculus appears as subscript of the turnstile, to avoid ambiguity, e.g. $\Gamma \vdash_{F_{<:}} S <: U$ is a subtyping judgment in the system $F_{<:}$ (to be discussed in the next section). When a subscript is missing, it is meant to be a general discussion and applies to all calculi within the discussion in this thesis.

Definition 2.1. *A free variable of a term or a type is a variable that is not bound to any binder.*

$fv(\cdot)$ is an overloaded notation denoting the set of free variables (of a term or a type).

Convention 6. *Following the standard mathematical notations, I will use $\cup$ for union, $\cap$ for intersection, $\setminus$ for set difference. $\emptyset$ denotes the empty set. To simplify the text, if a single variable participates in these operations, it's regarded as a singleton set, e.g. $x \cup y = \{x, y\}$.*

Definition 2.2. *$dom(\Gamma)$ is the domain of the context Γ, which is the set of variables bound to something in the context.*

Definition 2.3. *For some object τ, it is closed w.r.t. a context Γ, if $fv(\tau) \subseteq dom(\Gamma)$.*

Definition 2.4. *Well-formedness is a predicate on contexts defined inductively as follows.*

1. *The empty context $\bullet$ is well-formed.*
2. *If Γ is well-formed and τ is closed w.r.t. Γ, and $x \notin dom(\Gamma)$, then $\Gamma; x : \tau$ is well-formed.*

Convention 7. *In the formal proofs of the conclusions presented in this thesis, well-formedness may or may not be an actual condition. To make the discussion more concise, I intend to not mention this condition, and it is safe to simply assume well-formedness everywhere. If a conclusion can be proved without well-formness, it is still true with an extra condition.*

However, there are conclusions that directly involve reasoning about well-formedness. For those conclusions, I will make the well-formedness condition explicit.

2.2 Full $F_{<:}$ and Kernel $F_{<:}$

In this section, I will define a few variants of $F_{<:}$ [Curien and Ghelli, 1990, Cardelli et al., 1994].

Definition 2.5. *Full $F_{<:}$ (or, in short, $F_{<:}$) is the calculus defined in Figure 2.1.*

This system was first introduced in Cardelli and Wegner [1985], as an attempt to combine polymorphism and subtyping. Prior to $F_{<:}$, polymorphism was understood to be like in Haskell or ML, where a universal type must work for *every* type. $F_{<:}$ allows interactions between polymorphism and subtyping, so that a universal type can be refined by additional restrictions on what types it needs to work with. This substantially increases the expressive power.

In the rules, Refl and Trans say that the subtyping relation is reflexive and transitive, two properties that are typically desired of subtyping. Top says that $\top$ is a supertype of all types. The Tvar rule looks up a type variable the context and asserts that it is a subtype of its bound in the context. Fun defines the subtyping relation between term level functions. Notice that the subtyping relation between parameter types of functions is flipped because they are in contravariant position.

All deserves more attention. First, it defines the subtyping relation between two universal types. Universal types implement polymorphism. A polymorphic type is like

X, Y, Z	**Type variable**
$S, T, U ::=$	**Type**
$\top$	top type
X	type variable
$S \to U$	function
$\forall X <: S.U_X$	universal type

Subtyping

$$\frac{}{\Gamma \vdash_{F_{<:}} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{F_{<:}} T <: T}\ \textsc{Refl}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash_{F_{<:}} X <: T}\ \textsc{Tvar}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{F_{<:}} S' <: S \\ \Gamma \vdash_{F_{<:}} U <: U'\end{array}}{\Gamma \vdash_{F_{<:}} S \to U <: S' \to U'}\ \textsc{Fun}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{F_{<:}} S' <: S \\ \Gamma; X <: S' \vdash_{F_{<:}} U <: U'\end{array}}{\Gamma \vdash_{F_{<:}} \forall X <: S.U <: \forall X <: S'.U'}\ \textsc{All}$$

$$\frac{\Gamma \vdash_{F_{<:}} S <: T \quad \Gamma \vdash_{F_{<:}} T <: U}{\Gamma \vdash_{F_{<:}} S <: U}\ \textsc{Trans}$$

Figure 2.1: Definition of subtyping in $F_{<:}$ [Pierce, 2002, Figure 26-2]

a type level function, and can be materialized to another type if the type variable is instantiated. In $F_{<:}$, universal types are more interesting than those in languages without subtyping because type variables can be quantified by type upper bounds, hence *bounded quantification*. An upper bound requires that the type variable it quantifies can only be instantiated by its subtype. Second, the subtyping judgment between U and U', the return types, depends on the subtyping judgment between S and S', the parameter types, as the typing context needs to be extended by $X <: S'$. This happens to be enough to turn the subtyping problem undecidable and it will be discussed in the next section.

Term typing of $F_{<:}$ will not be relevant in this thesis. Moreover, in $F_{<:}$, the typing relation depends on subtyping in the subsumption rule, but the subtyping relation never depends on typing. The following theorems ensure that term variable bindings in the typing context do not affect the subtyping relation.

Theorem 2.1. *(term variable weakening [Cardelli et al., 1994])*

If $\Gamma_1; \Gamma_2 \vdash_{F_{<:}} S <: U$*, then* $\Gamma_1; x : T; \Gamma_2 \vdash_{F_{<:}} S <: U$.

Theorem 2.2. *(term variable strengthening [Cardelli et al., 1994])*

If $\Gamma_1; x : T; \Gamma_2 \vdash_{F_{<:}} S <: U$*, then* $\Gamma_1; \Gamma_2 \vdash_{F_{<:}} S <: U$.

A very straightforward extension of $F_{<:}$ is to add a bottom type to it.

$$\frac{}{\Gamma \vdash_{F_{<:}} T <: \top}\ \textsc{Top} \qquad \boxed{\frac{}{\Gamma \vdash_{F_{<:}} X <: X}\ \textsc{VarRefl}} \qquad \boxed{\frac{X <: T \in \Gamma \quad \Gamma \vdash_{F_{<:}} T <: U}{\Gamma \vdash_{F_{<:}} X <: U}\ \textsc{Tvar'}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{F_{<:}} S' <: S \\ \Gamma \vdash_{F_{<:}} U <: U'\end{array}}{\Gamma \vdash_{F_{<:}} S \to U <: S' \to U'}\ \textsc{Fun} \qquad \frac{\begin{array}{c}\Gamma \vdash_{F_{<:}} S' <: S \\ \Gamma; X <: S' \vdash_{F_{<:}} U <: U'\end{array}}{\Gamma \vdash_{F_{<:}} \forall X <: S.U <: \forall X <: S'.U'}\ \textsc{All}$$

Figure 2.2: Definition of $F_{<:}$ normal form

Definition 2.6. *[Pierce, 1997] $F_{<:}$ with bottom is defined by extending Figure 2.2 with an additional type $\perp$ and the following rule.*

$$\frac{}{\Gamma \vdash_{F_{<:}} \perp <: T}\ \textsc{Bot}$$

It turns out that the formulation of $F_{<:}$ can be simplified, so that the transitivity rule TRANS becomes a provable consequence, instead of an explicit rule of the system. I follow Pierce and call this equivalent form the $F_{<:}$ *normal form*.

Definition 2.7. *$F_{<:}$ normal form is defined in Figure 2.2. The difference from $F_{<:}$ (non-normal form) is shaded.*

Theorem 2.3. *[Curien and Ghelli, 1990] $F_{<:}$ subtyping is equivalent to $F_{<:}$ normal form. Namely $\Gamma \vdash_{F_{<:}} S <: U$ holds in non-normal form, iff it holds in normal form.*

This theorem is true, because VARREFL is sufficient to prove reflexivity and the second premise in the TVAR' rule is enough to prove transitivity.

One immediate advantage of considering normal form is that it becomes obvious which pairs of types can have subtyping. For a judgment $\Gamma \vdash_{F_{<:}} S <: U$, if both S and U are universal types, then through normal form, ALL clearly is the only applicable rule. If S is a universal type and U is a function type, then they cannot be subtypes of each other, because there is no applicable rule. In non-normal form, due to TRANS, none of the previous observations is immediate.

The definition of $F_{<:}$ normal form might look innocently computable, but it is in fact undecidable.

Theorem 2.4. *[Pierce, 1992] $F_{<:}$ subtyping is undecidable.*

This is quite unfortunate but motivates another variant of $F_{<:}$, called *kernel* $F_{<:}$.

Definition 2.8. *Kernel $F_{<:}$ is defined by replacing the* All *rule with the following rule.*

$$\frac{\Gamma; X <: S \vdash_{F_{<:}} U_1 <: U_2}{\Gamma \vdash_{F_{<:}} \forall X <: S.U_1 <: \forall X <: S.U_2} \text{ All'}$$

The difference between All and All' is that All' requires the parameter types of both universal types are syntactically identical, while in All, different parameter types are allowed, as long as they have subtyping relation.

This modification turns kernel $F_{<:}$ decidable and there is a decision procedure for it [Pierce, 2002, Chapter 26]. This modification hints that allowing subtyping comparison for parameter types of universal types introduces undecidability. The reason for this will be discussed in the next section.

2.3 Undecidability of Full $F_{<:}$

Pierce [1992] described a detailed chain of equivalences from $F_{<:}$ subtyping to the halting problem of two-counter machines, a well-known Turing equivalent problem. From there, we know that solving $F_{<:}$ subtyping would solve the halting problem, so we can conclude that $F_{<:}$ subtyping is undecidable according to Definition 3.2. As hinted in the previous section, the problematic rule is All.

$$\frac{\Gamma \vdash_{F_{<:}} S' <: S \quad \Gamma; X <: S' \vdash_{F_{<:}} U_X <: U'_X}{\Gamma \vdash_{F_{<:}} \forall X <: S.U_X <: \forall X <: S'.U'_X} \text{ All}$$

I highlighted the free occurrences of X's in both U and U'. Consider the type $\forall X <: S.U_X$. Intuitively, in this type, all X's in U_X are really meant to be S, because no additional information is available for any further refinement. However, the situation changes in the second premise of All rule, where $X <: S$ is effectively replaced by $X <: S'$. This constraint is tighter and more informative, which is proved by the first premise $\Gamma \vdash_{F_{<:}} S' <: S$. Now let us consider a special case where $S = \top$. Then $\Gamma \vdash_{F_{<:}} S' <: S$ is naturally true due to Top. In this case, there is no bound on what X means in U_X and that is fully controlled by another type $\forall X <: S'.U'_X$ which is unknown to U_X. This is the essence of Pierce's construction. Following this intuition, he showed that this refinement effectively becomes a substitution operation in its full strength, and renders the whole problem undecidable.

Fortunately, in this thesis, I do not need to get deep into the full chain of Pierce's construction. In Pierce [1992], the translation goes from the $F_{<:}$ normal form defined in Figure 2.2 to two-counter machines via another calculus called $F_{<:}$ deterministic, $F^d_{<:}$. I use $F^d_{<:}$ as the trust base of my undecidability proofs of other calculi.

Definition 2.9. *$F^d_{<:}$ is defined in Figure 2.3.*

$$
\begin{aligned}
T^+ &::= \top \quad | \quad \forall X_0 <: T_0^- .. X_n <: T_n^- . \neg T^- \\
T^- &::= X \quad | \quad \forall X_0 .. X_n . \neg T^+
\end{aligned}
$$

$$
\frac{}{\Gamma \vdash_{F^d_{<:}} X <: \top}\ \textsc{Top} \qquad \frac{\Gamma \vdash_{F^d_{<:}} \Gamma(X) <: \forall X_0 <: T_0^- .. X_n <: T_n^- . \neg T^-}{\Gamma \vdash_{F^d_{<:}} X <: \forall X_0 <: T_0^- .. X_n <: T_n^- . \neg T^-}\ \textsc{Var}
$$

$$
\frac{\Gamma; X_0 <: T_0^-; ..; X_n <: T_n^- \vdash_{F^d_{<:}} U^- <: S^+}{\Gamma \vdash_{F^d_{<:}} \forall X_0 .. X_n . \neg S^+ <: \forall X_0 <: T_0^- .. X_n <: T_n^- . \neg U^-}\ \textsc{AllNeg}
$$

Figure 2.3: Definition of $F^d_{<:}$ for a fixed $n \in \mathbb{N}$ [Pierce, 1992]

The calculus is parameterized by a natural number. Let us fix the natural number to be n. n eventually maps to the number of instructions in two-counter machines, so the undecidability proof must consider all natural numbers. We call T^+ *positive types* and T^- *negative types*. Positive types are either $\top$, or a universal type with precisely $n+1$ type variables (from 0 to n) upper bounded by $n+1$ negative types respectively. Negative types are either a type variable, or a universal type with $n+1$ type variables but without upper bounds. A context Γ is called *negative*, when all type variables in it bind to negative types. Contexts in $F^d_{<:}$ are all negative. A subtyping judgment in $F^d_{<:}$, $\Gamma \vdash_{F^d_{<:}} S^- <: U^+$, is a ternary predicate where Γ and S^- are always negative and U^+ is always positive. Notice that there is no subtyping relation between two negative or positive types.

One can define a type interpretation function to translate types in $F^d_{<:}$ to types in $F_{<:}$.

$$\begin{aligned} [\![\top]\!] &= \top \\ [\![\forall X_0 <: T_0^- .. X_n <: T_n^- . \neg T^-]\!] &= \forall X_0 <: [\![T_0^-]\!] .. \forall X_n <: [\![T_n^-]\!] . \forall X <: [\![T^-]\!] . X \end{aligned}$$

$$\begin{aligned} [\![X]\!] &= X \\ [\![\forall X_0 .. X_n . \neg T^+]\!] &= \forall X_0 <: \top .. \forall X_n <: \top . \forall X <: [\![T^+]\!] . X \end{aligned}$$

Types in $[\![\cdot]\!]$ are in $F^d_{<:}$ and the result types are in $F_{<:}$. For example, there are two occurrences of $\top$ but they are distinct types in different calculi. Notice that the negative universal types in $F^d_{<:}$ are translated to universal types in $F_{<:}$ with upper bounds as $\top$. This ensures that the upper bounds of the positive universal types can always be added to the context in the AllNeg rule and corresponds to the intuition described above. This translation witnesses the undecidability of $F_{<:}$.

Theorem 2.5. *(restate Theorem 2.4) $F_{<:}$ subtyping is undecidable.*

Proof. Since the contexts in $F^d_{<:}$ are negative, one can define an interpretation function of contexts from $F^d_{<:}$ to $F_{<:}$ as follows.

$$\begin{aligned} \langle\!\langle \bullet \rangle\!\rangle &= \bullet \\ \langle\!\langle \Gamma ; X <: T^- \rangle\!\rangle &= \langle\!\langle \Gamma \rangle\!\rangle ; X <: [\![T^-]\!] \end{aligned}$$

Then the conclusion is induced by the following equivalence.

$$\Gamma \vdash_{F^d_{<:}} S <: U \text{ iff } \langle\!\langle \Gamma \rangle\!\rangle \vdash_{F_{<:}} [\![S]\!] <: [\![U]\!]$$

for all n. □

The previous theorem requires to show that $F^d_{<:}$ is undecidable.

Theorem 2.6. *[Pierce, 1992] $F^d_{<:}$ is undecidable.*

Proof. The proof involves showing that an even smaller restriction of $F^d_{<:}$ is equivalent to the halting problem of two-counter machines, where n in $F^d_{<:}$ corresponds to the number of instructions in two-counter machines. The undecidability is induced by the fact that the proof holds for every n, so that the two-counter machines have unbounded computational power, and hence their halting problem is equivalent to the halting problem of Turing machines. □

2.4 Bi-directional Type Assignment

Bi-directional type assignment was originally developed in Pierce and Turner [2000] to address the type inference problem in $F_{<:}$ with $\bot$ (Definition 2.6). A bi-directional type assignment algorithm is composed of two modes: the checking mode and the synthesis mode. A usual type assignment algorithm mixes both type checking and type synthesis and therefore has no such distinction. In this thesis, I developed bi-directional type assignment algorithms for $\mu DART$ and $jDOT$ and they are discussed in Chapters 6 and 7.

The checking mode is represented by the following judgment:

$$\Gamma \vdash t \overleftarrow{:} T$$

This judgment states that "a term t is checked to have type T in the context Γ". As an algorithm, all three of Γ, t and T are inputs, and the algorithm outputs yes if and only if the relation is admissible. In general, in this thesis, I will use inference rules to represent algorithms. An actual algorithm can be extracted from a set of inference rules by viewing the premises of each inference rule as nested function calls (potentially recursive calls). Inputs and outputs will be explicitly specified.

The synthesis mode is represented by the following judgment:

$$\Gamma \vdash t \overrightarrow{:} T$$

This judgment states that "a type T can be synthesized from the term t in the context Γ". The arrows above the colons suggest the mode of the type assignment algorithm. As an algorithm, the synthesis judgment has Γ and t as inputs and T as the output.

2.5 Scala and DOT Calculi

Scala is a programming language with a large set of features. It supports both paradigms of object orientation and functional programming. Over time, new programming styles unique in the language have been developed which utilize the interactions of various language constructs. The Dotty compiler has even more features. Out of these features, many are related to types. On one hand, these features grant the users excessive freedom to express their logic. On the other hand, the interactions between features are in an unknown realm to the maintainers of the compiler and the theorists who try to understand and study the language. The *DOT* family of calculi extracts an essential portion of the language so that it can be formally reasoned about. In particular, there are three important features we would like to capture in a core calculus.

1. path dependent types (or paths in short),
2. intersection types, and
3. recursive types (or μ types).

Among them, recursive types are just object types, in which each member has access to other members defined in the same object. This behavior is common in object oriented languages like Java and C#. This phrase is overloaded and not to be confused with recursive types as defined in Amadio and Cardelli [1993]. The other two features are explained via some examples.

2.5.1 Path Dependent Types

In Figure 2.4, I defined a pair of simple interfaces to model retail banks and their accounts in Dotty. A Scala `trait` is just like a Java or C# interface, except that it allows function bodies. `Account` defines what a bank account looks like. Every `Account` has an id and balance. It knows which bank it belongs to, as indicated by

```
type B <: Bank
```

This line defines that every `Account` has a type member `B` and it needs to be a subtype of `Bank` which is to be defined later. Visually, this is just bounded quantification, but we will see that it is more general than that. `deposit` and `withdraw` are two basic operations of a bank account.

`Bank` is another `trait` defined to model a bank. On line 24, `Bank` defines a type member `A` to represent the type of `Accounts` in it. It additionally requires the bank of the account refers back to the singleton type of the `trait Bank`. A singleton type in Scala is written as `x.type`, and `x` is the only object of this type. One can look up an account based on her id (`lookupAccount`), and create one (`createAccount`). `transfer` is the function to transfer money from one account to another potentially in a different bank. In this case, we do want to make sure that the source account is indeed from the current bank, and the destination account is indeed from some known bank. Let me repeat its signature.

```scala
import scala.collection.mutable

class NotEnoughBalance(amount : Long) extends RuntimeException

trait Account {
  type B <: Bank
  def id : Int
  protected var _balance : Long
  def balance : Long = _balance
  def deposit(amount : Long) : Long = {
    _balance += amount
    balance
  }
  def withdraw(amount : Long) : Long = {
    if (amount > balance) {
      throw new NotEnoughBalance(amount)
    }
    _balance -= amount
    balance
  }
}

trait Bank { self =>
  type A <: Account { type B = self.type }
  def lookupAccount(id : Int) : A
  def createAccount(initialBalance : Long = 0) : A
  def transfer(amount : Long, from : self.A,
    toBank : Bank, to : toBank.A) : Unit = {
    from.withdraw(amount)
    to.deposit(amount)
  }
}
```

Figure 2.4: Simple interfaces for banks and accounts

```
object BankOfWaterloo extends Bank { bank =>
  class A(val id : Int, protected var _balance : Long) extends Account {
    type B = bank.type
  }
  private val accounts : mutable.ArrayBuffer[A] = mutable.ArrayBuffer()
  def lookupAccount(id : Int) = accounts(id)
  def createAccount(initialBalance : Long = 0) : A = {
    val account = new A(accounts.size, initialBalance)
    accounts += account
    account
  }
}

object WaterlooBank extends Bank {
  // defined similarly
}
```

Figure 2.5: A implementation of banks and accounts

```
def transfer(amount : Long, from : self.A, toBank : Bank, to : toBank.A)
```

This signature says that `from` needs to be of the account type held by this bank. A similar restriction is put on `to`. Looking at `to`, it has type `toBank.A`, which is a type dependent on a previous parameter `toBank`. This is called a *path dependent type*. As we can see from the definition of `Bank`, `toBank.A` is determined to be upper bounded by `Account`, and therefore I call this kind of bounded quantification *dependent bounded quantification*.

Two concrete implementations of `Bank`s are shown in Figure 2.5. In `BankOfWaterloo`, the abstract type member `A` is instantiated to be a class definition, in which `B` is instantiated to be the singleton type of the bank itself. This implementation chooses to use a mutable buffer to manage the accounts, and implements the account operations accordingly. `WaterlooBank` is another bank implemented in a similar manner.

Consider three fictional characters with bank accounts:

```
val catherina = BankOfWaterloo.createAccount(100)
val david = BankOfWaterloo.createAccount(200)
val elly = WaterlooBank.createAccount(300)
```

the following are some (correct) invocations of the `transfer` function.

```
// Catherina transfers 10 dollars to David
BankOfWaterloo.transfer(10, catherina, BankOfWaterloo, david)
// transfer to a different bank  also works
BankOfWaterloo.transfer(10, david, WaterlooBank, elly)
```

Intuitively, we do not want a bank to transfer money from an account that is not its customer, or transfer money to an account in the wrong bank. Both requirements are captured by the signature of `transfer` and enforced by the compiler during compilation time.

```
// David is with Bank of Waterloo
WaterlooBank.transfer(10, david, WaterlooBank, elly)
// 1 |WaterlooBank.transfer(10, david, WaterlooBank, elly)
//   |                          ^^^^^
//   |                          Found:    BankOfWaterloo.A(david)
//   |                          Required: WaterlooBank.A'

// Elly is with Waterloo Bank
BankOfWaterloo.transfer(10, catherina, BankOfWaterloo, elly)
// 1 |BankOfWaterloo.transfer(10, catherina, BankOfWaterloo, elly)
//   |                                                       ^^^^
//   |                          Found:    WaterlooBank.A(elly)
//   |                          Required: BankOfWaterloo.A'
```

As we can see, with path dependent types, one has finer grained control over the business logic, and can rely more on the type system to guard the program correctness. On the other hand, this also clearly poses realistic problems: how should path dependent types be type checked, and what does it mean to type check path dependent types correctly?

2.5.2 Intersection Types

Intersection types have a much longer history than path dependent types. They were introduced in the late 70's by a number of independent works, e.g. Coppo and Dezani-Ciancaglini [1978], Coppo et al. [1979], Hindley et al. [2013]. There are different variants of

```scala
trait Foo {
  def baz : A
}

trait Bar {
  def baz : B
}

val x : Foo & Bar = ???
val y : A & B = x.baz
```

Figure 2.6: An example of intersection types in Dotty

intersection types. Leivant [1990] introduced infinite intersection types. Davies [2005] introduced sort level checking with intersection types. Pierce [1991] investigated interactions between intersection types and the system $F_{<:}$, where the intersection types distributes over function types and universal types.

Intersection types help to express ideas frequently seen in object orientation elegantly in functional settings. For example, function overloading works very well with intersection types[1].

$$id : (\mathbb{Z} \to \mathbb{Z}) \wedge (\mathbb{R} \to \mathbb{R})$$

This type represents an identity function of both integers and real numbers, exclusively. In particular, it will not work for, e.g., complex numbers. Usually, in object oriented languages, the compiler is required to handle overloaded functions in a way not represented by the core calculus, but with intersection types, the core calculus has the capability to express overloading. Note that this type is more specific than a polymorphic identity function, which needs to work for all types.

The intersection types in Dotty and *DOT* share some similarities with the references above but are somewhat more restrictive. For example, one cannot find a term in *DOT* typed to the type of the *id* function above (in a consistent context). There is also no way to construct an instance of an intersection type $S \wedge U$ out of an instance of type S and one of type U. More details can be seen in the references above.

In the example shown in Figure 2.6, I defined two `trait`s `Foo` and `Bar`. They both have a member `baz` of types `A` and `B` respectively. If an instance of `Foo & Bar` is provided (am-

[1] However, this is not how overloading is implemented in Dotty.

persand is the syntax in Dotty to denote intersection types), then `baz` of this intersection type must be `A & B`, as expected. The reason behind this is that the language requires concrete implementations of `Foo & Bar` to implement `baz` to be at least `A & B`, enforced by the typing rules.

2.5.3 Wadlerfest *DOT*

There are multiple different calculi with the name *DOT*, the most prominent being the *OOPSLA DOT* [Rompf and Amin, 2016] and the *Wadlerfest DOT* [Amin et al., 2016]. In this thesis, Wadlerfest *DOT* is exclusively discussed.

Convention 8. *Generally, DOT might refer to a family of calculi. When referring to a concrete calculus, DOT exclusively means Wadlerfest DOT.* [2]

Definition 2.10. *The abstract syntax of DOT is defined in Figure 2.7. The typing rules are defined in Figure 2.8 and the subtyping rules are defined in Figure 2.9.*

In *DOT*, a term can be a variable, a value, a data field selection, an application of a function, or a let binding. A value can be either a definition of a lambda expression or an object. The body of an object is composed of an aggregation of any number of definitions. A definition may contain a data field member or a type member. Field members and type members are indexed by corresponding member labels, and the sets of labels for field members and type members are disjoint.

Types in *DOT* are rich. The top type $\top$, the bottom type $\bot$, type declarations, field declarations and function types should be straightforward and correspond to terms described above. $x.A$ denotes a path dependent type, or a path in short, and models the feature demonstrated in Section 2.5.1. Recursive types are types for objects. Intersection types model the feature demonstrated in Section 2.5.2.

Some typing rules in Figure 2.8 are quite standard. Var says a variable has the type which it binds to in the context. Sub is a standard rule to coerce any term to its supertype. All-I says that a function is typed with its body, with the parameter type assumed and pushed into the context. All-E types an application; it requires x to be typed as a function. Obj-I types an object; the body is typed with the object type itself pushed into

[2]The distinctions between both *DOT*'s are not negligible, and a rigorous comparison between these two requires substantial work. Some comparison was done and discussed in `https://hustmphrrr.github.io/blog/2019/compare-dots.html`.

x, y, z		**Variable**
a, b, c		**Term member**
A, B, C		**Type member**
$s, t, u ::=$		**Term**
	x	variable
	v	value
	$x.a$	selection
	$x\ y$	application
	$\text{let } x = t \text{ in } u_x$	let binding
$v ::=$		**Value**
	$\lambda(x : T)t_x$	lambda
	$\nu(x : T_x)d_x$	object

$d ::=$		**Definition**
	$\{a = t\}$	field definition
	$\{A = T\}$	type definition
	$d_1 \wedge d_2$	aggregation
$S, T, U ::=$		**Type**
	$\top$	top type
	$\bot$	bottom type
	$\{A : S..U\}$	type declaration
	$\{a : T\}$	field declaration
	$x.A$	path type
	$\forall(x : S)U_x$	function
	$\mu(x : T_x)$	recursive type
	$S \wedge U$	intersection

Figure 2.7: Abstract syntax of Wadlerfest *DOT* [Amin et al., 2016]

the context, bound to the self reference. This is why definitions in the same object can refer to each other. This rule also requires the domain of definitions to be unique so that the same label can never be defined twice in the same object. The definitions in the object body are typed using the Def-Trm, Def-Typ and Def-And rules. Obj-E says that a data field selection is typed if the variable x has a field declaration with label a. And refines the type of a variable if it can be typed in two ways. The Let rule is standard. Notice that x is required to not be free in the type of the body.

Rec-I and Rec-E are very specific to *DOT* to handle recursive types. These two rules permit packing / unpacking of recursive types. After Rec-E, if T is an intersection type, Obj-E can be used to access the data field, as in this proof.

$$\dfrac{\dfrac{\dfrac{}{x : \mu(z : \{a : T_z\}) \vdash_{DOT} x : \mu(z : \{a : T_z\})}\ \textsc{Var}}{x : \mu(z : \{a : T_z\}) \vdash_{DOT} x : \{a : T_x\}}\ \textsc{Rec-E}}{x : \mu(z : \{a : T_z\}) \vdash_{DOT} x.a : T_x}\ \textsc{Obj-E}$$

The subtyping rules are shown in in Figure 2.9. Refl and Trans are reflexivity and

Type Assignment

$$\frac{}{\Gamma \vdash_{DOT} x : \Gamma(x)}\ \textsc{Var} \qquad \frac{\Gamma \vdash_{DOT} t : S \quad \Gamma \vdash_{DOT} S <: U}{\Gamma \vdash_{DOT} t : U}\ \textsc{Sub}$$

$$\frac{\Gamma; x : S \vdash_{DOT} t : U}{\Gamma \vdash_{DOT} \lambda(x : S)t : \forall(x : S)U}\ \textsc{All-I} \qquad \frac{\Gamma \vdash_{DOT} x : \forall(z : S)U_z \quad \Gamma \vdash_{DOT} y : S}{\Gamma \vdash_{DOT} x\ y : U_y}\ \textsc{All-E}$$

$$\frac{\Gamma; x : T_x \vdash_{DOT} d : T_x \quad dom(d) \text{ is unique}}{\Gamma \vdash_{DOT} \nu(x : T_x)d_x : \mu(x : T_x)}\ \textsc{Obj-I} \qquad \frac{\Gamma \vdash_{DOT} x : \{a : T\}}{\Gamma \vdash_{DOT} x.a : T}\ \textsc{Obj-E}$$

$$\frac{\Gamma \vdash_{DOT} x : S \quad \Gamma \vdash x : U}{\Gamma \vdash_{DOT} x : S \wedge U}\ \textsc{And} \qquad \frac{\Gamma \vdash_{DOT} x : T_x}{\Gamma \vdash_{DOT} x : \mu(x : T_x)}\ \textsc{Rec-I}$$

$$\frac{\Gamma \vdash_{DOT} x : \mu(z : T_z)}{\Gamma \vdash_{DOT} x : T_x}\ \textsc{Rec-E} \qquad \frac{\Gamma \vdash_{DOT} t : S \quad \Gamma; x : S \vdash_{DOT} u : U \quad x \notin fv(U)}{\Gamma \vdash_{DOT} \text{let } x = t \text{ in } u : U}\ \textsc{Let}$$

Object Definition Type Assignment

$$\frac{\Gamma \vdash_{DOT} t : T}{\Gamma \vdash_{DOT} \{a = t\} : \{A : T\}}\ \textsc{Def-Trm} \qquad \frac{}{\Gamma \vdash_{DOT} \{A = T\} : \{A : T..T\}}\ \textsc{Def-Typ}$$

$$\frac{\Gamma \vdash_{DOT} d_1 : S \quad \Gamma \vdash_{DOT} d_2 : U}{\Gamma \vdash_{DOT} d_1 \wedge d_2 : S \wedge U}\ \textsc{Def-And}$$

Figure 2.8: Typing rules of Wadlerfest *DOT* [Amin et al., 2016]

transitivity, two basic structural properties of the calculus. Top and Bot define the bounded nature of the subtyping relation. And-I, And-E1 and And-E2 assert that the subtyping relation has a free meet-semilattice structure under intersection.

Bnd and Fld define the subtyping relations of type members and field members. Notice that in Bnd the first types in type declarations are in contravariant position, and therefore their subtyping relation is flipped. All expresses subtyping between dependent functions. This rule looks very similar to subtyping between universal types in $F_{<:}$.

The Sel1 and Sel2 rules express dependent bounded quantification. A path $x.A$ is

Subtyping

$$\frac{}{\Gamma \vdash_{DOT} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{DOT} \bot <: T}\ \textsc{Bot} \qquad \frac{}{\Gamma \vdash_{DOT} T <: T}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash_{DOT} S_2 <: S_1 \quad \Gamma \vdash_{DOT} U_1 <: U_2}{\Gamma \vdash_{DOT} \{A : S_1..U_1\} <: \{A : S_2..U_2\}}\ \textsc{Bnd} \qquad \frac{\Gamma \vdash_{DOT} T_1 <: T_2}{\Gamma \vdash_{DOT} \{a : T_1\} <: \{a : T_2\}}\ \textsc{Fld}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{DOT} S_2 <: S_1 \\ \Gamma ; x : S_2 \vdash_{DOT} U_1 <: U_2\end{array}}{\Gamma \vdash_{DOT} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2}\ \textsc{All} \qquad \frac{\begin{array}{c}\Gamma \vdash_{DOT} S <: T \\ \Gamma \vdash_{DOT} T <: U\end{array}}{\Gamma \vdash_{DOT} S <: U}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash_{DOT} x : \{A : S..U\}}{\Gamma \vdash_{DOT} S <: x.A}\ \textsc{Sel1} \qquad \frac{\Gamma \vdash_{DOT} x : \{A : S..U\}}{\Gamma \vdash_{DOT} x.A <: U}\ \textsc{Sel2} \qquad \frac{\begin{array}{c}\Gamma \vdash_{DOT} T <: S \\ \Gamma \vdash_{DOT} T <: U\end{array}}{\Gamma \vdash_{DOT} T <: S \wedge U}\ \textsc{And-I}$$

$$\frac{}{\Gamma \vdash_{DOT} S \wedge U <: S}\ \textsc{And-E1} \qquad \frac{}{\Gamma \vdash_{DOT} S \wedge U <: U}\ \textsc{And-E2}$$

Figure 2.9: Subtyping rules of Wadlerfest *DOT* [Amin et al., 2016]

bounded from two directions by the type declaration x can type to. In combination with the Trans rule, one can use the Sel1 and Sel2 to derive $\Gamma \vdash_{DOT} S <: U$. This ability can be used to introduce new subtyping relationships between any two types and has caused many of the challenges in the soundness proofs of *DOT* and related calculi.

2.5.4 Other *DOT* Calculi

Over time, many variants of *DOT* have been studied with various subsets of features. These calculi form a family of *DOT* calculi. Table 2.2 provides a summary of the calculi, together with their features. In order to make the names consistent in this thesis, I adopt the following convention.

Convention 9. *There are two different calculi defined in the literature with the name $D_{<:}$. I call the calculus defined in Amin et al. [2016] with lower bounds $D_{<:}$ and the one of Amin and Rompf [2017] without lower bounds $D_{<<:}$.*

[3] "Function" here means first-class function. "No" means functions are modeled as methods, which turns them second-class.

features	$\wedge$	$\vee$	lower bound	upper bound	recursive types	function[3]
DOT[Amin and Rompf, 2017, Rompf and Amin, 2016]	yes	yes	yes	yes	object directly	no
DOT[Amin et al., 2016, Rapoport et al., 2017]	yes	no	yes	yes	yes	yes
μDOT[Amin et al., 2014]	no	no	yes	yes	object directly	no
D[Amin and Rompf, 2017]	no	no	no	no	no	yes
$D_{<:}$[Amin and Rompf, 2017]/$D_{<<:}$	no	no	no	yes	no	yes
$D_{<:>}$[Amin and Rompf, 2017]/$D_{<:}$[Amin et al., 2016]	no	no	yes	yes	no	yes
$D_{\wedge}$	yes	no	yes	yes	no	yes
$\mu DART$	no	no	yes	yes	yes	yes
$jDOT$	yes	no	yes	yes	yes	yes

Table 2.2: *DOT* summary table

2.6 Related Work

2.6.1 Formalizations of Scala

Scala is a complex language and its formalization was a long process. In Odersky et al. [2003], νObj was introduced as the first attempt. The thesis introduced path-dependent types and came with a type soundness proof. It showed that type checking is undecidable. Featherweight Scala was later proposed in Cremet et al. [2006]. It provided a type checking algorithm for the calculus while the calculus itself was not proven sound. Scalina was introduced in Moors et al. [2007], and its soundness was also not proved.

The first version of *DOT* was formulated in Amin et al. [2012]. It came with intersection types and union types, object creations and many other constructs. This version of *DOT* contains a type refinement construct similar to the Scala language. Amin et al. [2014] introduced μDOT_T and μDOT, two other variants of DOT. These two variants have type members with upper bounds, but with no intersection and union type. Both calculi were proven sound. The soundness proof of OOPSLA DOT was established in Rompf and Amin [2016]. OOPSLA DOT has intersection and union types, as well as subtyping relation between objects. The soundness property in the thesis was proved with respect to a small step operational semantics. Amin and Rompf [2017] provided the soundness proof of OOPSLA DOT in a different style, with respect to a big step operational semantics. Amin et al. [2016] provided another version of DOT, Wadlerfest DOT, which has intersection types but no union types and uses administrative normal form. It also formulated $D_{<:}$, a weakened form of DOT. Rapoport et al. [2017] simplified the soundness proof of Wadlerfest DOT by modularizing it, and proved soundness with respect to a small step operational semantics using evaluation contexts. All of these works focused on the declarative forms of the calculi.

2.6.2 Undecidability of subtyping

There has been much work related to proving undecidability under certain settings of subtyping. Pierce [1992] presented a chain of reductions from two counter machines (TCM) to $F_{<:}$ and showed $F_{<:}$ undecidable. Kennedy and Pierce [2006] investigated a nominal calculus with variance, modelling the situations in Java, C# and Scala, and showed that this calculus is undecidable due to three factors: contraviarant generics, large class hierarchies, and multiple inheritance. Wehr and Thiemann [2009] considered two calculi with existential types, $\mathcal{EX}_{impl}$ and $\mathcal{EX}_{uplo}$, and proved both to be undecidable. Moreover, in $\mathcal{EX}_{uplo}$, each type variable has either upper or lower bounds but not both, so this calculus is related to $D_{<:}$, but since no variable has both lower and upper bounds, it does not expose the bad bounds phenomenon. Grigore [2017] proved Java generics undecidable by reducing Turing machines to a fragment of Java with contravariance.

2.6.3 Algorithmic (sub)typing

So far, work on the DOT calculi mainly focused on soundness proofs Amin et al. [2016], Rompf and Amin [2016], Rapoport et al. [2017]. Nieto [2017] presented step subtyping as a partial algorithm for DOT typing. In this thesis, we have shown that the fragment of $D_{<:}$ typed by step subtyping is kernel $D_{<:}$. Aspinall and Compagnoni [2001] showed a calculus with dependent types and subtyping that is decidable due to the lack of a $\top$ type.

Greenman et al. [2014] identified the Material-Shape Separation. This separation describes two different usages of interfaces, and as long as no interface is used in both ways, the type checking problem is decidable by a simple algorithm.

2.6.4 Formalization of undecidability proofs

The undecidability proof in this thesis has been mechanized in Agda. There are other fundamental results on formalizing proofs of undecidability. Forster et al. [2018] mechanized undecibility proofs of various well-known undecidable problems, including the post correspondence problem (PCP), string rewriting (SR) and the modified post correspondence problem. Their proofs are based on Turing machines. In contrast, Forster and Smolka [2017] used a call-by-value lambda calculus as computational model. Forster and Larchey-Wendling [2019] proved undecibility of intuitionistic linear logic by reducing from PCP.

Chapter 3

The Undecidability of $D_{<:}$

In this chapter, I will present decidability analysis of $D_{<:}$. Decidability analysis focuses on understanding whether there exist algorithms to decide type checking and subtyping problems of a calculus, and the result can contribute to deeper insights in algorithmic analysis. For $D_{<:}$, the answer is unfortunately negative.

At the beginning, I start with comparing $D_{<:}$ and $F_{<:}$, revealing a number of phenomena that exist in $D_{<:}$ but not in $F_{<:}$. Then I define *reduction*, the main technique used in this thesis to prove undecidability of decision problems. Amin et al. [2016] defines an interpretation from $F_{<:}$ types to $D_{<:}$ types and proves that if subtyping relates between two $F_{<:}$ types, then it also holds between the corresponding $D_{<:}$ types. However, reduction requires an if and only if proof. As I will show later, there is a counterexample to the only if direction and therefore the proposed interpretation function cannot be used to establish undecidability of $D_{<:}$. It turns out that proving undecidability of $D_{<:}$ is very challenging and it requires to reformulate the definition of the calculus into *normal form*. A normal form definition does not have transitivity as an explicit rule in definition, but transitivity can be derived from the defining rules as a theorem. In order to derive the normal form for $D_{<:}$, I propose *small-step subtyping*, a conceptual tool used to analyze variants of $F_{<:}$ and $D_{<:}$. After applying small-step subtyping analysis, I am able to derive normal forms for multiple calculi, including $D_{<:}$, which directly leads to the undecidability proof of subtyping. Finally, I also establish the undecidability proof of type checking in $D_{<:}$.

The calculi examined in this thesis with machine-verified undecidability proofs include: $F_{<:}$, $F^{-}_{<:}$, $F_{<:}$ with bottom ($\bot$), $F^{-}_{<:>}$, $D_{<<:}$, $D_{<:}$ excluding transitivity rule, and $D_{<:}$ (these calculi will be defined later). The conclusion in turn depends on the fact that $F_{<:}$ deterministic ($F^{d}_{<:}$) is undecidable, which is proved in Pierce [1992] and is discussed in

Section 2.3.

To complete this chapter, I will also discuss what the proof and the techniques can say about the undecidability of all calculi that extends $D_{<:}$, including DOT.

3.1 Definition of $D_{<:}$

Definition 3.1. *$D_{<:}$ is the calculus defined in Figure 3.1.*

$D_{<:}$ is a syntactic subset of DOT. Recall the definition of DOT can be found in Definition 2.10. $D_{<:}$ is simpler because it has no intersection types and recursive types; among the three core features mentioned in Section 2.5, only path dependent types are kept. Moreover, in $D_{<:}$, there is only one type member label, which is A. Correspondingly, it has no objects as values. Instead, it has a kind of values called type tags. In a type tag, a type is bound to the only type member label A. Intuitively, $D_{<:}$ is a simpler calculus than DOT.

The type assignment and subtyping rules of $D_{<:}$ are the results of reasonably removing the rules related to the dropped features.

Notice that in $D_{<:}$, the type assignment rules and subtyping rules are mutually recursive. This is because the Sel1 and Sel2 rules refer to type assignment, in which the Sub rule refers back to subtyping. The following lemma shows that the definition of subtyping can be separated from typing.

Lemma 3.1. *(unravelling of $D_{<:}$ subtyping) Sel1 and Sel2 can be changed to the following form, and the resulting calculus is equivalent to the original one.*

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : S..\top\}}{\Gamma \vdash_{D_{<:}} S <: x.A}\ \textsc{Sel1'} \qquad \frac{\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : \bot..U\}}{\Gamma \vdash_{D_{<:}} x.A <: U}\ \textsc{Sel2'}$$

Proof. This follows directly from the fact that the only typing rules that apply to variables are the Var and Sub rules. □

Since this equivalent form is simpler to work with, from now on, I will use this form as the default $D_{<:}$ subtyping.

Compared to $F_{<:}$ (Figure 2.1), $D_{<:}$ does not syntactically differ much. The only differences are the bottom type $\bot$, type declarations, and dependent function types. Dependent

x, y, z		**Variable**
$v ::=$		**Value**
	$\{A = T\}$	type tag
	$\lambda(x : T)t_x$	lambda
$s, t, u ::=$		**Term**
	x	variable
	v	value
	$x\ y$	application
	let $x = t$ in u_x	let binding

$S, T, U ::=$		**Type**
	$\top$	top type
	$\bot$	bottom type
	$\{A : S..U\}$	type declaration
	$x.A$	path type
	$\forall(x : S)U_x$	function

Type Assignment

$$\frac{}{\Gamma \vdash_{D_{<:}} x : \Gamma(x)}\ \textsc{Var} \qquad \frac{\Gamma \vdash_{D_{<:}} t : S \quad \Gamma \vdash_{D_{<:}} S <: U}{\Gamma \vdash_{D_{<:}} t : U}\ \textsc{Sub}$$

$$\frac{\Gamma; x : S \vdash_{D_{<:}} t : U}{\Gamma \vdash_{D_{<:}} \lambda(x : S)t : \forall(x : S)U}\ \textsc{All-I} \qquad \frac{\Gamma \vdash_{D_{<:}} x : \forall(z : S)U_z \quad \Gamma \vdash_{D_{<:}} y : S}{\Gamma \vdash_{D_{<:}} x\ y : U_y}\ \textsc{All-E}$$

$$\frac{}{\Gamma \vdash_{D_{<:}} \{A = T\} : \{A : T..T\}}\ \textsc{Typ-I} \qquad \frac{\Gamma \vdash_{D_{<:}} t : S \quad \Gamma; x : S \vdash_{D_{<:}} u : U \quad x \notin fv(U)}{\Gamma \vdash_{D_{<:}} \text{let } x = t \text{ in } u : U}\ \textsc{Let}$$

Subtyping

$$\frac{}{\Gamma \vdash_{D_{<:}} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{D_{<:}} \bot <: T}\ \textsc{Bot} \qquad \frac{}{\Gamma \vdash_{D_{<:}} T <: T}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash_{D_{<:}} S_2 <: S_1 \quad \Gamma \vdash_{D_{<:}} U_1 <: U_2}{\Gamma \vdash_{D_{<:}} \{A : S_1..U_1\} <: \{A : S_2..U_2\}}\ \textsc{Bnd} \qquad \frac{\begin{array}{c}\Gamma \vdash_{D_{<:}} S_2 <: S_1 \\ \Gamma; x : S_2 \vdash_{D_{<:}} U_1 <: U_2\end{array}}{\Gamma \vdash_{D_{<:}} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2}\ \textsc{All}$$

$$\frac{\Gamma \vdash_{D_{<:}} x : \{A : S..U\}}{\Gamma \vdash_{D_{<:}} S <: x.A}\ \textsc{Sel1} \qquad \frac{\Gamma \vdash_{D_{<:}} x : \{A : S..U\}}{\Gamma \vdash_{D_{<:}} x.A <: U}\ \textsc{Sel2} \qquad \frac{\begin{array}{c}\Gamma \vdash_{D_{<:}} S <: T \\ \Gamma \vdash_{D_{<:}} T <: U\end{array}}{\Gamma \vdash_{D_{<:}} S <: U}\ \textsc{Trans}$$

Figure 3.1: Definition of $D_{<:}$ [Amin et al., 2016]

function types allow return types to depend on the input parameter (as indicated by the subscript U_x), which combines functions and universal types in $F_{<:}$. Nonetheless, it turns out that path dependent types in $D_{<:}$ have more complicated behaviors than type variables in $F_{<:}$, as illustrated by the following examples.

Path types cascade: Consider the following subtyping derivation in $D_{<:}$.

$$\Gamma = x : \{A : \bot..\{A : \bot..T\}\}; y : x.A$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{D_{<:}} \{A : \bot..\{A : \bot..T\}\} <: \{A : \bot..\{A : \bot..T\}\}}\ \textsc{Refl}}{\Gamma \vdash_{D_{<:}} x.A <: \{A : \bot..T\}}\ \textsc{Sel2'}}{\Gamma \vdash_{D_{<:}} y.A <: T}\ \textsc{Sel2'}$$

In this subtyping derivation, since x has type $\{A : \bot..\{A : \bot..T\}\}$, $x.A$ is shown a subtype of $\{A : \bot..T\}$. As y has type $x.A$, by the Sel2' rule $y.A$ is a subtype of T.

In this derivation, the Sel2' rule applying to $y.A$ allows the subtyping derivation to look one layer deeper into $\{A : \bot..\{A : \bot..T\}\}$ by showing $x.A$ being a subtype of the type declaration. This behavior makes nested type declarations a non-trivial type structure. In $F_{<:}$, on the other hand, this phenomenon does not happen because type variable bindings are just context lookup and do not have this kind of nested structure.

In Section 3.9, I will show that how this impacts the proof of undecidability of $D_{<:}$.

Bad bounds: The pattern of bad bounds is discussed in Rapoport et al. [2017], Rompf and Amin [2016]. Using the Sel1', Sel2' and Trans rules, one can prove subtyping between *any* type S and U in an appropriate context. Consider the following derivation tree:

$$\text{assume } \Gamma(x) = \{A : S..U\}$$

$$\dfrac{\dfrac{\dfrac{\text{straightforward}}{\Gamma \vdash_{D_{<:}} \{A : S..\top\} <: \{A : S..U\}}\ \textsc{Bnd}}{\Gamma \vdash_{D_{<:}} S <: x.A}\ \textsc{Sel1'} \qquad \dfrac{\text{same as left}}{\Gamma \vdash_{D_{<:}} x.A <: U}\ \textsc{Sel2'}}{\Gamma \vdash_{D_{<:}} S <: U}\ \textsc{Trans}$$

The derivation uses Sel1' to find the lower bound of $x.A$ and Sel2' to find the upper bound of $x.A$, and finally uses Trans on $x.A$ to make S a subtype of U.

In $F_{<:}$, on the other hand, it is easy to show that, for example, a supertype of a universal type is either a universal type or $\top$. Such properties are called *inversion properties* and they do not hold in general for $D_{<:}$ because it is possible to derive subtyping between any types S and U.

$\bot$ proves everything: Recall the definition of $F_{<:}$ with $\bot$ from Definition 2.6. In $F_{<:}$ with $\bot$, even if $X <: \bot \in \Gamma$ for some type variable X, other subtyping relations remain unchanged. However, in $D_{<:}$, $\bot$ can be used to introduce subtyping between all types using bad bounds:

$$\text{assume } \Gamma(x) = \bot, \text{then for all } S \text{ and } U,$$

$$\cfrac{\cfrac{}{\Gamma \vdash_{D_{<:}} \bot <: \{A : S..\top\}}\ \textsc{Bot}}{\Gamma \vdash_{D_{<:}} S <: x.A}\ \textsc{Sel1'} \qquad \cfrac{\cfrac{}{\Gamma \vdash_{D_{<:}} \bot <: \{A : \bot..U\}}\ \textsc{Bot}}{\Gamma \vdash_{D_{<:}} x.A <: U}\ \textsc{Sel2'}$$

These derivations hold for *any* S and U. Using the bad bound derivation, we can see that $\Gamma \vdash_{D_{<:}} S <: U$ holds.

These examples show that $D_{<:}$ has very different behaviors from $F_{<:}$. This can be confusing, because syntactically, $D_{<:}$ is not very differerent from $F_{<:}$, and it can misguide one to think that $D_{<:}$ just "moderately extends" $F_{<:}$. As will be shown in Section 3.3, this claim, however, is mistaken.

3.2 Definition of Undecidability

In this section, let us first consider a formal definition of undecidability of decision problems. The definition I use here relies on the concept of reducibility between decision problems.

Definition 3.2. *[Martin, 2010, Definition 12.1a] If Q and P are decision problems, we say Q is reducible to P ($Q \leq P$) if there is an algorithmic procedure F that allows us, given an arbitrary instance I_1 of Q, to find an instance $F(I_1)$ of P so that for every I_1, I_1 is a yes-instance of Q if and only if $F(I_1)$ is a yes-instance of P.*

Theorem 3.2. *[Martin, 2010, Theorem 12.1a] If Q and P are decision problems and $Q \leq P$, then if P can be solved algorithmically, so can Q.*

Theorem 3.3. *[Martin, 2010, Theorem 12.2] The halting problem $HALT$ is undecidable. Namely, it cannot be solved by any algorithm.*

Definition 3.3. *If P is a decision procedure, and $HALT \leq P$, then P is undecidable.*

Intuitively, if $P \leq Q$, then P is a simpler problem than Q and a procedure for Q can be used to solve P. Notice that the definition of reducibility requires an equivalence proof due to the "if and only if" statement. To emphasize the equivalence nature, I would define undecidability in a different form which can make the problem more obvious.

Definition 3.4. *(Undecidability as an adversarial game) Consider a target decision problem P. Merlin is a wizard who claims to have access to true magic, and therefore be able to decide P. He is so confident that he would also offer a complete proof accompanying each yes answer he gives.*

Sherlock is a skeptical detective. He questions the Merlin's ability, and comes up with the following scheme in order to disprove Merlin's claim.

$$\begin{tikzcd}
\textit{an instance of } Q \arrow[r, "\textit{Step 1}"] & \textit{an instance of } P \arrow[dr] & \\
& & \textit{Merlin} \arrow[dl] \\
\textit{a proof of } Q & \textit{a proof of } P \arrow[l, "\textit{Step 2}"] &
\end{tikzcd}$$

Q is some selected problem that is equivalent to the halting problem. Sherlock will first encode an instance of problem Q to P in step 1, and ask Merlin to solve it. When Merlin gives positive answer, Sherlock will continue with step 2, so that he obtains a proof of a positive answer of the same Q.

P is undecidable, iff Sherlock achieves both steps and therefore proves Merlin is wrong.

This definition requires a reducibility proof of $Q \leq P$ and that Q is undecidable, namely $HALT \leq Q$. Therefore one can conclude $HALT \leq P$ and P is indeed undecidable based on the formal definition. This definition additionally sets up a game scenario which helps to understand the particular problem of the incomplete proof in Amin et al. [2016].

3.3 The Incomplete Proof

The incomplete proof of the undecidability of $D_{<:}$ subtyping presented in Amin et al. [2016] is the following. Based on Theorem 2.4, $F_{<:}$ is chosen to be Q. Then Amin et al. [2016] defines the following two interpretation functions.

Definition 3.5. *The interpretation function, $[\![\cdot]\!]$, interprets types in $F_{<:}$ to types in $D_{<:}$.*

$$
\begin{aligned}
[\![\top]\!] &= \top \\
[\![X]\!] &= x_X.A \\
[\![S \to U]\!] &= \forall(x : [\![S]\!])[\![U]\!] && \text{(function case)} \\
[\![\forall X <: S.U]\!] &= \forall(x_X : \{A : \bot..[\![S]\!]\})[\![U]\!]
\end{aligned}
$$

The context interpretation function, $\langle\!\langle\cdot\rangle\!\rangle$, interprets contexts in $F_{<:}$ to ones in $D_{<:}$.

$$
\begin{aligned}
\langle\!\langle\bullet\rangle\!\rangle &= \bullet \\
\langle\!\langle\Gamma; X <: T\rangle\!\rangle &= \langle\!\langle\Gamma\rangle\!\rangle; x_X : \{A : \bot..[\![T]\!]\}
\end{aligned}
$$

Convention 10. *For the rest of this thesis, I will use $[\![\cdot]\!]$ to denote a type interpretation function and $\langle\!\langle\cdot\rangle\!\rangle$ to denote a context interpretation function. These functions convert corresponding objects from one calculus to another. The domain and codomain calculi of these interpretation functions will be clear from the context.*

In the interpretation function $[\![\cdot]\!]$, α-conversion is needed, and some correspondence between type variables in $F_{<:}$ and variables in $D_{<:}$ is assumed, as indicated by the notation x_X.

The following theorem is proved in Amin et al. [2016], corresponding to step 1 in Definition 3.4.

Theorem 3.4. *If $\Gamma \vdash_{F_{<:}} S <: U$, then $\langle\!\langle\Gamma\rangle\!\rangle \vdash_{D_{<:}} [\![S]\!] <: [\![U]\!]$.*

What is missing is the only if direction, namely the following conjecture:

Conjecture 1. *If $\langle\!\langle\Gamma\rangle\!\rangle \vdash_{D_{<:}} [\![S]\!] <: [\![U]\!]$, then $\Gamma \vdash_{F_{<:}} S <: U$.*

This conjecture corresponds to step 2 and to establish the proof, this step is necessary.

Conjecture 1 is necessary. Consider the situation in which the following rule is added into $D_{<:}$ subtyping.

$$\frac{}{\Gamma \vdash_{D_{<:}} S <: U}\ \textsc{Trivial}$$

This rule trivializes the whole subtyping relation and admits subtyping between any pair of types in $D_{<:}$. It is clear that with this rule, $D_{<:}$ subtyping is trivially decidable: simply all subtyping relations are admitted. If Theorem 3.4 sufficed, then it means an obviously decidable problem is undecidable, which is a contradiction. Projecting the situation in the game, step 2 provides Sherlock a chance to question Merlin's yes answers and therefore this step is definitely necessary.[1]

A counterexample to Conjecture 1: Conjecture 1 might appear to be quite convincing and one might think that it should be trivial to show, but this conjecture is, unfortunately, *false*. Consider following subtyping problem in $F_{<:}$.

$$\vdash_{F_{<:}} \top \to \top <: \forall X <: \top.\top$$

This is clearly false in $F_{<:}$: there is no subtyping relation between function types and universal types in any context.

However, these two types have the following interpretations as $D_{<:}$ types.

$$\llbracket \top \to \top \rrbracket = \forall(x : \top).\top$$
$$\llbracket \forall X <: \top.\top \rrbracket = \forall(x_X : \{A : \bot..\top\}).\top$$

These two interpreted types in $D_{<:}$ *are* subtypes, as witnessed by the following derivation tree.

$$\dfrac{\dfrac{}{\vdash_{D_{<:}} \{A : \bot..\top\} <: \top}\textsc{Top} \qquad \dfrac{}{x : \{A : \bot..\top\} \vdash_{D_{<:}} \top <: \top}\textsc{Refl}}{\vdash_{D_{<:}} \forall(x : \top).\top <: \forall(x_X : \{A : \bot..\top\}).\top}\textsc{All}$$

For this input, Merlin gives a yes answer, together with the proof above. However, Sherlock can never show that $F_{<:}$ also has a positive answer.

Based on the two counterarguments, the interpretation functions from $F_{<:}$ cannot be used to establish the undecidability of $D_{<:}$. To unblock the situation, let us consider a modified version of $F_{<:}$.

3.4 $F^{-}_{<:}$ as Q

Notice that while both function types and universal types are unrelated types in $F_{<:}$, the interpretation function $\llbracket\cdot\rrbracket$ maps them to dependent function types in $D_{<:}$ and thus

[1]The credit for this counterargument goes to Abel Nieto.

allows both kinds of types to interfere in the image. However, some careful review of Pierce [1992] indicates that function types do not actually participate in the undecidability proof of $F_{<:}$. This implies that there can be an undecidable variant of $F_{<:}$ without function types, so that the interference is eliminated.

Definition 3.6. *$F_{<:}^{-}$ is obtained from $F_{<:}$ defined in Figure 2.2 by removing function types ($\rightarrow$) and the* FUN *rule.*

Indeed, it can be shown that $F_{<:}^{-}$ subtyping is undecidable.

Theorem 3.5. *$F_{<:}^{-}$ subtyping is undecidable.*

Proof. This conclusion is drawn from reduction from $F_{<:}^{d}$ (defined in Definition 2.9). Recall that $F_{<:}^{d}$ is a calculus parameterized over a natural number n. The following reduction therefore needs to hold for all n.

$$\Gamma \vdash_{F_{<:}^{d}} S <: U \text{ iff } \langle\!\langle \Gamma \rangle\!\rangle \vdash_{F_{<:}^{-}} [\![S]\!] <: [\![U]\!]$$

The interpretation functions here are the same as the ones in the proof of Theorem 2.5. Though the interpretation functions maps types and contexts from $F_{<:}^{d}$ to $F_{<:}$. Since these functions do not map to function types in their images, switching their codomain to $F_{<:}^{-}$ is well defined.

Since $F_{<:}^{d}$ is undecidable (Theorem 2.6), the undecidability of $F_{<:}^{-}$ follows by reduction. $\square$

The interpretation functions from Definition 3.5 which map from $F_{<:}$ to $D_{<:}$ are applicable to types and contexts in $F_{<:}^{-}$ but without the function case since function types are removed in $F_{<:}^{-}$.

Reviewing Pierce [1992], $F_{<:}^{-}$ has all the essential ingredients of undecidability of bounded quantification: $\top$, type variables, and universal types, and the interpretation function $[\![\cdot]\!]$ no longer has the interference pointed out above. Therefore, this calculus looks to be the right calculus to reduce from. Later development indeed agrees with this intuition.

3.5 An Attempt at An Undecidability Proof

In order to proceed to the undecidability proof, I will first characterize the interpretation functions $[\![\cdot]\!]$ and $\langle\!\langle \cdot \rangle\!\rangle$. First I define covariant types in order to capture the image of $[\![\cdot]\!]$ in covariant positions.

Definition 3.7. *(covariant types) A type T in $D_{<:}$ is covariant if it inductively satisfies the following predicates.*

1. *T is $\top$, or*

2. *T is a path type $x.A$, or*

3. *T is a function of the form $\forall(x : \{A : \bot..S\})U$, where both S and U are covariant.*

Similarly, contraviant types are used to capture the image of $[\![\cdot]\!]$ in contravariant positions.

Definition 3.8. *(contravariant types) A type T in $D_{<:}$ is contravariant if it is in the form $\{A : \bot..U\}$, where U is covariant.*

The concept of contravariant types is generalized to contexts.

Definition 3.9. *(contravariant contexts) A context Γ in $D_{<:}$ is contravariant if all the types in it are contravariant.*

I use covariant types to capture the image of $[\![\cdot]\!]$ and use contravariant contexts to capture the image of $\langle\!\langle\cdot\rangle\!\rangle$. It turns out that the concept of covariant types is complete in this sense.

Lemma 3.6. *$[\![\cdot]\!]$ is an isomorphism between types in $F^-_{<:}$ and covariant types in $D_{<:}$.*

$$[\![\cdot]\!] : T_{F^-_{<:}} \xrightarrow{\sim} \{T_{D_{<:}} \,|\, T_{D_{<:}} \text{ is covariant}\}$$

In Section 3.1, I showed that $D_{<:}$ has bad bounds and therefore does not have inversion properties. Nonetheless, it is still possible to identify a class of contexts in which inversion properties are recovered.

Definition 3.10. *(invertible contexts) A context Γ in $D_{<:}$ is invertible, if all of the following hold.*

1. *No variable binds to $\bot$,*

2. *No variable binds to types in the form $\{A : S..\bot\}$ for any S,*

3. *No variable binds to types in the form $\{A : T..\{A : S..U\}\}$ for any T, S and U, and*

4. *If a variable binds to $\{A : S..U\}$, then $S = \bot$.*

The following lemma connects contravariant contexts with invertible contexts and shows that all contexts in the image of $\langle\!\langle \cdot \rangle\!\rangle$ are invertible.

Lemma 3.7. *(properties of contexts) Both of the following hold.*

1. *All contravariant contexts are invertible.*
2. *For a context Γ in $F^-_{<:}$, $\langle\!\langle \Gamma \rangle\!\rangle$ is contravariant and therefore also invertible.*

Within invertible contexts, inversion properties are recovered, as indicated by the following lemmas.

Lemma 3.8. *(super types in invertible contexts) If a context Γ is invertible, then all of the following hold.*

1. *If $\Gamma \vdash_{D_{<:}} \top <: T$, then $T = \top$.*
2. *If $\Gamma \vdash_{D_{<:}} \{A : S..U\} <: T$, then $T = \top$ or T has the form $\{A : S'..U'\}$.*
3. *If $\Gamma \vdash_{D_{<:}} \forall(x : S)U <: T$, then $T = \top$ or T has the form $\forall(x : S')U'$.*

Lemma 3.9. *(subtypes in invertible contexts) If a context Γ is invertible, then all of the following hold.*

1. *If $\Gamma \vdash_{D_{<:}} T <: \bot$, then $T = \bot$.*
2. *If $\Gamma \vdash_{D_{<:}} T <: \{A : S..U\}$, then $T = \bot$ or T has the form of $\{A : S'..U'\}$.*
3. *If $\Gamma \vdash_{D_{<:}} T <: \forall(x : S)U$, then $T = \bot$ or T is some path $y.A$, or T has the form of $\forall(x : S')U'$.*
4. *If $\Gamma \vdash_{D_{<:}} T <: x.A$, then $T = \bot$, $T = x.A$, or T is some path $y.A$ from which $x.A$ can be reached.*

Lemma 3.10. *(inversion of subtyping in invertible contexts) If a context Γ is invertible, then the following hold.*

1. *If $\Gamma \vdash_{D_{<:}} \{A : S_1..U_1\} <: \{A : S_2..U_2\}$, then $\Gamma \vdash_{D_{<:}} S_2 <: S_1$ and $\Gamma \vdash_{D_{<:}} U_1 <: U_2$.*
2. *If $\Gamma \vdash_{D_{<:}} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2$, then $\Gamma \vdash_{D_{<:}} S_2 <: S_1$ and $\Gamma; x : S_2 \vdash_{D_{<:}} U_1 <: U_2$.*

The above lemmas show that $D_{<:}$ starts to have much closer trait to $F^-_{<:}$ within invertible contexts (and hence in the image of $\langle\!\langle \cdot \rangle\!\rangle$), and suggest that we are just one step away from proving that $D_{<:}$ subtyping is undecidable, but the Trans rule gives one more problem.

A proof fragment: To prove undecidability of $D_{<:}$ subtyping, one of the necessary lemmas is the following conjecture.

Conjecture 2.

$$\textit{If } \langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} [\![S]\!] <: [\![U]\!], \textit{ then } \Gamma \vdash_{F^-_{<:}} S <: U$$

In order to apply induction directly, the statement needs to be rewritten to an equivalent form so that the types in $D_{<:}$ are exposed. Due to Lemma 3.6, we know the type interpretation function $[\![\cdot]\!]$ is an isomorphism, so the types can be exposed by applying the inverse interpretation function.

Conjecture 3. *Let* $[\![\cdot]\!]^{-1}$ *be the inverse function of* $[\![\cdot]\!]$ *The target conjecture becomes the following:*

$$\textit{If } \langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} S <: U \textit{ and } S \textit{ and } U \textit{ are covariant, then } \Gamma \vdash_{F^-_{<:}} [\![S]\!]^{-1} <: [\![U]\!]^{-1}$$

When attempting to prove this theorem by induction on the derivation in $D_{<:}$, the Trans rule generates a case with the following antecedents:

1. For some T, $\langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} S <: T$,
2. $\langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} T <: U$,
3. S and U are covariant,
4. Inductive hypothesis: if $\langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} T_1 <: T_2$, and T_1 and T_2 are covariant, then $\Gamma \vdash_{F^-_{<:}} [\![T_1]\!]^{-1} <: [\![T_2]\!]^{-1}$.

The problem is that T does not have to be covariant, so the induction hypothesis cannot be applied to antecedent 1 or antecedent 2.

A counterexample for the Trans case: Here is a concrete example of how Merlin can take advantage of Trans and Top to cause Sherlock trouble in coming up with a proper step 2. Let $x <: T$ be syntactical sugar for $x : \{A : \bot..T\}$. The following is a proof of $\vdash_{D_{<:}} \forall(x <: \top)\top <: \top$ given by Merlin.

$$\dfrac{\dfrac{\mathcal{D}}{\vdash_{D_{<:}} \forall(x <: \top)\top <: \forall(x : \{A : \top..\bot\})x.A} \qquad \dfrac{\text{Top}}{\vdash_{D_{<:}} \forall(x : \{A : \top..\bot\})x.A <: \top}}{\vdash_{D_{<:}} \forall(x <: \top)\top <: \top}\ \text{Trans}$$

In the above, $\mathcal{D}$ represents the following subderivation.

$$\dfrac{\dfrac{\text{straightforward}}{\vdash_{D_{<:}} \{A : \top..\bot\} <: \{A : \bot..\top\}}\textsc{Bnd} \qquad \dfrac{\text{straightforward}}{x : \{A : \top..\bot\} \vdash_{D_{<:}} \top <: x.A}\textsc{Sel1}}{\vdash_{D_{<:}} \forall(x <: \top)\top <: \forall(x : \{A : \top..\bot\})x.A}\textsc{All}$$

Here, Merlin proves $\vdash_{D_{<:}} \forall(x <: \top)\top <: \top$ via transitivity through $\forall(x : \{A : \top..\bot\})x.A$. In this proof, both resulting types $\forall(x <: \top)\top$ and $\top$ are covariant. $\forall(x : \{A : \top..\bot\})x.A$ is T in the inductive proof step above and is not covariant, as it would at least require the lower bound of the type declaration to be $\bot$. This counterexample shows that the target conjecture cannot be proven by induction.

In the game scenario setup, due to the adversarial status between Sherlock and Merlin, it serves Merlin's interest to complicate the derivation for a yes answer to prevent Sherlock from coming up with a proper step 2. This counterexample uses the fact that the Top rule can conclude $\Gamma \vdash T <: \top$ without requiring T to be covariant and hence break the intended invariant. This T can then be subsequently hidden inside of the Trans rule. In general, this counterexample raises the following question.

Question 3. *Can Merlin make use of* Trans *and* Top *smartly, so that there is a subtyping derivation between covariant types in $D_{<:}$ that cannot be convertible back to a derivation in $F^-_{<:}$?*

In particular, this question asks whether it is possible for Sherlock to remove *all* non-covariant types hidden inside of the Trans rule. If the answer is negative, then Merlin wins and $F^-_{<:}$ can also not be used to establish the undecidability of $D_{<:}$.

As a side note, the previous analysis focuses on *one* particular set of interpretation functions. This analysis cannot deny the existence of other possible restrictions or interpretations, from which the undecidability of $D_{<:}$ can be established. However, I consider this alternative approach sub-optimal. Firstly, the current interpretation functions are the most straightforward interpretation from $F^-_{<:}$ and $D_{<:}$ so other interpretations are harder to obtain intuition from. Secondly, the purpose of the analysis is to understand the calculus extensively and generally. In contrast, analysis under certain restrictions is too specific; different problems might require different restrictions, so techniques working well with one restriction might not work well for problems requiring different restrictions. Hence, the path I take here is to concentrate on the study of the whole $D_{<:}$ without any restrictions. Now, let us rewind the thoughts and start from the beginning of the problem.

3.6 How Was Undecidability of $F_{<:}$ Proved?

Transitivity is also a property of subtyping in $F_{<:}$, so Sherlock wonders why this issue does not appear in Pierce [1992]. The answer is that Pierce worked on $F_{<:}$ normal form directly (Figure 2.2). Recall that Curien and Ghelli [1990] proved that two definitions of $F_{<:}$ subtyping, non-normal form (Figure 2.1) and normal form, are equivalent (Theorem 2.3). In $F_{<:}$ normal form, transitivity is not part of the definition but a provable property.

More generally, the previous proof fragment failed because the Trans rule in $D_{<:}$ uses a type that does not appear anywhere in the input context and types. If all the rules in a subtyping definition do not rely on hidden types, then the undecidability of this definition should be much easier to prove. To capture the characteristics of this particular kind of declarative forms of calculi, I make the definition explicit.

Definition 3.11. *A subtyping definition is in normal form if the premises of every rule are defined in terms of syntactic subterms of the conclusion.*

Notice that subterms here include subterms of the context Γ. Consider the following rule in $F_{<:}$ normal form.

$$\frac{X <: T \in \Gamma \quad \Gamma \vdash_{F_{<:}} T <: U}{\Gamma \vdash_{F_{<:}} X <: U} \text{ TVAR'}$$

Though T is not a subterm of X or U, notice that T is a result of context lookup of X and therefore T is a subterm of Γ.

Consider the Trans rule in $F_{<:}$ that is removed from normal form.

$$\frac{\Gamma \vdash_{F_{<:}} S <: T \quad \Gamma \vdash_{F_{<:}} T <: U}{\Gamma \vdash_{F_{<:}} S <: U} \text{ TRANS}$$

T in this rule is arbitrary and therefore not a subterm of types or the context in the conclusion. Therefore, a definition in normal form should not contain rules like this.

To unblock the problem caused by the Trans rule presented in Section 3.5, it is tempting to consider what is the normal form of $D_{<:}$ subtyping. It turns out that if the Trans rule is just removed from the calculus, then I can readily prove the undecidability of the resulting calculus. This shows that transitivity is the only blocker and further motivates the idea of a normal form.

Theorem 3.11. *$D_{<:}$ subtyping without the* Trans *rule is undecidable.*

Proof. The proof is done by reduction from $F^{-}_{<:}$ using the interpretation functions in Definition 3.5 but without the function case.

The if direction is immediate.

The only if direction is proved by an induction on the derivation tree. Since the Trans rule is removed, the difficulty in Section 3.5 is gone and the rest can be shown by invoking inductive hypothesis or discharged by contradiction. □

Though the idea of normal form looks promising, there is one last fundamental question.

Question 4. *How to come up with a $D_{<:}$ normal form?*

This question is very tricky. To some extent, $F_{<:}$ normal form is actually an intuitive formulation, while as shown in Section 3.1, there are many phenomena of $D_{<:}$ that are unseen in $F_{<:}$. This question poses more detailed sub-questions as follows.

1. How many rules are needed to replace Trans?
2. What are those rules?
3. Should other rules be changed?
4. Will the definition of normal form pose other problems?
5. How do I know whether I am investigating a correct candidate?

The last two sub-questions ask about *the confidence*. Indeed, even if I have formulated a correct normal form by luck, I might not have strong enough confidence to carry out all the proofs because of the large number of case analysis to work through in order to show its equivalence to non-normal form and its undecidability. In other words, to answer this question, it is required to have a thorough understanding of subtyping in $D_{<:}$, and the definition of $D_{<:}$ normal form had better be a result of an intentional construction.

Non-normal form

$$\frac{}{\Gamma \vdash_{F^-_{<:}} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{F^-_{<:}} T <: T}\ \textsc{Refl} \qquad \frac{X <: T \in \Gamma}{\Gamma \vdash_{F^-_{<:}} X <: T}\ \textsc{Tvar}$$

$$\frac{\Gamma \vdash_{F^-_{<:}} S' <: S \quad \Gamma; X <: S' \vdash_{F^-_{<:}} U <: U'}{\Gamma \vdash_{F^-_{<:}} \forall X <: S.U <: \forall X <: S'.U'}\ \textsc{All} \qquad \frac{\Gamma \vdash_{F^-_{<:}} S <: T \quad \Gamma \vdash_{F^-_{<:}} T <: U}{\Gamma \vdash_{F^-_{<:}} S <: U}\ \textsc{Trans}$$

Normal form

$$\frac{}{\Gamma \vdash_{F^{-NF}_{<:}} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{F^{-NF}_{<:}} T <: T}\ \textsc{Refl} \qquad \frac{X <: T \in \Gamma \quad \Gamma \vdash_{F^{-NF}_{<:}} T <: U}{\Gamma \vdash_{F^{-NF}_{<:}} X <: U}\ \textsc{Tvar'}$$

$$\frac{\Gamma \vdash_{F^{-NF}_{<:}} S' <: S \quad \Gamma; X <: S' \vdash_{F^{-NF}_{<:}} U <: U'}{\Gamma \vdash_{F^{-NF}_{<:}} \forall X <: S.U <: \forall X <: S'.U'}\ \textsc{All}$$

Figure 3.2: Non-normal form and normal form of $F^-_{<:}$

3.7 Small-step Subtyping

Motivated by the idea of normal form, in this section, I will introduce a conceptual tool, *small-step subtyping*, which helps to derive $D_{<:}$ normal form. As indicated by its name, one helpful intuition is to draw a parallel between subtyping and operational semantics. In operational semantics, big-step specifies the overall evaluation from terms to values, and produces an end-to-end relation, whereas small-step defines the smallest evaluation steps between terms. A big-step reduction can be viewed as taking some small-step reductions for zero or more times. Considered in this way, what we normally have as a subtyping relation is just like big-step, as it defines an end-to-end relation indicated by the usual ternary predicate. However, this is insufficient if the more fine grained behavior of subtyping becomes important.

For introduction purposes, I first apply the method to $F^-_{<:}$ to explain the equivalence between its non-normal form and normal form. In the next section, I will apply this method to $F^-_{<:>}$, which is a calculus with bad bounds. Then finally I will apply this method to $D_{<:}$

$$\frac{T \neq \top}{\Gamma \vdash_{F^-_{<:}} T \uparrow \top}\ \textsc{Ss-Top} \qquad \frac{\Gamma \vdash_{F^-_{<:}} S_2 \uparrow S_1}{\Gamma \vdash_{F^-_{<:}} \forall X <: S_1.U \uparrow \forall X <: S_2.U}\ \textsc{Ss-All1}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash_{F^-_{<:}} X \uparrow T}\ \textsc{Ss-Var} \qquad \frac{\Gamma; X <: S \vdash_{F^-_{<:}} U_1 \uparrow U_2}{\Gamma \vdash_{F^-_{<:}} \forall X <: S.U_1 \uparrow \forall X <: S.U_2}\ \textsc{Ss-All2}$$

Figure 3.3: Definition of small-step $F^-_{<:}$

in Section 3.9 to derive $D_{<:}$ normal form and conclude its undecidability.

For self-containment, the definitions of $F^-_{<:}$ and its normal form are shown in Figure 3.2, and the differences between both definitions are shaded. The fundamental idea of small-step (subtyping) analysis is to stratify the declarative subtyping rules into two layers, the first representing the smallest steps in subtyping, and the second connecting the steps to form a representation equivalent to the original subtyping definition.

More concretely, the small-step subtyping of $F^-_{<:}$ can be defined as follows.

Definition 3.12. *The small-step subtyping of $F^-_{<:}$ is defined in Figure 3.3.*

In general, small-step subtyping should achieve the following criteria.

1. It should be irreflexive.
2. It should be non-transitive.
3. For each rule, there should be at most one recursive step.

Ss-Top and Ss-Var correspond directly to Top and Tvar' in Figure 2.2. The extra predicate $T \neq \top$ in Ss-Top is to achieve item 1 (irreflexivity). Ss-All1 and Ss-All2 break the All rule into two pieces. Notice that, in Ss-All1, when the parameter type takes a step, it is required that the return type remains the same (as indicated by the identical U being used). This is to comply with item 3. Ss-All2 is analogous.

Clearly small-step subtyping on its own cannot be equivalent to the subtyping rules. For that, we need a second layer.

Definition 3.13. *For a relation R over a set S, namely $R \subseteq S \times S$, its reflexive transitive closure, $R^* \subseteq S \times S$, is constructed inductively as follows.*

1. *Empty, denoted by ϵ, is the reflexivity relation and in R^*, namely $(a, a) \in R^*$ for all $a \in S$.*

2. *Given an instance of R, $(a, b) \in R$, and an instance of R^*, $(b, c) \in R^*$, their concatenation is in R^*, namely $(a, b) \lhd (b, c) \in R^*$.*

Reflexive transitive closure grants the permission to use regular expressions to describe the patterns of sequences of small-steps. I adopt the following convention.

Convention 11. *Given an alphabet Σ,*

1. *$\cdot$ matches any letter in Σ;*

2. *ϵ matches the empty string;*

3. *$*$ means to match the pattern preceding it zero or more times;*

4. *$+$ means to match the pattern preceding it one or more times;*

5. *? means to match the pattern preceding it zero time or once;*

6. *$|$ means to disjoin two patterns;*

7. *() group patterns.*

Non-normal form and small-steps can be shown equivalent.

Theorem 3.12. *$\Gamma \vdash_{F^-_{<:}} S <: U$ is equivalent to $\Gamma \vdash_{F^-_{<:}} S \uparrow^* U$, where $\Gamma \vdash_{F^-_{<:}} S \uparrow^* U$ is a sequence of small-step subtyping given a context Γ.*

Proof. The equivalence is based on the following correspondences. The alphabet of the regular expression is the rules of small-step $F^-_{<:}$.

TOP	$\Leftrightarrow$	SS-TOP?
REFL	$\Leftrightarrow$	ϵ
ALL	$\Leftrightarrow$	SS-ALL1 $*$ SS-ALL2$*$
TVAR	$\Leftrightarrow$	SS-VAR
TRANS	$\Leftrightarrow$	$(\cdot *)(\cdot *)$

□

It is clear that for $F^-_{<:}$ normal form, the following correspondences hold.

$$\begin{array}{lcl}
\textsc{Top} & \Leftrightarrow & \textsc{Ss-Top}? \\
\textsc{Refl} & \Leftrightarrow & \epsilon \\
\textsc{All} & \Leftrightarrow & \textsc{Ss-All1} * \textsc{Ss-All2}* \\
\textsc{Tvar'} & \Leftrightarrow & \textsc{Ss-Var}(\cdot *)
\end{array}$$

Note that though I use regular expression to capture patterns of small-steps, not all sequences of small-steps make sense. For example, Ss-Top Ss-All1 is an invalid sequence, because Ss-Top steps to $\top$ while Ss-All1 can only step from a universal type. When I write $.*$, I mean all *valid* sequences.

The correspondences are stated in a form so that each subpattern corresponds to a premise in its subtyping rule. Reading off the regular expressions of the corresponding rules, it is clear that Trans in non-normal form permits all strings and that there are strings which can be matched by non-normal form but not normal form, e.g. Ss-All2 Ss-All1 Ss-All2. In general, all sequences inexpressible without Trans are captured by the regular expression (Ss-All1 | Ss-All2)$*$.

To summarize, both non-normal form and normal form find their correspondences in small-steps. Small-step subtyping provides a common language to describe non-normal form and normal form, so that their differences surface and can be precisely stated and compared. To understand the equivalence between non-normal form and normal form, the goal is to show that every string s of $\Gamma \vdash_{F^-_{<:}} S \uparrow^* U$ can be converted to another string s' of $\Gamma \vdash_{F^-_{<:}} S \uparrow^* U$ so that only the four patterns from normal form above are used.

3.7.1 An example

The conversion from non-normal form to small-steps is quite trivial. Consider the following derivation showing $\vdash_{F^-_{<:}} (\forall X <: \top . X) <: (\forall X <: (\forall Y <: \top . Y) . \top)$.

$$\frac{\mathcal{D}_1 \quad \mathcal{D}_2}{\vdash_{F^-_{<:}} (\forall X <: \top . X) <: (\forall X <: (\forall Y <: \top . Y) . \top)}\ \textsc{Trans}$$

$\mathcal{D}_1$ is the following derivation.

$$\frac{\vdash_{F^-_{<:}} (\forall Y <: \top . \top) <: \top \quad X <: (\forall Y <: \top . \top) \vdash_{F^-_{<:}} X <: (\forall Z <: \top . \top)}{\vdash_{F^-_{<:}} (\forall X <: \top . X) <: (\forall X <: (\forall Y <: \top . \top) . \forall Z <: \top . \top)}\ \textsc{All}$$

$\mathcal{D}_2$ is the following derivation.

$$\dfrac{\vdash_{F^-_{<:}} (\forall Y <: \top.Y) <: (\forall Y <: \top.\top) \quad X <: (\forall Y <: \top.Y) \vdash_{F^-_{<:}} (\forall Z <: \top.\top) <: \top}{\vdash_{F^-_{<:}} (\forall X <: (\forall Y <: \top.\top).\forall Z <: \top.\top) <: (\forall X <: (\forall Y <: \top.Y).\top)}\ \textsc{All}$$

Conversion to small-steps simply applies the correspondences shown above, particularly $\textsc{All} \Leftrightarrow \textsc{Ss-All1} * \textsc{Ss-All2}*$.

$$\begin{array}{lll}
 & \forall X <: \top.X & \textsc{Ss-All1} \\
\uparrow & \forall X <: \boxed{(\forall Y <: \top.\top)}.X & \lhd\ \textsc{Ss-All2}(\alpha\text{-renamed}) \\
\uparrow & \forall X <: (\forall Y <: \top.\top).\boxed{\forall Z <: \top.\top} & \lhd\ \textsc{Ss-All1} \\
\uparrow & \forall X <: (\forall Y <: \top.\boxed{Y}).\forall Z <: \top.\top & \lhd\ \textsc{Ss-All2} \\
\uparrow & \forall X <: (\forall Y <: \top.Y).\boxed{\top} & \lhd\ \epsilon
\end{array}$$

The changed type in each step is shaded.

Instead of a tree, a subtyping witness in small-steps is represented as a sequence, and one can directly read off how the subtyping witness is constructed from the sequence.

The conversion from small-steps to non-normal form is also trivial, because each concatenation can be mapped to TRANS, since $(\cdot *)(\cdot *)$ is always satisfiable.

3.7.2 Rearranging universal types

Now consider conversion between small-steps and normal form. Due to the correspondences shown previously, it is clear that conversion from normal form to small-steps is achievable. The difficulties lie in the conversion from small-steps to normal form, because only four specific patterns are applicable in normal form.

Consider the small-step sequence presented in Section 3.7.1; it has the pattern SS-ALL1 SS-ALL2 SS-ALL1 SS-ALL2, while in normal form, only SS-ALL1 $*$ SS-ALL2$*$ is allowed. Therefore, in order to show non-normal form is equivalent to normal form, there needs to be a way to convert any (SS-ALL1 | SS-ALL2)$*$ sequence to SS-ALL1 $*$ SS-ALL2$*$. That requires to rearrange the sequences so that all SS-ALL1s appear before SS-ALL2s.

For this particular example, the following rearrangement can be done.

1. The second SS-ALL1 step is pulled up by one step.

2. The first Ss-All2 step needs to be made consistent because the parameter type has become $\forall Y <: \top . Y$.

Namely, the same subtyping can be witnessed by the following small-steps instead.

$$\begin{array}{lll}
 & \forall X <: \top . X & \textsc{Ss-All1} \\
\uparrow & \forall X <: (\forall Y <: \top . \top) . X & \lhd \textsc{Ss-All1} \text{ (pulled up)} \\
\uparrow & \forall X <: (\forall Y <: \top . Y) . X & \lhd \textsc{Ss-All2} \\
\uparrow & \forall X <: (\forall Y <: \top . Y) . \forall Z <: \top . Z & \lhd \textsc{Ss-All2} \text{ (adjustment step)} \\
\uparrow & \forall X <: (\forall Y <: \top . Y) . \forall Z <: \top . \top & \lhd \textsc{Ss-All2} \\
\uparrow & \forall X <: (\forall Y <: \top . Y) . \top & \lhd \epsilon
\end{array}$$

This new sequence matches Ss-All1 $*$ Ss-All2$*$ and therefore can be expressed in $F^{-}_{<:}$ normal form.

Notice that this new small-step sequence takes one more step than the previous sequence, and the extra step is marked as an adjustment step. This step is necessary. In the original sequence, the following Ss-All2 step is taken.

$$\vdash_{F^{-}_{<:}} \forall X <: (\forall Y <: \top . \top) X \uparrow \forall X <: (\forall Y <: \top . \top) \forall Z <: \top . \top$$

Due to the rearrangement in the new sequence, the type to take the step from has become $\forall X <: (\forall Y <: \top . Y) X$, so this adjustment step is to ensure the new sequence is well-defined, as there is no rule to justify the following step.

$$\vdash_{F^{-}_{<:}} \forall X <: (\forall Y <: \top . Y) X \uparrow \forall X <: (\forall Y <: \top . Y) \forall Z <: \top . \top$$

In general, this rearrangement tends to grow the length of the sequence and usually insert many more adjustment steps. The rearrangement itself corresponds to the proof of transitivity in normal form, and the adjustment corresponds to the proof of narrowing, as will be shown below.

3.7.3 Proofs of transitivity and narrowing in normal form

As hinted previously, small-step analysis is introduced to obtain insight into normal form. In the example in Section 3.7.2, I have explained why rearrangement is necessary

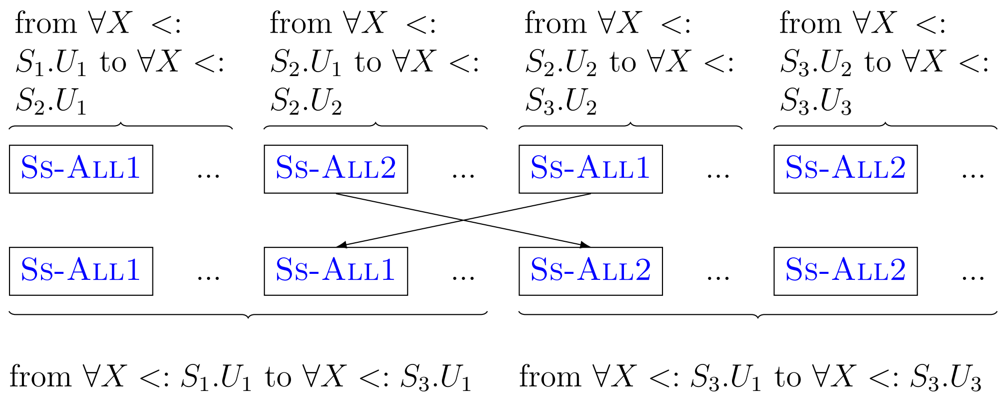
from $\forall X <: S_1.U_1$ to $\forall X <: S_2.U_1$
from $\forall X <: S_2.U_1$ to $\forall X <: S_2.U_2$
from $\forall X <: S_2.U_2$ to $\forall X <: S_3.U_2$
from $\forall X <: S_3.U_2$ to $\forall X <: S_3.U_3$
Ss-All1 ...
Ss-All2 ...
Ss-All1 ...
Ss-All2 ...
Ss-All1 ...
Ss-All1 ...
Ss-All2 ...
Ss-All2 ...
from $\forall X <: S_1.U_1$ to $\forall X <: S_3.U_1$
from $\forall X <: S_3.U_1$ to $\forall X <: S_3.U_3$

Figure 3.4: First row to second row rearranges the universal types.

to express a given subtyping relation in normal form. This rearrangement corresponds to the proof of narrowing in normal form. Narrowing is a desirable property of calculi with subtyping. Informally, it states that an admissible judgment remains admissible if the context becomes more "precise". The property is formally expressed as follows:

Theorem 3.13. *$F^-_{<:}$ normal form satisfies transitivity and narrowing.*

1. *(transitivity) If $\Gamma \vdash_{F^{-NF}_{<:}} S <: T$ and $\Gamma \vdash_{F^{-NF}_{<:}} T <: U$, then $\Gamma \vdash_{F^{-NF}_{<:}} S <: U$.*
2. *(narrowing) If $\Gamma_1; X <: T; \Gamma_2 \vdash_{F^{-NF}_{<:}} S <: U$ and $\Gamma_1 \vdash_{F^{-NF}_{<:}} T' <: T$, then $\Gamma_1; X <: T'; \Gamma_2 \vdash_{F^{-NF}_{<:}} S <: U$.*

Proof. In proving transitivity, narrowing for a strict subterm of T is assumed, and in proving narrowing, transitivity for the same T is assumed. I first prove transitivity.

In transitivity, the proof begins by induction on T and then case analysis on both subtyping derivations. Most of the cases are easy. I will only discuss the case in which both subtyping derivations are constructed by All. Therefore all three S, T and U are universal types and let $S = \forall(X <: S_1).U_1$, $T = \forall(X <: S_2).U_2$ and $U = \forall(X <: S_3).U_3$ for some S_1, U_1, S_2, U_2, S_3 and U_3.

Recall that in normal form, there is the following correspondence: All $\Leftrightarrow$ Ss-All1 $*$ Ss-All2$*$. Therefore, in this case, transitivity between two derivations constructed by All matches the pattern Ss-All1 $*$ Ss-All2 $*$ Ss-All1 $*$ Ss-All2$*$ and was discussed in the example in Section 3.7.2. The goal is to conclude $\Gamma \vdash_{F^{-NF}_{<:}} \forall(X <: S_1).U_1 <: \forall(X <: S_3).U_3$

via the All rule, which corresponds to the pattern Ss-All1 $*$ Ss-All2$*$ and thus requires rearrangement.

In the proof context of this case, there are the following antecedents. This case is diagrammatically presented in Figure 3.4.

1. (The first Ss-All1$*$) $\Gamma \vdash_{F^{-NF}_{<:}} S_2 <: S_1$
2. (The first Ss-All2$*$) $\Gamma; X <: S_2 \vdash_{F^{-NF}_{<:}} U_1 <: U_2$
3. (The second Ss-All1$*$) $\Gamma \vdash_{F^{-NF}_{<:}} S_3 <: S_2$
4. (The second Ss-All2$*$) $\Gamma; X <: S_3 \vdash_{F^{-NF}_{<:}} U_2 <: U_3$

In the diagram of Figure 3.4, the third premise is pulled up as in Section 3.7.2. Since S_2 is a strict subterm, the inductive hypothesis of transitivity is applied to obtain $\Gamma \vdash_{F^{-NF}_{<:}} S_3 <: S_1$. The next target is to obtain $\Gamma; X <: \boxed{S_3} \vdash_{F^{-NF}_{<:}} U_1 <: U_3$. To apply the inductive hypothesis of transitivity again, it is required to have $\Gamma; X <: \boxed{S_3} \vdash_{F^{-NF}_{<:}} U_1 <: U_2$ but the first Ss-All2$*$ only provides $\Gamma; X <: \boxed{S_2} \vdash_{F^{-NF}_{<:}} U_1 <: U_2$. Notice the typing contexts are mismatched. The solution is to apply the adjustment step in Section 3.7.2, which corresponds to narrowing.

Since narrowing applies to strict subterm of $T = \forall(X <: S_2).U_2$, it is applied to obtain $\Gamma; X <: S_3 \vdash_{F^{-NF}_{<:}} U_1 <: U_2$ and subsequently $\Gamma \vdash_{F^{-NF}_{<:}} \forall(X <: S_1).U_1 <: \forall(X <: S_3).U_3$ is concluded by the inductive hypothesis of transitivity followed by the All rule.

The proof of narrowing is significantly easier. It is proved by induction on $\Gamma_1; X <: T; \Gamma_2 \vdash_{F^{-NF}_{<:}} S <: U$. There are places where transitivity of the same T applies. The detailed proof is omitted. □

Theorem 3.14. *$F^{-}_{<:}$ non-normal form and normal form are equivalent.*

$$\Gamma \vdash_{F^{-}_{<:}} S <: U \Leftrightarrow \Gamma \vdash_{F^{-NF}_{<:}} S <: U$$

Proof. The only if direction is immediate. In the if direction, the Trans case is concluded by applying transitivity of normal form. □

In this section, I showed that small-step subtyping provides a common language to describe inference rules in non-normal form and normal form and exposes necessary proof steps in transitivity.

$S, T, U ::=$	**Type**
$\top$	top type
$\bot$	bottom type
X	type variable
$\forall X <: S >: U.T_X$	universal type

Subtyping

$$\frac{}{\Gamma \vdash_{F^-_{<:>}} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{F^-_{<:>}} \bot <: T}\ \textsc{Bot}$$

$$\frac{}{\Gamma \vdash_{F^-_{<:>}} T <: T}\ \textsc{Refl}$$

$$\frac{X >: T \in \Gamma}{\Gamma \vdash_{F^-_{<:>}} T <: X}\ \textsc{Tvar1}$$

$$\frac{\Gamma \vdash_{F^-_{<:>}} S <: T \quad \Gamma \vdash_{F^-_{<:>}} T <: U}{\Gamma \vdash_{F^-_{<:>}} S <: U}\ \textsc{Trans} \qquad \frac{X <: T \in \Gamma}{\Gamma \vdash_{F^-_{<:>}} X <: T}\ \textsc{Tvar2}$$

$$\frac{\Gamma \vdash_{F^-_{<:>}} S_2 <: S_1 \quad \Gamma \vdash_{F^-_{<:>}} U_1 <: U_2 \quad \Gamma; X <: S_2 >: U_2 \vdash_{F^-_{<:>}} T_1 <: T_2}{\Gamma \vdash_{F^-_{<:>}} \forall X <: S_1 >: U_1.T_1 <: \forall X <: S_2 >: U_2.T_2}\ \textsc{All}$$

Figure 3.5: Definition of subtyping in $F^-_{<:>}$

3.8 Undecidability of $F^-_{<:>}$

In Section 3.1, I showed that bad bounds is a pattern which admits subtyping between any two types. In $F^-_{<:}$, however, this pattern does not exist, so I will first investigate another calculus which also has bad bounds, $F^-_{<:>}$, before I investigate $D_{<:}$. Intuitively, $F^-_{<:>}$ extends $F^-_{<:}$ with lower bounded quantification.

Definition 3.14. *$F^-_{<:>}$ is defined in Figure 3.5.*

$F^-_{<:>}$ has the bottom type $\bot$ and the universal types have lower bounded quantification. The context in $F^-_{<:>}$ binds each type variable with two types, its upper bound and its lower bound. The Bot rule says $\bot$ is a subtype of all types. The Tvar1 rule is added for the lower bound lookup. The All rule has an additional predicate, $\Gamma \vdash_{F^-_{<:>}} U_1 <: U_2$, to compare the lower bounds of the type variables. Combining the Tvar1, Tvar2 and Trans rules, we can see that $F^-_{<:>}$ indeed has bad bounds. $F^-_{<:>}$ also satisfies some standard structural properties.

Lemma 3.15. *(weakening) If $\Gamma \vdash_{F^-_{<:>}} S <: U$, then $\Gamma; \Gamma' \vdash_{F^-_{<:>}} S <: U$.*

$$\frac{T \neq \top}{\Gamma \vdash_{F^-_{<:>}} T \uparrow \top}\ \textsc{Ss-Top} \qquad \frac{T \neq \bot}{\Gamma \vdash_{F^-_{<:>}} \bot \uparrow T}\ \textsc{Ss-Bot} \qquad \frac{X >: T \in \Gamma}{\Gamma \vdash_{F^-_{<:>}} T \uparrow X}\ \textsc{Ss-Tvar1}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash_{F^-_{<:>}} X \uparrow T}\ \textsc{Ss-Tvar2} \qquad \frac{\Gamma \vdash_{F^-_{<:>}} S_2 \uparrow S_1}{\Gamma \vdash_{F^-_{<:>}} \forall X <: S_1 >: U.T \uparrow \forall X <: S_2 >: U.T}\ \textsc{Ss-All1}$$

$$\frac{\Gamma \vdash_{F^-_{<:>}} U_1 \uparrow U_2}{\Gamma \vdash_{F^-_{<:>}} \forall X <: S >: U_1.T \uparrow \forall X <: S >: U_2.T}\ \textsc{Ss-All2}$$

$$\frac{\Gamma; X <: S >: U \vdash_{F^-_{<:>}} T_1 \uparrow T_2}{\Gamma \vdash_{F^-_{<:>}} \forall X <: S >: U.T_1 \uparrow \forall X <: S >: U.T_2}\ \textsc{Ss-All3}$$

Figure 3.6: Definition of small-step $F^-_{<:>}$

Lemma 3.16. *(narrowing) If $\Gamma; x <: T_1 >: T_2; \Gamma' \vdash_{F^-_{<:>}} S <: U$ and $\Gamma \vdash_{F^-_{<:>}} T_1' <: T_1$ and $\Gamma \vdash_{F^-_{<:>}} T_2 <: T_2'$, then $\Gamma; x <: T_1' >: T_2'; \Gamma' \vdash_{F^-_{<:>}} S <: U$.*

This definition is clearly not in normal form. Similar to $D_{<:}$, when proving its undecidability by reducing from $F^-_{<:}$, the TRANS rule also gets in the way. For the same motivation as in Section 3.6, a normal form of $F^-_{<:>}$ is needed, and small-step analysis is used to derive it. First, let us recall that small-step subtyping is required to be irreflexive and non-transitive, and to have at most one recursive step in each rule. Acknowledging that, the small-step subtyping of $F^-_{<:>}$ is defined by adapting Figure 3.3.

Definition 3.15. *The small-step subtyping for $F^-_{<:>}$ is defined in Figure 3.6.*

SS-TOP, SS-TVAR2, SS-ALL1 and SS-ALL3 are inherited from $F^-_{<:}$. SS-BOT is added for $\bot$. The premise $T \neq \bot$ is to avoid reflexivity. SS-TVAR1 is added to allow a step to go from a lower bound to its type variable. SS-ALL2 allows the lower bound of a universal type to take a step.

Following the analysis in Section 3.7, small-step subtyping is used as an alphabet to describe what pattern each subtyping rule expresses. Matching the rules one by one, we can see that the following correspondences hold.

Top	$\Leftrightarrow$	Ss-Top?	
Bot	$\Leftrightarrow$	Ss-Bot?	
Refl	$\Leftrightarrow$	ϵ	
All	$\Leftrightarrow$	Ss-All1 $*$ Ss-All2 $*$ Ss-All3$*$	
Tvar1	$\Leftrightarrow$	Ss-Tvar1	
Tvar2	$\Leftrightarrow$	Ss-Tvar2	
Trans	$\Leftrightarrow$	$(\cdot *)(\cdot *)$	(to be dropped)

The next goal is to identify patterns to capture the sequences that are no longer expressible after removal of the Trans rule.

Proposition 3.1. *After the removal of the* Trans *rule, all inexpressible sequences are matched by one of the following patterns:*

1. (Ss-All1 | Ss-All2 | Ss-All3)$*$,
2. $(\cdot *)$Ss-Tvar1,
3. Ss-Tvar2$(\cdot *)$, *and*
4. $(\cdot *)$Ss-Tvar1Ss-Tvar2$(\cdot *)$ *i.e. bad bounds.*

Following the situation in $F^-_{<:}$, (Ss-All1 | Ss-All2 | Ss-All3)$*$ and Ss-Tvar2$(\cdot *)$ are identified. $(\cdot *)$Ss-Tvar1 is dual to Ss-Tvar2$(\cdot *)$. The bad bound patterns naturally follows as the language is designed to have this phenomenon.

The first pattern is a complication of universal types due to addition of lower bounds. In Section 3.7, I showed that patterns like this can be rearranged and in this case, a sequence in that pattern can be rearranged to Ss-All1 $*$ Ss-All2 $*$ Ss-All3$*$, which corresponds to a single All rule. The actual treatment is very similar to the one in $F^-_{<:}$ and will not be discussed here.

In a situation similar to $F^-_{<:}$, both patterns of $(\cdot *)$Ss-Tvar1 and Ss-Tvar2$(\cdot *)$ cannot be expressed by Tvar1 and Tvar2 unmodified. Luckily, the adjustments to make in both rules are as straightforward as follows.

$$\frac{X >: T \in \Gamma \quad \Gamma \vdash_{F^-_{<:>}} T' <: T}{\Gamma \vdash_{F^-_{<:>}} T' <: X}\text{TVAR1'} \qquad \frac{X <: T \in \Gamma \quad \Gamma \vdash_{F^-_{<:>}} T <: T'}{\Gamma \vdash_{F^-_{<:>}} X <: T'}\text{TVAR2'}$$

$$\frac{}{\Gamma \vdash_{F^-_{<:>}} T <: \top}\textsc{Top} \qquad \frac{}{\Gamma \vdash_{F^-_{<:>}} \bot <: T}\textsc{Bot} \qquad \frac{}{\Gamma \vdash_{F^-_{<:>}} T <: T}\textsc{Refl}$$

$$\frac{X <: S >: U \in \Gamma \quad \Gamma \vdash_{F^-_{<:>}} S <: S' \quad \Gamma \vdash_{F^-_{<:>}} U' <: U}{\Gamma \vdash_{F^-_{<:>}} U' <: S'}\text{BB}$$

$$\frac{X >: T \in \Gamma \quad \Gamma \vdash_{F^-_{<:>}} T' <: T}{\Gamma \vdash_{F^-_{<:>}} T' <: X}\textsc{Tvar1'} \qquad \frac{X <: T \in \Gamma \quad \Gamma \vdash_{F^-_{<:>}} T <: T'}{\Gamma \vdash_{F^-_{<:>}} X <: T'}\textsc{Tvar2'}$$

$$\frac{\Gamma \vdash_{F^-_{<:>}} S_2 <: S_1 \quad \Gamma \vdash_{F^-_{<:>}} U_1 <: U_2 \quad \Gamma; X <: S_2 >: U_2 \vdash_{F^-_{<:>}} T_1 <: T_2}{\Gamma \vdash_{F^-_{<:>}} \forall X <: S_1 >: U_1.T_1 <: \forall X <: S_2 >: U_2.T_2}\textsc{All}$$

Figure 3.7: Definition of $F^-_{<:>}$ normal form

The last pattern describes bad bounds. With all the adjustments above, bad bounds can still not expressed. In particular, bad bounds require lookup of both lower and upper bounds in the same rule. That implies that there needs to be a separate rule explicitly describing this pattern. The following rule is a direct transcription of the regular expression and it is the right choice:

$$\frac{X <: S >: U \in \Gamma \quad \Gamma \vdash_{F^-_{<:>}} S <: S' \quad \Gamma \vdash_{F^-_{<:>}} U' <: U}{\Gamma \vdash_{F^-_{<:>}} U' <: S'}\text{BB}$$

Since all four missing patterns are considered, we can be confident that the resulting calculus should admit transitivity and be equivalent to the original definition of $F^-_{<:>}$. The step-by-step analysis provides a promising reason for each modification of rules.

Definition 3.16. *$F^-_{<:>}$ normal form is defined in Figure 3.7.*

The changed rules are shaded. Notice that Trans is gone. It is easy to check the rules one by one and see that the definition is indeed in normal form. A small detail worth noting is that reflexivity is directly admitted by the Refl rule while in $F^-_{<:}$, only reflexivity of type variables is part of the definition. This is not a problem here as reflexivity is not a blocker of the undecidability proof and these two definitions can be easily shown equivalent.

After defining normal form, the next step is to formally verify that the two definitions are indeed equivalent. I shall begin with proving the transitivity of $F^-_{<:>}$ normal form.

Theorem 3.17. *The following hold.*

1. *(transitivity of normal form) If $\Gamma \vdash_{F^-_{<:>}} S <: T$ and $\Gamma \vdash_{F^-_{<:>}} T <: U$, then $\Gamma \vdash_{F^-_{<:>}} S <: U$.*

2. *(narrowing of normal form) If $\Gamma; x <: T_1 >: T_2; \Gamma' \vdash_{F^-_{<:>}} S <: U$ and $\Gamma \vdash_{F^-_{<:>}} T'_1 <: T_1$ and $\Gamma \vdash_{F^-_{<:>}} T_2 <: T'_2$, then $\Gamma; x <: T'_1 >: T'_2; \Gamma' \vdash_{F^-_{<:>}} S <: U$.*

Proof. The proof idea is the same as Theorem 3.13. I will only discuss one additional interesting case here. When the two derivations of transitivity are constructed by TVAR1' and TVAR2' respectively, the following antecedents are in the proof context:

1. $X >: S' \in \Gamma$,

2. $\Gamma \vdash_{F^-_{<:>}} S <: S'$,

3. $X <: U' \in \Gamma$, and

4. $\Gamma \vdash_{F^-_{<:>}} U' <: U$.

Notice that the BB rule applies to obtain $\Gamma \vdash_{F^-_{<:>}} S <: U$. This case shows that the BB rule describes transitivity only for type variables. □

Theorem 3.18. *$F^-_{<:>}$ normal form is equivalent to $F^-_{<:>}$ defined in Figure 3.6.*

Proof. The if direction is immediate. In the only if direction, transitivity of normal form is required, which has been proved above. □

From $F^-_{<:>}$ normal form, it becomes straightforward to show its undecidability, because TRANS is gone.

Theorem 3.19. *$F^-_{<:>}$ is undecidable.*

Proof. This can be shown by reduction from $F^-_{<:}$:

$$\Gamma \vdash_{F^-_{<:}} S <: U \text{ iff } \langle\!\langle \Gamma \rangle\!\rangle \vdash_{F^-_{<:>}} [\![S]\!] <: [\![U]\!]$$

The interpretation functions are defined as follows.

$$\begin{aligned}
[\![\top]\!] &= \top \\
[\![X]\!] &= X \\
[\![\forall X <: S.U]\!] &= \forall X <: [\![S]\!] >: \bot.[\![U]\!]
\end{aligned}$$

$$\begin{aligned}
\langle\!\langle\bullet\rangle\!\rangle &= \bullet \\
\langle\!\langle\Gamma; X <: T\rangle\!\rangle &= \langle\!\langle\Gamma\rangle\!\rangle; X <: \langle\!\langle T\rangle\!\rangle >: \bot
\end{aligned}$$

The if direction is immediate. In the only if direction, most cases are easy to prove by direct reasoning with the inductive hypothesis. Both the TVAR1' and BB cases can be discharged by contradiction using the following argument. We know that if $X >: T \in \langle\!\langle\Gamma\rangle\!\rangle$, then $T = \bot$ by inspecting the context interpretation function. In both rules, there is an antecedent of $\Gamma \vdash_{F^-_{<:>}} [\![T']\!] <: T$, from which we can conclude $\Gamma \vdash_{F^-_{<:>}} [\![T']\!] <: \bot$ and $[\![T']\!] = \bot$. But $\bot$ is not in the image of $[\![\cdot]\!]$ and therefore these rules cannot be applied under the restriction of interpretation functions. □

3.9 Small-step Analysis of $D_{<:}$

I briefly summarize what has been introduced at this point:

1. In Section 3.7, I showed that small-steps of universal types in $F^-_{<:}$ can be rearranged so that the ALL rule is capable of handling all cases of transitivity between universal types.

2. In Section 3.8, I showed that instead of the TRANS rule, $F^-_{<:>}$ normal form introduces the BB rule, which makes bad bounds as an explicit rule, and that is enough to show the admissibility of transitivity.

In this section, I will present small-step analysis of $D_{<:}$. Recall that the definition of $D_{<:}$ is presented in Figure 3.1, with the modification in Lemma 3.1. Small-step subtyping exposes another characteristic, *higher order absorption*, which requires careful consideration of arbitrarily deep type declarations $\{A : S_1..\{A : S_2..\cdots\{A : S_n..T\}\cdots\}\}$ and is often hidden behind transitivity, and only surfaces once transitivity is removed.

Definition 3.17. *The small-step subtyping for $D_{<:}$ is defined in Figure 3.8.*

The rules to be focused on are shaded.

$$\frac{T \neq \top}{\Gamma \vdash_{F^-_{<:>}} T \uparrow \top}\ \textsc{Ss-Top} \qquad \frac{\Gamma \vdash_{F^-_{<:}} S_2 \uparrow S_1}{\Gamma \vdash_{F^-_{<:}} \forall(x : S_1)U \uparrow \forall(x : S_2)U}\ \textsc{Ss-All1}$$

$$\frac{T \neq \bot}{\Gamma \vdash_{F^-_{<:>}} \bot \uparrow T}\ \textsc{Ss-Bot} \qquad \frac{\Gamma; x : S \vdash_{F^-_{<:}} U_1 \uparrow U_2}{\Gamma \vdash_{F^-_{<:}} \forall(x : S)U_1 \uparrow \forall(x <: S)U_2}\ \textsc{Ss-All2}$$

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) \uparrow^* \{A : S..\top\}}{\Gamma \vdash_{D_{<:}} S \uparrow x.A}\ \textsc{Ss-Sel1} \qquad \frac{\Gamma \vdash_{D_{<:}} S_2 \uparrow S_1}{\Gamma \vdash_{D_{<:}} \{A : S_1..U\} \uparrow \{A : S_2..U\}}\ \textsc{Ss-Bnd1}$$

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) \uparrow^* \{A : \bot..U\}}{\Gamma \vdash_{D_{<:}} x.A \uparrow U}\ \textsc{Ss-Sel2} \qquad \frac{\Gamma \vdash_{D_{<:}} U_1 \uparrow U_2}{\Gamma \vdash_{D_{<:}} \{A : S..U_1\} \uparrow \{A : S..U_2\}}\ \textsc{Ss-Bnd2}$$

Figure 3.8: Selected rules of small-step subtyping for $D_{<:}$

Similar to Section 3.7 and Section 3.8, after defining small-step subtyping, it is useful to consider what pattern each subtyping rule corresponds to with small-steps as alphabet. For $D_{<:}$, this is captured by the following correspondences.

Top	$\Leftrightarrow$	Ss-Top?	
Bot	$\Leftrightarrow$	Ss-Bot?	
Refl	$\Leftrightarrow$	ϵ	
All	$\Leftrightarrow$	Ss-All1 $*$ Ss-All2$*$	
Sel1'	$\Leftrightarrow$	Ss-Sel1	
Sel2'	$\Leftrightarrow$	Ss-Sel2	
Bnd	$\Leftrightarrow$	Ss-Bnd1 $*$ Ss-Bnd2	
Trans	$\Leftrightarrow$	$(\cdot*)(\cdot*)$	(to be dropped)

The next step is to consider what patterns can no longer be expressed after dropping the Trans rule. The following list exhausts all such patterns. The situation greatly resembles $F^-_{<:>}$ in Section 3.8.

Proposition 3.2. *After the removal of the* Trans *rule, all inexpressible sequences are matched by one of the following patterns:*

1. (Ss-All1 | Ss-All2)$*$
2. (Ss-Bnd1 | Ss-Bnd2)$*$
3. $(\cdot *)$Ss-Sel1
4. Ss-Sel2$(\cdot *)$
5. $(\cdot *)$Ss-Sel1Ss-Sel2$(\cdot *)$

I claim that the first two patterns are readily expressible by rearrangement. The first pattern can be rearranged in the same way as shown in Section 3.7. The second pattern can be easily rearranged because the Ss-Bnd1 and Ss-Bnd2 steps do not even rely on the context, so their rearrangement is trivial.

In Section 3.7 and Section 3.8, patterns similar to the remaining three patterns are not expressible unless the rules for type variables are adjusted. For example, in $F^-_{<:}$, Tvar needs to be adjusted to Tvar'.

$$\frac{X <: T \in \Gamma}{\Gamma \vdash_{F^-_{<:}} X <: T}\ \textsc{Tvar} \qquad \frac{X <: T \in \Gamma \quad \Gamma \vdash_{F^{-NF}_{<:}} T <: U}{\Gamma \vdash_{F^{-NF}_{<:}} X <: U}\ \textsc{Tvar'}$$

Attempting to adapt the same treatment from $F^-_{<:}$ and $F^-_{<:>}$ naively, one might think that the Sel2' rule should be adjusted to the following form:

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : \bot..U\} \quad \Gamma \vdash_{D_{<:}} U <: U'}{\Gamma \vdash_{D_{<:}} x.A <: U'}\ \textsc{Sel2'-Wrong}$$

However, I claim that this adjustment is *wrong*. In fact, there is no need to adjust either of the Sel1' and Sel2' rules. To understand why this is the case, it is important to first see that with type declarations and path dependent types, subtyping in $D_{<:}$ is *higher-dimensional*.

Higher-dimensional subtyping: Recall that small-step analysis is to convert a subtyping derivation tree into a sequence of small-steps. In $F^-_{<:}$ and $F^-_{<:>}$, this sequence is one dimensional: one can easily see that any subtyping derivation in both calculi can be

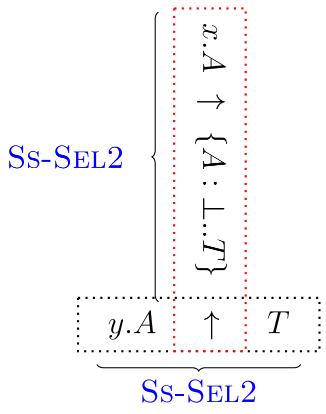

Figure 3.9: A higher-dimensional sequence for $\Gamma \vdash_{D_{<:}} y.A <: T$

expressed entirely by *one* sequence of small-steps. On the other hand, this is no longer true in $D_{<:}$. This can be seen from the Ss-Sel2 step.

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) \uparrow^* \{A : \bot..U\}}{\Gamma \vdash_{D_{<:}} x.A \uparrow U} \text{ Ss-Sel2}$$

Notice that the premise of this step is $\Gamma \vdash_{D_{<:}} \Gamma(x) \uparrow^* \{A : \bot..U\}$. That is, in order to take a step up from $x.A$, there needs to be another complete sequence of small-steps taken first. This lifts the dimension of small-step analysis *up by one*. In general, the small-steps in $\Gamma \vdash_{D_{<:}} \Gamma(x) \uparrow^* \{A : \bot..U\}$ can have higher dimensions, and there is no bound of how many dimensions there need to be. Therefore, the subtyping in $D_{<:}$ is *higher-dimensional*. Let us consider a concrete example:

$$\Gamma = x : \{A : \bot..\{A : \bot..T\}\}; y : x.A$$

$$\cfrac{\cfrac{\epsilon}{\Gamma \vdash_{D_{<:}} x.A \uparrow \{A : \bot..T\}} \text{ Ss-Sel2} \quad \lhd\ \epsilon}{\Gamma \vdash_{D_{<:}} y.A \uparrow T} \text{ Ss-Sel2}$$

The small-step witness is shown in Figure 3.9. The horizontal step (surrounded by black box) is the top level step, which provides a witness for $\Gamma \vdash_{D_{<:}} y.A \uparrow T$. Since $y : x.A$, the Ss-Sel2 step requires a supertype of $x.A$ of the form $\{A : \bot..T\}$. Before this form is reached, $y.A$ cannot take any step. The step from $x.A$ to $\{A : \bot..T\}$ is witnessed by the vertical step, surrounded by the red box. Since x binds to a type declaration, there is no need for any further dimension and the premise of the Ss-Sel2 step of the red box is justified by ϵ.

Imagine that x binds to yet another path type, say $z.A$. Then another sequence of small-steps is needed to find a supertype of $z.A$ that is a type declaration and the dimension goes further up by one. The same situation keeps on going if z binds to another path type. Therefore, a small-step derivation can go up to arbitrarily high dimensions.

However, a derivation cannot have infinite dimensions, as it would require either an infinitely long context or a (finite but) cyclic one. With the well-formedness condition, neither is feasible.

After understanding the higher-dimensional nature of subtyping in $D_{<:}$, let us turn our attention back to the unresolved three patterns left above. Without loss of generality, I focus on the Ss-Sel2$(\cdot *)$ pattern. Consider the case in which Ss-Sel2$(\cdot *)$ witnesses $\Gamma \vdash_{D_{<:}} y.A \uparrow^* T'$ in the context $\Gamma = x : \{A : \bot..\{A : \bot..T\}\}; y : x.A$. Further assume the first Ss-Sel2 step witnesses $\Gamma \vdash_{D_{<:}} y.A \uparrow T$ and $(.*)$ represents some small-step sequence witnessing $\Gamma \vdash_{D_{<:}} T \uparrow^* T'$. In this example, this pattern represents the following steps.

$$\dfrac{\dfrac{\epsilon}{\Gamma \vdash_{D_{<:}} x.A \uparrow \{A : \bot..T\}}\ \textsc{Ss-Sel2} \qquad \lhd\ \epsilon}{\Gamma \vdash_{D_{<:}} y.A \uparrow T}\ \textsc{Ss-Sel2} \qquad \lhd\ \Gamma \vdash_{D_{<:}} T \uparrow^* T'$$

This form admits $\Gamma \vdash_{D_{<:}} x.A <: T'$ and directly motivates the Sel2'-Wrong rule above. However, next I will show a less obvious but equivalent way to admit the same judgment which is more suitable for defining normal form.

Higher-dimensional absorption: In Figure 3.10, I present two diagrams, both of which witness $\Gamma \vdash_{D_{<:}} x.A <: T'$. The diagram on the left shows what has been discussed: it directly represents the pattern Ss-Sel2$(\cdot *)$ and corresponds to the Sel2'-Wrong rule.

The diagram on the right also works but is less obvious. For a given sequence $\Gamma \vdash_{D_{<:}} T \uparrow^* T'$, one can easily show $\Gamma \vdash_{D_{<:}} \{A : \bot..T\} \uparrow^* \{A : \bot..T'\}$ by wrapping each step in the Ss-Bnd2 step. Combined with the original Ss-Sel2 step in the second dimension, which does $\Gamma \vdash_{D_{<:}} x.A \uparrow \{A : \bot..T\}$, $\Gamma \vdash_{D_{<:}} x.A \uparrow^* \{A : \bot..T'\}$ can be shown, and therefore

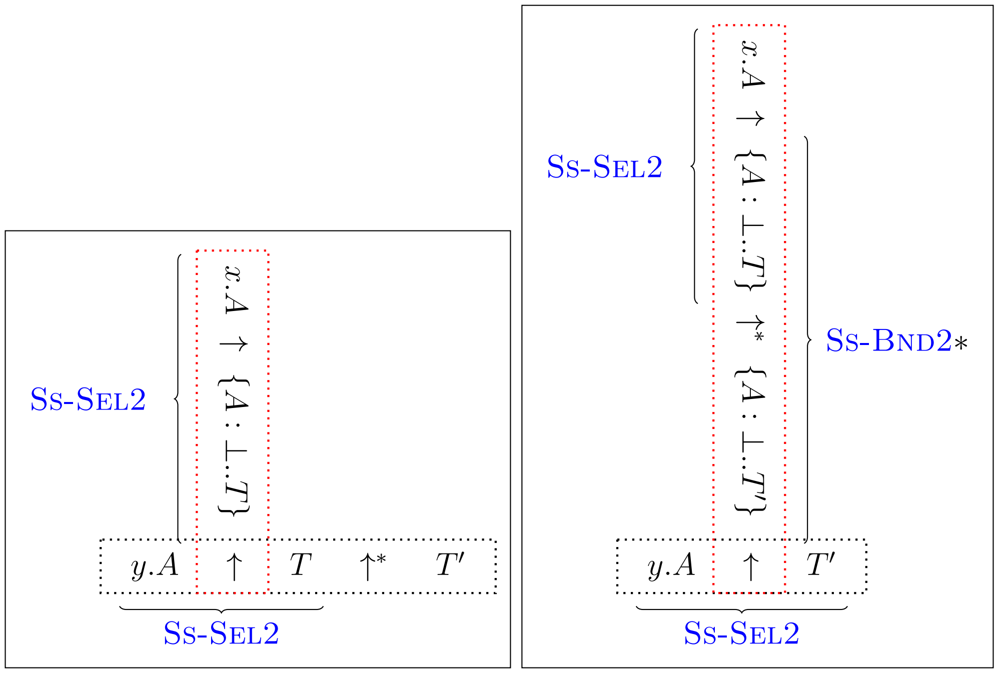
Ss-Sel2
$x.A \uparrow \{A : \bot..T\}$
$y.A$ $\uparrow$ T $\uparrow^*$ T'
Ss-Sel2
Ss-Sel2
$x.A \uparrow \{A : \bot..T\} \uparrow^* \{A : \bot..T'\}$
Ss-Bnd2$*$
$y.A$ $\uparrow$ T'
Ss-Sel2

Figure 3.10: Two sequences to show $\Gamma \vdash_{D_{<:}} y.A \uparrow^* T'$

$\Gamma \vdash_{D_{<:}} y.A \uparrow^* T'$ is justified by one single Ss-Sel2 step. The derivation is the following.

$$\dfrac{\dfrac{\epsilon}{\Gamma \vdash_{D_{<:}} x.A \uparrow \{A : \bot..T\}}\,\textsc{Ss-Sel2} \quad \lhd \quad \dfrac{\Gamma \vdash_{D_{<:}} T \uparrow^* T'}{\Gamma \vdash_{D_{<:}} \{A : \bot..T\} \uparrow^* \{A : \bot..T'\}}\,\textsc{Ss-Bnd2}*}{\Gamma \vdash_{D_{<:}} y.A \uparrow T'}\,\textsc{Ss-Sel2}$$

Effectively, instead of directly applying transitivity connecting $\Gamma \vdash_{D_{<:}} y.A \uparrow T$ and $\Gamma \vdash_{D_{<:}} T \uparrow^* T'$, the steps $\Gamma \vdash_{D_{<:}} T \uparrow^* T'$ are *absorbed* into $\Gamma \vdash_{D_{<:}} \{A : \bot..T\} \uparrow^* \{A : \bot..T'\}$ which resides in a higher dimension. The transition from the diagram on the left to right is called *higher-dimensional absorption*. More formally,

Proposition 3.3. *The following two derivations are equivalent and they both witness* $\Gamma \vdash_{D_{<:}} x.A \uparrow^* U$*.*

$$\dfrac{\Gamma \vdash_{D_{<:}} \Gamma(x) \uparrow^* \{A : \bot..T\}}{\Gamma \vdash_{D_{<:}} x.A \uparrow T}\,\textsc{Ss-Sel2} \quad \lhd \Gamma \vdash_{D_{<:}} T \uparrow^* U$$

$$\frac{}{\Gamma \vdash_{D_{<:}} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{D_{<:}} \bot <: T}\ \textsc{Bot} \qquad \frac{}{\Gamma \vdash_{D_{<:}} T <: T}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash_{D_{<:}} S_2 <: S_1 \quad \Gamma \vdash_{D_{<:}} U_1 <: U_2}{\Gamma \vdash_{D_{<:}} \{A : S_1..U_1\} <: \{A : S_2..U_2\}}\ \textsc{Bnd} \qquad \frac{\begin{array}{c}\Gamma \vdash_{D_{<:}} S_2 <: S_1 \\ \Gamma; x : S_2 \vdash_{D_{<:}} U_1 <: U_2\end{array}}{\Gamma \vdash_{D_{<:}} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2}\ \textsc{All}$$

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : S..\top\}}{\Gamma \vdash_{D_{<:}} S <: x.A}\ \textsc{Sel1'} \qquad \frac{\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : \bot..U\}}{\Gamma \vdash_{D_{<:}} x.A <: U}\ \textsc{Sel2'}$$

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : S..\top\} \quad \Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : \bot..U\}}{\Gamma \vdash_{D_{<:}} S <: U}\ \text{BB}$$

Figure 3.11: Definition of subtyping of $D_{<:}$ normal form

$$\Leftrightarrow$$

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) \uparrow^* \{A : \bot..T\} \lhd \dfrac{\Gamma \vdash_{D_{<:}} T \uparrow^* U}{\Gamma \vdash_{D_{<:}} \{A : \bot..T\} \uparrow^* \{A : \bot..U\}}\ \textsc{Ss-Bnd2}*}{\Gamma \vdash_{D_{<:}} x.A \uparrow T}\ \textsc{Ss-Sel2}$$

This proposition explains why the SEL2'-WRONG rule is entirely unnecessary: the extra predicate not only violates the definition of normal form but also adds no more expressive power to the SEL2' rule! Therefore, SEL2' on its own is powerful enough to express all cases of SS-SEL2$(\cdot*)$.

Dually, the SEL1' rule is capable of expressing $(\cdot*)$SS-SEL1 on its own.

The remaining pattern is the bad bound pattern $(\cdot*)$SS-SEL1SS-SEL2$(\cdot*)$. As seen in Section 3.8, bad bounds require their own explicit rule. $D_{<:}$ normal form is defined by putting all these observations together.

Definition 3.18. *$D_{<:}$ normal form is defined by replacing* TRANS *with* BB*.*

For completeness, the definition of subtyping of $D_{<:}$ normal form is shown in Figure 3.11.

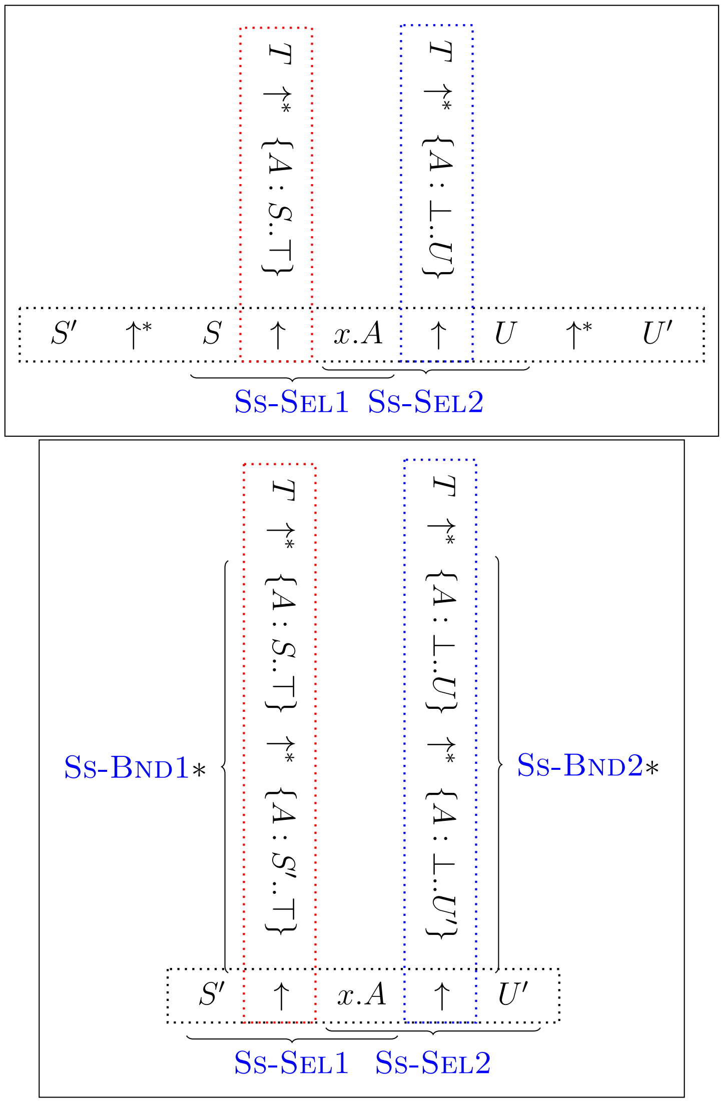
$T \uparrow^* \{A : S..\top\}$
$T \uparrow^* \{A : \bot..U\}$
$S' \uparrow^* S \uparrow x.A \uparrow U \uparrow^* U'$
Ss-Sel1
Ss-Sel2
$T \uparrow^* \{A : S..\top\} \uparrow^* \{A : S'..\top\}$
$T \uparrow^* \{A : \bot..U\} \uparrow^* \{A : \bot..U'\}$
Ss-Bnd1∗
Ss-Bnd2∗
$S' \uparrow x.A \uparrow U'$
Ss-Sel1
Ss-Sel2

Figure 3.12: Effect of higher-dimensional absorption in BB rule, assuming $\Gamma(x) = T$

The BB rule combines both the Sel1' and Sel2' rules. Higher-dimensional absorption might occur in either or both of the type declarations in the premises of the rule, as illustrated in Figure 3.12. In Figure 3.12, due to higher-dimensional absorption, the diagram on top is converted to the diagram at the bottom.

One possible doubt is why BB requires two premises. For example, it might be tempting to adopt the following alternative rule:

$$\frac{\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : S..U\}}{\Gamma \vdash_{D_{<:}} S <: U} \text{ BB-WRONG}$$

This rule will not work, because it is strictly weaker than the BB rule. In general, the witnesses of the premises $\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : S..\top\}$ and $\Gamma \vdash_{D_{<:}} \Gamma(x) <: \{A : \bot..U\}$ can be completely different. This alternative rule disables that possibility and therefore is not able to achieve transitivity.

The next step is to show the equivalence between the two definitions. I shall begin with the proof of transitivity. In $F^-_{<:}$ and $F^-_{<:>}$, the proof requires a mutual induction between transitivity and narrowing, but in $D_{<:}$, the situation is more complicated due to the higher-dimensional nature of subtyping in $D_{<:}$. In the proof I have to reason about arbitrarily nested type declarations of the form $\{A : S_1..\{A : S_2..\cdots\{A : S_n..T\}\cdots\}\}$. This is captured by the following definition.

Definition 3.19. *A type declaration hierarchy is a type defined by another type T and a list of types l inductively as follows.*

$$tdh(T,l) = \begin{cases} T, \text{ if } l \text{ is } nil, \text{ or} \\ \{A : T'..tdh(T,l')\}, \text{ if } l = T' :: l' \end{cases}$$

In correspondence to small-step analysis, the list of types l keeps track of the dimensions. *nil* represents the top level dimension and as the list grows, the dimension increases.

Theorem 3.20. *For any type T and two subtyping derivations in $D_{<:}$ normal form $\mathcal{D}_1$ and $\mathcal{D}_2$, the following hold:*

(1) (transitivity) If $\mathcal{D}_1$ concludes $\Gamma \vdash_{D_{<:}} S <: T$ and $\mathcal{D}_2$ concludes $\Gamma \vdash_{D_{<:}} T <: U$, then $\Gamma \vdash S <: U$.

(2) (narrowing) If $\mathcal{D}_1$ concludes $\Gamma \vdash_{D_{<:}} S <: T$ and $\mathcal{D}_2$ concludes $\Gamma; x : T; \Gamma' \vdash_{D_{<:}} S' <: U'$, then $\Gamma; x : S; \Gamma' \vdash_{D_{<:}} S' <: U'$.

(3) If $\mathcal{D}_1$ concludes $\Gamma \vdash_{D_{<:}} T' <: tdh(\{A : S'..T\}, l)$ and $\mathcal{D}_2$ concludes $\Gamma \vdash_{D_{<:}} T <: U$, then $\Gamma \vdash_{D_{<:}} T' <: tdh(\{A : S'..U\}, l)$.

(4) If $\mathcal{D}_1$ concludes $\Gamma \vdash_{D_{<:}} S <: T$ and $\mathcal{D}_2$ concludes $\Gamma \vdash_{D_{<:}} T' <: tdh(\{A : T..U'\}, l)$, then $\Gamma \vdash_{D_{<:}} T' <: tdh(\{A : S..U'\}, l)$.

Proof. The proof is done by induction on the lexicographical order of the structure of the triple $(T, \mathcal{D}_1, \mathcal{D}_2)$. That is, the inductive hypotheses of the theorem are:

(a) If T^* is a strict syntactic subterm of T, then the theorem holds for T^* and any other two subtyping derivations $\mathcal{D}'_1$ and $\mathcal{D}'_2$.

(b) If $\mathcal{D}^*_1$ is a strict subderivation of $\mathcal{D}_1$, then the theorem holds for the same type T, the subderivation $\mathcal{D}^*_1$ and any subtyping derivation $\mathcal{D}'_2$.

(c) If $\mathcal{D}^*_2$ is a strict subderivation of $\mathcal{D}_2$, then the theorem holds for the same type T, the same derivation $\mathcal{D}_1$ and the subderivation $\mathcal{D}^*_2$.

This form of induction is motivated by the dependencies between the four clauses of the theorem and can be found in other literature [Pfenning, 2000, Theorem 5 (Cut)]. Specifically, (a) addresses that transitivity (1) and narrowing (2) are mutually dependent, but when transitivity uses narrowing, T is replaced with a syntactic subterm T^*. Similarly, (b) addresses that transitivity (1) and (3) are mutually dependent, but in each dependence cycle, $\mathcal{D}_1$ is replaced with a subderivation $\mathcal{D}^*_1$. Finally, (c) addresses that transitivity (1) and (4) are mutually dependent, but in each dependence cycle, $\mathcal{D}_2$ is replaced with a subderivation $\mathcal{D}^*_2$.

Most of the cases in transitivity are straightforward. The ALL-ALL case is proved in a way similar to the proof in Section 3.7 and the SEL1'-SEL2' case is proved in a way similar to the proof in Section 3.8. I will only discuss the SEL2'-*any* case in detail.

SEL2'-*any* case: When $\Gamma \vdash_{D_{<:}} S <: T$ is derived by SEL2', we know $S = y.A$ for some y. The antecedents are:

i. $\Gamma \vdash_{D_{<:}} \Gamma(y) <: \{A : \bot..T\}$, and

ii. $\Gamma \vdash_{D_{<:}} T <: U$.

The intention is to show that $\Gamma \vdash_{D_{<:}} \Gamma(y) <: \{A : \bot..U\}$ holds and hence conclude $\Gamma \vdash_{D_{<:}} y.A <: U$ by SEL2'. To derive this conclusion, we need to apply the induction hypothesis (b) with $\Gamma \vdash_{D_{<:}} \Gamma(y) <: \{A : \bot..T\}$ as the subderivation $\mathcal{D}^*_1$. The induction hypothesis (b) provides the necessary $\Gamma \vdash_{D_{<:}} \Gamma(y) <: \{A : \bot..U\}$ via clause (3), and hence $\Gamma \vdash_{D_{<:}} y.A <: U$. The BB-*any* case can be proved in the same way. The *any*-SEL1' and *any*-BB cases can be proved in a symmetric way, by invoking inductive hypothesis (c) instead of inductive hypothesis (b) in the corresponding places.

Narrowing is proved by induction on $\mathcal{D}_2$ and concluded by inductive hypothesis. There are some cases requiring transitivity, which is provided by the induction hypothesis (c).

Clause (3) of the theorem is proved by case analysis on $\mathcal{D}_1$, the derivation of $\Gamma \vdash_{D_{<:}} T' <: \text{tdh}(\{A : S'..T\}, l)$, and then by an inner induction on the list l. I will discuss two interesting cases.

Bnd-nil case: $\text{tdh}(\{A : S'..T\}, nil) = \{A : S'..T\}$ and $\Gamma \vdash_{D_{<:}} T' <: \text{tdh}(\{A : S'..T\}, nil)$ is constructed by Bnd. From the Bnd rule, we know that $T' = \{A : S_0..U_0\}$ and have the following antecedents:

i. $\Gamma \vdash_{D_{<:}} S' <: S_0$, and

ii. $\Gamma \vdash_{D_{<:}} U_0 <: T$, and

iii. $\Gamma \vdash_{D_{<:}} T <: U$.

Transitivity (1) of antecedents ii. and iii. can be obtained by invoking the induction hypothesis (b) with the antecedent ii. $\Gamma \vdash_{D_{<:}} U_0 <: T$ as $\mathcal{D}_1^*$ and conclude $\Gamma \vdash_{D_{<:}} U_0 <: U$. Then Bnd is applied to $\Gamma \vdash_{D_{<:}} S' <: S_0$ and $\Gamma \vdash_{D_{<:}} U_0 <: U$ to obtain $\Gamma \vdash_{D_{<:}} \{A : S_0..U_0\} <: \{A : S'..U\}$ as required. This case shows the mutual dependence between clause (3) and transitivity (1).

Sel2'-*any* case: In this case, we know that $T' = z.A$ for some z and have the following antecedents:

i. $\Gamma \vdash_{D_{<:}} \Gamma(z) <: \{A : \bot..\text{tdh}(\{A : S'..T\}, l)\}$, and

ii. $\Gamma \vdash_{D_{<:}} T <: U$.

Notice that $\{A : \bot..\text{tdh}(\{A : S'..T\}, l)\}$ can be rewritten as $\text{tdh}(\{A : S'..T\}, (\bot :: l))$, so the induction hypothesis (b) of (3) applies to yield $\Gamma \vdash_{D_{<:}} \Gamma(z) <: \text{tdh}(\{A : S'..U\}, (\bot :: l))$, which can be rewritten as $\Gamma \vdash_{D_{<:}} \Gamma(z) <: \{A : \bot..\text{tdh}(\{A : S'..U\}, l)\}$. Finally, by Sel2', $\Gamma \vdash_{D_{<:}} z.A <: \text{tdh}(\{A : S'..U\}, l)$ as required. Comparing this case with Proposition 3.3, this case corresponds to higher-dimensional absorption by requiring U to be absorbed into $\text{tdh}(\cdot)$. In the inductive hypothesis, the list of types grows to $\bot :: l$ and that corresponds to getting deeper into the dimensions.

Clause (4) of the theorem is dual to clause (3) and is proven in a symmetric way. Instead of the inductive hypothesis (b), clause (4) uses the inductive hypothesis (c). $\square$

We can see that the proof steps are closely related to the phenomena described in small-step analysis. Once transitivity of normal form is proved, I can show the equivalence between the two definitions.

Theorem 3.21. *$D_{<:}$ normal form is equivalent to $D_{<:}$ non-normal form defined in Figure 3.1.*

Proof. The if direction is immediate. In the only if direction, the TRANS case can be discharged by the transitivity of $D_{<:}$ normal form. □

Now I have shown that both definitions of $D_{<:}$ subtyping represent the same relation. However, $D_{<:}$ normal form has the advantage of proving undecidability rather straightforwardly.

Theorem 3.22. *$D_{<:}$ subtyping is undecidable.*

Proof. This is shown by reduction from $F^{-}_{<:}$:

$$\Gamma \vdash_{F^{-}_{<:}} S <: U \text{ iff } \langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} [\![S]\!] <: [\![U]\!]$$

The if direction is immediate.

The only if direction is proved by induction on $D_{<:}$ normal form and relies on Theorem 3.21 to go back to non-normal form. $D_{<:}$ normal form does not have the problem described by Section 3.5 and most of the cases can easily be concluded by using the inductive hypothesis.

The SEL1' and BB cases require the following argument. In both cases, we have the antecedent:

$$\text{for some } x,\ \langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} \langle\!\langle \Gamma \rangle\!\rangle(x) <: \{A : [\![S]\!]..\top\}$$

By inspecting $\langle\!\langle \cdot \rangle\!\rangle$, we know that $\langle\!\langle \Gamma \rangle\!\rangle(x)$ must be $\{A : \bot..T\}$ for some T, and therefore the antecedent becomes

$$\langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} \{A : \bot..T\} <: \{A : [\![S]\!]..\top\}$$

Recall that $\langle\!\langle \Gamma \rangle\!\rangle$ is invertible. By Lemma 3.10, we know

$$\langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_{<:}} [\![S]\!] <: \bot$$

Furthermore, by Lemma 3.9, we know $[\![S]\!] = \bot$. By inspecting $[\![\cdot]\!]$, we see that $\bot$ is not in the image, and therefore both SEL1' and BB cases are discharged by contradiction. □

Finally, Sherlock has caught Merlin's tail.

3.10 Undecidability of Type Assignment of $D_{<:}$

Even at this point, Merlin is still not giving up. He turns his attention to type assignment, and claims that he can decide type assignment in $D_{<:}$. Sherlock then turns to eliminate Merlin's last straw.

The typical idea of the undecidability proof of type assignment is to construct a kind of terms so that any given type can be assigned to one such term if and only if a desired subtyping problem holds. Undecidability of type assignment usually does not receive much attention because in most calculi, finding such a kind of terms is immediate. However, in $D_{<:}$, we have to be more careful. Consider a target subtyping problem $\Gamma \vdash_{D_{<:}} S <: U$ and the following type checking problem:

$$\Gamma \vdash_{D_{<:}} \{A = S\} : \{A : \bot..U\}$$

In order to type check the term $\{A = S\}$, the base case must be Typ-I.

$$\frac{}{\Gamma \vdash_{D_{<:}} \{A = S\} : \{A : S..S\}} \text{ Typ-I}$$

The intention, then, is to turn the typing problem into the following subtyping problem and hope it is equivalent to $\Gamma \vdash_{D_{<:}} S <: U$.

$$\Gamma \vdash_{D_{<:}} \{A : S..S\} <: \{A : \bot..U\}$$

But both subtyping problems are definitely not equivalent! In particular, the following direction is true.

$$\Gamma \vdash_{D_{<:}} S <: U \Rightarrow \Gamma \vdash_{D_{<:}} \{A : S..S\} <: \{A : \bot..U\}$$

However, the other direction is *not*.

$$\Gamma \vdash_{D_{<:}} \{A : S..S\} <: \{A : \bot..U\} \not\Rightarrow \Gamma \vdash_{D_{<:}} S <: U$$

This is because in an unconstrained context in $D_{<:}$, bad bounds can be used to admit subtyping between any two types. If $\Gamma(w) = \{A : \{A : S..S\}..\{A : \bot..U\}\}$ is true, then the type checking problem is admissible even if $\Gamma \vdash_{D_{<:}} S <: U$ is false.

In general, proving the undecidability of type assignment from subtyping requires inversion properties which do not always hold due to bad bounds. The solution, therefore, is to reduce from $F_{<:}^{-}$, which has inversion properties, instead of $D_{<:}$ subtyping.

Theorem 3.23. *For all* Γ*,* S *and* U *in* $F^-_{<:}$*,*

$$\Gamma \vdash_{F^-_{<:}} S <: U \text{ iff } \langle\!\langle\Gamma\rangle\!\rangle \vdash_{D_{<:}} \{A = [\![S]\!]\} : \{A : \bot..[\![U]\!]\}$$

Proof. The if direction is immediate.

Notice the following fact:

$$\langle\!\langle\Gamma\rangle\!\rangle \vdash_{D_{<:}} \{A = [\![S]\!]\} : \{A : \bot..[\![U]\!]\} \Rightarrow \langle\!\langle\Gamma\rangle\!\rangle \vdash_{D_{<:}} \{A : [\![S]\!]..[\![S]\!]\} <: \{A : \bot..[\![U]\!]\}$$

Since $\langle\!\langle\Gamma\rangle\!\rangle$ is invertible, according to Lemma 3.10, the subtyping problem further implies $\langle\!\langle\Gamma\rangle\!\rangle \vdash_{D_{<:}} [\![S]\!] <: [\![U]\!]$, which has been shown equivalent to $\Gamma \vdash_{F^-_{<:}} S <: U$ in Theorem 3.21. □

Theorem 3.24. $D_{<:}$ *type assignment is also undecidable.*

Proof. This follows immediately from the previous theorem. □

Listening to Sherlock's careful elaboration, Merlin could not hide his shocked face, and has to admit that he cannot achieve either of his previous claims. After all, Sherlock's new adventure has come to its end.

3.11 Discussions

3.11.1 $D_{<<:}$

At the beginning of this chapter, I mentioned a list of calculi whose subtyping relations are all undecidable and the list includes $D_{<<:}$.

Definition 3.20. $D_{<<:}$ *is defined in Figure 3.13.*

x, y, z	**Variable**
$S, T, U ::=$	**Type**
$\top$	top type
$\bot$	bottom type
$\{A <: U\}$	type declaration
$x.A$	path type
$\forall(x : S)U_x$	function

Subtyping

$$\frac{}{\Gamma \vdash_{D_{<<:}} T <: \top}\ \textsc{Top}$$

$$\frac{}{\Gamma \vdash_{D_{<<:}} \bot <: T}\ \textsc{Bot}$$

$$\frac{}{\Gamma \vdash_{D_{<<:}} T <: T}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash_{D_{<<:}} U <: U'}{\Gamma \vdash_{D_{<<:}} \{A <: U\} <: \{A <: U'\}}\ \textsc{Bnd}$$

$$\frac{\Gamma \vdash_{D_{<<:}} \Gamma(x) <: \{A <: U\}}{\Gamma \vdash_{D_{<<:}} x.A <: U}\ \textsc{Sel}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{D_{<<:}} S_2 <: S_1 \\ \Gamma; x : S_2 \vdash_{D_{<<:}} U_1 <: U_2\end{array}}{\Gamma \vdash_{D_{<<:}} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2}\ \textsc{All}$$

$$\frac{\Gamma \vdash_{D_{<<:}} S <: T \quad \Gamma \vdash_{D_{<<:}} T <: U}{\Gamma \vdash_{D_{<<:}} S <: U}\ \textsc{Trans}$$

Figure 3.13: Definition of $D_{<<:}$

$D_{<<:}$ is a simplification of $D_{<:}$ the type declarations of which have no lower bounds. Its normal form is defined straightforwardly.

Definition 3.21. *$D_{<<:}$ normal form is defined by removing the* TRANS *rule.*

Reading the rules (ignoring TRANS), it is clear that $D_{<<:}$ normal form satisfies inversion properties. $D_{<<:}$ normal form also satisfies the following properties.

Theorem 3.25. *$D_{<<:}$ normal form is transitive.*

Theorem 3.26. *$D_{<<:}$ non-normal form is equivalent to $D_{<<:}$ normal form.*

Theorem 3.27. *$D_{<<:}$ is undecidable.*

The value of this calculus is that it is on one hand much closer to other *DOT* calculi than $F^{-}_{<:}$, while on the other hand it maintains inversion properties. For this reason, I have the following conjecture.

Conjecture 4. *It is easier to show undecidability of DOT calculi via $D_{<<:}$ than via $F^{-}_{<:}$.*

By easier, I mean with significantly fewer proof steps. With the growth of the feature set of other *DOT* calculi, they become more and more different from $F^{-}_{<:}$. This can potentially add various difficulties to the proof of their undecidability. $D_{<<:}$ being a more complicated calculus than $F^{-}_{<:}$ but still having inversion properties makes it a potentially better candidate for further proofs of undecidability. A detailed investigation is left as future work.

3.11.2 What about *DOT*?

DOT is introduced and briefly discussed in Section 2.5. What we can learn about *DOT* from $D_{<:}$?

The original motivation for decidability analysis of $D_{<:}$ was to obtain a better understanding of how one approaches an undecidability proof of *DOT*. However, after proving $D_{<:}$ undecidable, it is not clear how this can be done for *DOT*.

There are at least the following difficulties to perform a rigorous decidability analysis of *DOT*.

The Trans rule in *DOT*: Notice that *DOT* has transitivity as an explicit rule. This means the undecibility proof of *DOT* is going to fail for the same reason as in Section 3.5. This also implies none of the new undecidability results can be applied to *DOT*.

Following the same thought, the next step is to design normal form for *DOT*, but it is unclear how to start approaching it.

Question 5. *What is the normal form definition for DOT?*

With intersection types, small-step analysis already fails to apply.

Small-step analysis of intersection types: In *DOT*, there is an additional feature called intersection types, which form a (bounded) meet semi-lattice in the subtyping relation. This is achieved by admitting the And-E1, And-E2 and And-I rules.

$$\frac{}{\Gamma \vdash T \wedge U <: T}\ \textsc{And-E1} \qquad \frac{}{\Gamma \vdash T \wedge U <: U}\ \textsc{And-E2} \qquad \frac{\Gamma \vdash T <: S \quad \Gamma \vdash T <: U}{\Gamma \vdash T <: S \wedge U}\ \textsc{And-I}$$

Consider the following proof of commutativity of intersection types.

$$\dfrac{\dfrac{}{\Gamma \vdash S \wedge U <: U}\ \textsc{And-E2} \qquad \dfrac{}{\Gamma \vdash S \wedge U <: S}\ \textsc{And-E1}}{\Gamma \vdash S \wedge U <: U \wedge S}\ \textsc{And-I}$$

This derivation preserves the same information as it begins with, while in small-step subtyping, every step always finds a supertype. So does that mean there should not be any step taken in small-step subtyping? But the type is indeed changed. This shows subtyping with intersection types become hard to interpret in small-step subtyping.

This is a tricky problem but luckily not a fundamental one. In Chapter 5, I will show a normal form calculus with intersection types by directly engineering its definition. The trickiest problem is the recursive types.

Mutual dependency between type assignment and subtyping, and the recursive types: These two problems come in a pair and are more fundamental. Consider the following three rules in *DOT*: Rec-I, Rec-E and Sub.

$$\dfrac{\Gamma \vdash_{DOT} x : T_x}{\Gamma \vdash_{DOT} x : \mu(x : T_x)}\ \textsc{Rec-I} \qquad \dfrac{\Gamma \vdash_{DOT} x : \mu(z : T_z)}{\Gamma \vdash_{DOT} x : T_x}\ \textsc{Rec-E} \qquad \dfrac{\begin{array}{c}\Gamma \vdash_{DOT} t : S \\ \Gamma \vdash_{DOT} S <: U\end{array}}{\Gamma \vdash_{DOT} t : U}\ \textsc{Sub}$$

The Rec-I and Rec-E rules make variable typing more general than subtyping, and the Sub rule behaves just like Trans in subtyping and uses an unknown and arbitrary type S in the rule. Recall that the reason why normal form is desired, is that normal form only uses syntactic subterms from the conclusions of the rules, so that external invariants can be propagated into the derivation tree.

Consider Sherlock and Merlin again, if Merlin complicates a subtyping derivation by using recursive types and the Sub rule, then some unknown type in the Sub rule can be selected to break the inductive invariant. This implies that with just these three rules, undecidability of *DOT* cannot be concluded by induction, orthogonal to the Trans rule.

In Section 8.3, I give a speculation of how a normal form for another calculus, *jDOT* which is a simplification of *DOT*, should look like, but that definition is to be verified and that is left as future work.

3.11.3 Calculi in Normal Forms

In the previous discussion, I have shown how helpful normal forms are. Normal forms are just easier to analyze in general contexts. In the soundness proofs in Amin et al. [2016], Rompf and Amin [2016], Rapoport et al. [2017], there are a number of methods proposed to resolve some difficulties introduced by the TRANS rule, but normal forms just do not have this rule. For example, Amin et al. [2016] uses a technique called "transitivity pushback" to prove the soundness of $D_{<:}$ and DOT. It raises a natural question.

Question 6. *Would soundness proofs of DOT calculi be easier if subtyping is expressed in normal forms?*

The attempt can be first done in $D_{<:}$ to see if it is indeed the case. I speculate the answer is yes and the proof will be significantly easier.

It then motivates another question.

Question 7. *Should calculi be developed / augmented based on normal forms?*

We saw that the undecidability of DOT is not direct from, say, $D_{<:}$ at all, because of the TRANS rule (at least). On the other hand, all the calculi investigated in this chapter have been transformed into normal forms first before connecting with $F_{<:}^{-}$. In the formalization, proofs of properties like narrowing, transitivity and undecidability all follow very similar traits throughout these calculi. This indicates that normal forms are very robust in terms of augmentation. Augmentation based on normal form seems a way to make proofs more modularized and conclusions easier to carry over.

Chapter 4

Algorithmic Typing of $D_{<:}$ Fragments

Decidability analysis in Chapter 3 showed the equivalence between the Trans rule and the BB rule. Namely, transitivity and bad bounds induce the same expressive power in $D_{<:}$. In this chapter, I will start with this observation, and propose two decidable fragments of $D_{<:}$, *kernel* $D_{<:}$ and *strong kernel* $D_{<:}$. As indicated by their names, strong kernel $D_{<:}$ is strictly more expressive than kernel $D_{<:}$. It turns out that these two kernel forms are both decidable, so the original $D_{<:}$ (or *full* $D_{<:}$) is strictly more expressive than both kernel forms.

After introducing both decidable fragments, I will discuss their decision procedures. The decision procedure for kernel $D_{<:}$, *step subtyping*, has already been introduced by Nieto [2017]. This decision procedure is the result of adapting the approaches introduced in Pierce [2002], Pierce and Turner [2000], Pierce [1997], Cardelli and Wegner [1985] to $D_{<:}$. Though this procedure has the advantage of being easy to understand, it also has shortcomings, as discussed in Section 4.7.1.

The decision procedure for strong kernel $D_{<:}$, *stare-at subtyping*, is a result of this thesis which lifts the shortcomings. I then prove the soundness and completeness of both decision procedures w.r.t. their corresponding kernel forms. At this point, the theories of $D_{<:}$ described in the two chapters have beautifully converged.

In the previous chapter, I presented a way to decompose a problem by projecting it to an adversarial game. In this chapter, I will take a different style and consider the problem as a collaborative game.

4.1 Kernel $D_{<:}$

In Chapter 3, I showed that $D_{<:}$ subtyping is undecidable. A natural question to ask is what is a decidable fragment of it? I first introduce one such decidable fragment by applying two adjustments to $D_{<:}$ normal form defined in Figure 3.11.

Consider the subtyping rule for dependent function types, All:

$$\frac{\Gamma \vdash_{D_{<:}} S_2 <: S_1 \quad \Gamma ; x : S_2 \vdash_{D_{<:}} U_1 <: U_2}{\Gamma \vdash_{D_{<:}} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2} \text{ All}$$

As indicated in Chapter 3, this subtyping rule contributes to the undecidability of $D_{<:}$. In Section 2.3, I explained that rules like this are tricky because the second premise effectively behaves like the substitution operation. After acknowledging the substitutive nature, one treatment to make the subtyping problem decidable is to make the parameter types identical in order to limit the power of substitution [Pierce, 2002]:

$$\frac{\Gamma ; x : S \vdash_{D_{<:}} U_1 <: U_2}{\Gamma \vdash_{D_{<:}} \forall(x : S)U_1 <: \forall(x : S)U_2} \text{ All-EqParam}$$

Since the parameter types are syntactically identical, the first premise in All holds by reflexivity and is hence omitted in this new rule. When comparing the return types, x's in U_1 have type S just as it is originally defined, so the power of substitution is bounded and the subtyping problem becomes decidable.

The second adjustment I make is to drop the BB rule. There are three reasons to do that:

1. Bad bounds are generally considered as unexpected consequences of having transitivity;
2. Nieto examined that the Scala compiler does not attempt to implement this rule [Nieto, 2017];
3. I conjecture that the BB rule alone is enough to introduce undecidability.

These adjustments indeed turn $D_{<:}$ decidable and its decision procedure is an existing work, step subtyping, by Nieto [2017]. Following the convention in Pierce [2002], I call the resulting calculus, *kernel* $D_{<:}$, and the original definition of $D_{<:}$, *full* $D_{<:}$. I will first discuss kernel $D_{<:}$ in the rest of this section and start discussing step subtyping in Section 4.2.

$$\frac{}{\Gamma \vdash_{D_{<:K}} T <: \top}\ \textsc{K-Top} \qquad \frac{}{\Gamma \vdash_{D_{<:K}} \bot <: T}\ \textsc{K-Bot} \qquad \frac{}{\Gamma \vdash_{D_{<:K}} T <: T}\ \textsc{K-VRefl}$$

$$\frac{\Gamma \vdash_{D_{<:K}} S_2 <: S_1 \quad \Gamma \vdash_{D_{<:K}} U_1 <: U_2}{\Gamma \vdash_{D_{<:K}} \{A : S_1..U_1\} <: \{A : S_2..U_2\}}\ \textsc{K-Bnd} \qquad \frac{\Gamma; x : S \vdash_{D_{<:K}} U_1 <: U_2}{\Gamma \vdash_{D_{<:K}} \forall(x : S)U_1 <: \forall(x : S)U_2}\ \textsc{K-All}$$

$$\frac{\Gamma \vdash_{D_{<:K}} \Gamma(x) <: \{A : S..\top\}}{\Gamma \vdash_{D_{<:K}} S <: x.A}\ \textsc{K-Sel1} \qquad \frac{\Gamma \vdash_{D_{<:K}} \Gamma(x) <: \{A : \bot..U\}}{\Gamma \vdash_{D_{<:K}} x.A <: U}\ \textsc{K-Sel2}$$

Figure 4.1: Definition of kernel $D_{<:}$

Definition 4.1. *Kernel $D_{<:}$ is defined in Figure 4.1.*

It is easy to show that kernel $D_{<:}$ is sound w.r.t. full $D_{<:}$.

Theorem 4.1. *(soundness of kernel $D_{<:}$ w.r.t. full $D_{<:}$) If $\Gamma \vdash_{D_{<:K}} S <: U$, then $\Gamma \vdash_{D_{<:}} S <: U$.*

Proof. By induction on the derivation of kernel $D_{<:}$. □

On the other hand, since this language is decidable, it is not possible to be complete w.r.t. full $D_{<:}$. For example, the following subtyping is admissible in full $D_{<:}$ by the All rule:

$$x : \{A : \top..\top\} \vdash_{D_{<:}} \forall(y : x.A)\top <: \forall(y : \top)\top$$

However, this judgment is rejected in kernel $D_{<:}$, because $x.A$ and $\top$ are *not* syntactically identical, due to the restriction of the K-All rule.

Moreover, since the BB rule is removed, kernel $D_{<:}$ is not transitive, so judgments which must require bad bounds or transitivity are no longer admissible, e.g.

$$x : \{A : \top..\bot\} \vdash_{D_{<:}} \top <: \bot$$

This conclusion requires transitivity on $x.A$ but it is impossible without either BB or Trans.

$$\frac{}{\Gamma \vdash_{D_{<:S}} \bot <: T}\ \textsc{S-Bot} \qquad \frac{}{\Gamma \vdash_{D_{<:S}} T <: \top}\ \textsc{S-Top} \qquad \frac{}{\Gamma \vdash_{D_{<:S}} x.A <: x.A}\ \textsc{S-VRefl}$$

$$\frac{\Gamma \vdash_{D_{<:S}} x.A \searrow S \quad \Gamma \vdash_{D_{<:S}} T <: S}{\Gamma \vdash_{D_{<:S}} T <: x.A}\ \textsc{S-Sel1} \qquad \frac{\Gamma \vdash_{D_{<:S}} x.A \nearrow U \quad \Gamma \vdash_{D_{<:S}} U <: T}{\Gamma \vdash_{D_{<:S}} x.A <: T}\ \textsc{S-Sel2}$$

$$\frac{\Gamma \vdash_{D_{<:S}} S' <: S \quad \Gamma \vdash_{D_{<:S}} U <: U'}{\Gamma \vdash_{D_{<:S}} \{A : S..U\} <: \{A : S'..U'\}}\ \textsc{S-Bnd} \qquad \frac{\Gamma; x : S \vdash_{D_{<:S}} U <: U'}{\Gamma \vdash_{D_{<:S}} \forall(x : S)U <: \forall(x : S)U'}\ \textsc{S-All}$$

Figure 4.2: Definition of step subtyping operation [Nieto, 2017]

4.2 Step Subtyping

Step subtyping is a partial decision algorithm of $D_{<:}$ introduced by Nieto [2017]. It adapts methods from Pierce [2002], Pierce and Turner [2000], Pierce [1997], Cardelli and Wegner [1985]. I will show that this algorithm is in fact a decision algorithm of kernel $D_{<:}$ in Section 4.5. Note that the rules presented here are not entirely identical to Nieto's presentation. I made some adjustments to help with formal proofs and set up a framework of algorithmic design, but the adjustments are moderate and have no impact on expressiveness.

Definition 4.2. *Step subtyping is defined in Figure 4.2.*

For a judgment $\Gamma \vdash_{D_{<:S}} S <: U$, all three of Γ, S and U are inputs, and an algorithm outputs true if it satisfies this definition.

Nieto has proved a number of properties of this algorithm.

Lemma 4.2. *[Nieto, 2017] Step subtyping is reflexive.*

$$\Gamma \vdash_{D_{<:S}} T <: T$$

Theorem 4.3. *[Nieto, 2017] Step subtyping as an algorithm is sound w.r.t to full $D_{<:}$.*

$$\text{If } \Gamma \vdash_{D_{<:S}} S <: U, \text{ then } \Gamma \vdash_{D_{<:}} S <: U$$

Theorem 4.4. *[Nieto, 2017] Step subtyping as an algorithm terminates.*

Exposure

$$\frac{T \text{ is not a path}}{\Gamma \vdash_{D_{<:}S} T \Uparrow T}\ \textsc{Exp-Stop} \qquad \frac{}{\Gamma \vdash_{D_{<:}S} T \Uparrow \top}\ \textsc{Exp-Top}^*$$

$$\frac{\Gamma_1 \vdash_{D_{<:}S} T \Uparrow \bot}{\Gamma_1 ; x : T ; \Gamma_2 \vdash_{D_{<:}S} x.A \Uparrow \bot}\ \textsc{Exp-Bot}$$

$$\frac{\Gamma_1 \vdash_{D_{<:}S} T \Uparrow \{A : S..U\} \quad \Gamma_1 \vdash_{D_{<:}S} U \Uparrow U'}{\Gamma_1 ; x : T ; \Gamma_2 \vdash_{D_{<:}S} x.A \Uparrow U'}\ \textsc{Exp-Bnd}$$

Upcast/ Downcast

$$\frac{}{\Gamma \vdash_{D_{<:}S} x.A \nearrow \top}\ \textsc{Uc-Top}^* \qquad \frac{}{\Gamma \vdash_{D_{<:}S} x.A \searrow \bot}\ \textsc{Dc-bot}^*$$

$$\frac{\Gamma_1 \vdash_{D_{<:}S} T \Uparrow \bot}{\Gamma_1 ; x : T ; \Gamma_2 \vdash_{D_{<:}S} x.A \nearrow \bot}\ \textsc{Uc-Bot} \qquad \frac{\Gamma_1 \vdash_{D_{<:}S} T \Uparrow \bot}{\Gamma_1 ; x : T ; \Gamma_2 \vdash_{D_{<:}S} x.A \searrow \top}\ \textsc{Dc-Top}$$

$$\frac{\Gamma_1 \vdash_{D_{<:}S} T \Uparrow \{A : S..U\}}{\Gamma_1 ; x : T ; \Gamma_2 \vdash_{D_{<:}S} x.A \nearrow U}\ \textsc{Uc-Bnd} \qquad \frac{\Gamma_1 \vdash_{D_{<:}S} T \Uparrow \{A : S..U\}}{\Gamma_1 ; x : T ; \Gamma_2 \vdash_{D_{<:}S} x.A \searrow S}\ \textsc{Dc-Bnd}$$

Figure 4.3: Definition of **Exposure** and **Upcast / Downcast** operations [Nieto, 2017]

The discussion on the termination proof is postponed to Section 4.7.1.

Compared with kernel $D_{<:}$, the rules of step subtyping are almost identical, except the S-SEL1 and S-SEL2 rules. Both rules are symmetric and rely on other operations, **Exposure**, **Upcast** and **Downcast**, to handle path dependent types. The following convention is adopted in order to discuss the operations.

Convention 12. *When describing algorithmic rules, though most of time syntax is enough to select which rule to apply, there are occasions where two rules apply to the same syntax and a certain execution order is preferred. In this case, I put an asterisk (*) at the end of the name of the rule to indicate that it has the* ***least*** *priority during execution.*

There are situations where the execution order requires more explanations. I will then discuss those situations more carefully in text.

Definition 4.3. *The* ***Exposure****,* ***Upcast*** *and* ***Downcast*** *operations are defined in Figure 4.3.*

1. *An* ***Exposure*** *judgment* $\Gamma \vdash_{D_{<:S}} S \Uparrow U$ *has* Γ *and a type* S *as inputs and a type* U *as output.*

2. *An* ***Upcast*** *judgment* $\Gamma \vdash_{D_{<:S}} x.A \nearrow U$ *has* Γ *and a variable* x *as inputs and a type* U *as output.*

3. *A* ***Downcast*** *judgment is similar to* ***Upcast****.*

Intuitively, **Upcast** and **Downcast** return bounds of a path type $x.A$ in the right direction. The complication comes when x binds to another path type $y.A$, and y could potentially bind to another path type. **Exposure** is designed to resolve this situation by finding a supertype of a path type $x.A$ that is not a path type. Exp-Stop is the base case and a non-path type is found. If the input type is a path $x.A$ and x binds to T in the context, then there are three cases.

1. (Exp-Bot) If T exposes to $\bot$, then the result type is $\bot$.

2. (Exp-Bnd) If T exposes to a type declaration $\{A : S..U\}$, then the result type is the **Exposure** of U. However, U can also be a path type, so a second **Exposure** is called to ensure that the final return type is not a path type.

3. (Exp-Top) If T exposes to any other types, e.g. a dependent function type, then this rule applies. This rule is here to make the operation a total function.

Notice that in the recursive case in Exp-Bot and Exp-Bnd, Γ_1 is used instead of the original context. This is fine because the context is well-formed and T is closed in Γ_1, so the rest of the context is guaranteed not to be used.

Lemma 4.5. *[Nieto, 2017] (****Exposure*** *returns no path) If* $\Gamma \vdash_{D_{<:S}} S \Uparrow U$ *then* U *is not a path type.*

Lemma 4.6. *[Nieto, 2017] (soundness of* ***Exposure****) If* $\Gamma \vdash_{D_{<:S}} S \Uparrow U$*, then* $\Gamma \vdash_{D_{<:}} S <: U$*.*

These two lemmas prove that the operation indeed achieves the intention.

Upcast and **Downcast** operations are symmetric. These two operations provide a shallow wrapper over **Exposure** and serve as entry points of S-Sel1 and S-Sel2. Notice that these two operations are not even recursive. In contrast to **Exposure**, these two operations are allowed to return path types, which is to admit the following case:

$$\text{Let } \Gamma = x : \{A : \bot..\top\}; y : \{A : \bot..x.A\}$$
$$\Gamma \vdash_{D_{<:S}} y.A <: x.A$$

This problem is admitted by the current definition.

$$\dfrac{\Gamma \vdash_{D_{<:S}} y.A \nearrow x.A \quad \dfrac{}{\Gamma \vdash_{D_{<:S}} x.A <: x.A}\ \text{S-VRefl}}{\Gamma \vdash_{D_{<:S}} y.A <: x.A}\ \text{S-Sel1}$$

It is necessary to have **Upcast** return a path type to admit this admissible case.

Applying the soundness of **Exposure**, the soundness of **Upcast** and **Downcast** can also be shown.

Lemma 4.7. *[Nieto, 2017](soundness of **Upcast** and **Downcast**)*

1. *If* $\Gamma \vdash_{D_{<:S}} x.A \nearrow U$*, then* $\Gamma \vdash_{D_{<:}} x.A <: U$.
2. *If* $\Gamma \vdash_{D_{<:S}} x.A \searrow S$*, then* $\Gamma \vdash_{D_{<:}} S <: x.A$.

4.3 Step Typing

Nieto [2017] also introduced a set of algorithmic typing rules. The rules are very straightforward and almost the same as the declarative rules. For completeness, I also define them here.

Definition 4.4. *Step typing is defined in Figure 4.4.*

For a step typing judgment $\Gamma \vdash_{D_{<:S}} t : T$*,* Γ *and* t *are inputs and* T *is the output.*

Compared to the declarative typing rules defined in Figure 3.1, we can see that these two forms are quite similar. The distinctions come from the S-All-E1, S-All-E2 and S-Let rules.

In S-All-E1 and S-All-E2, for an application $x\ y$, the goal is to make sure x has some function type. However, x can bind to some path type, which means it is unclear whether it is a function or not. Therefore, **Exposure** is first used to turn a potential path type into a non-path type. There are two cases.

$$\frac{}{\Gamma \vdash_{D_{<:S}} x : \Gamma(x)}\ \textsc{S-Var} \qquad \frac{\Gamma ; x : S \vdash_{D_{<:S}} t_x : U_x}{\Gamma \vdash_{D_{<:S}} \lambda(x : S)t_x : \forall(x : S)U_x}\ \textsc{S-All-I}$$

$$\frac{\Gamma \vdash_{D_{<:S}} \Gamma(x) \Uparrow \forall(z : S)U_z \quad \Gamma \vdash_{D_{<:S}} \Gamma(y) <: S}{\Gamma \vdash_{D_{<:S}} x\ y : U_y}\ \textsc{S-All-E1}$$

$$\frac{\Gamma \vdash_{D_{<:S}} \Gamma(x) \Uparrow \bot \quad y \in dom(\Gamma)}{\Gamma \vdash_{D_{<:S}} x\ y : \bot}\ \textsc{S-All-E2} \qquad \frac{}{\Gamma \vdash_{D_{<:S}} \{A = T\} : \{A : T..T\}}\ \textsc{S-Typ-I}$$

$$\frac{\Gamma \vdash_{D_{<:S}} t : T \quad \Gamma ; x : T \vdash_{D_{<:S}} u : U' \quad \Gamma ; x : T \vdash_{D_{<:S}} U' \Uparrow_x U}{\Gamma \vdash_{D_{<:S}} \text{let } x = t \text{ in } u : U}\ \textsc{S-Let}$$

Figure 4.4: Definition of step typing for $D_{<:}$ [Nieto, 2017]

$$\frac{}{\Gamma \vdash_{D_{<:S}} \top \Uparrow_x (\Downarrow_x)\top}\ \textsc{PD-Top} \qquad \frac{}{\Gamma \vdash_{D_{<:S}} \bot \Uparrow_x (\Downarrow_x)\bot}\ \textsc{PD-Bot} \qquad \frac{}{\Gamma \vdash_{D_{<:S}} T \Uparrow_x \top}\ \textsc{P-Top}^*$$

$$\frac{}{\Gamma \vdash_{D_{<:S}} T \Downarrow_x \bot}\ \textsc{D-Bot}^* \qquad \frac{x \neq y}{\Gamma \vdash_{D_{<:S}} y.A \Uparrow_x (\Downarrow_x) y.A}\ \textsc{PD-Var}$$

$$\frac{\Gamma \vdash_{D_{<:S}} \Gamma(x) \Uparrow \{A : S..U\}}{\Gamma \vdash_{D_{<:S}} x.A \Uparrow_x U}\ \textsc{P-Sel1} \qquad \frac{\Gamma \vdash_{D_{<:S}} \Gamma(x) \Uparrow \{A : S..U\}}{\Gamma \vdash_{D_{<:S}} x.A \Downarrow_x S}\ \textsc{D-Sel1}$$

$$\frac{\Gamma \vdash_{D_{<:S}} \Gamma(x) \Uparrow \bot}{\Gamma \vdash_{D_{<:S}} x.A \Uparrow_x \bot}\ \textsc{P-Sel2} \qquad \frac{\Gamma \vdash_{D_{<:S}} \Gamma(x) \Uparrow \bot}{\Gamma \vdash_{D_{<:S}} x.A \Downarrow_x \top}\ \textsc{D-Sel2}$$

$$\frac{\Gamma \vdash_{D_{<:S}} S \Downarrow_x (\Uparrow_x) S' \quad \Gamma \vdash_{D_{<:S}} U \Uparrow_x (\Downarrow_x) U'}{\Gamma \vdash_{D_{<:S}} \forall(z : S)U \Uparrow_x (\Downarrow_x) \forall(z : S')U'}\ \textsc{PD-All}$$

$$\frac{\Gamma \vdash_{D_{<:S}} S \Downarrow_x (\Uparrow_x) S' \quad \Gamma \vdash_{D_{<:S}} U \Uparrow_x (\Downarrow_x) U'}{\Gamma \vdash_{D_{<:S}} \{A : S..U\} \Uparrow_x (\Downarrow_x) \{A : S'..U'\}}\ \textsc{PD-Bnd}$$

Figure 4.5: Definitions of **Promotion** and **Demotion** for $D_{<:}$ [Nieto, 2017]

1. S-All-E1 says that x indeed has a function type. Then the next thing to do is to make sure y has the input type S, which is checked by step subtyping, described in the previous section.

2. S-All-E2 says that x is a $\bot$, then at this point, what type y has is irrelevant, as long as it is bound in the context.

S-Let is similar to Let, except that step typing needs to ensure the condition $x \notin fv(U)$ is met. The result of step typing of the body term u is in the context of $\Gamma; x : T$, so there is no guarantee that U' has no x in it. Therefore, an additional operation is needed to find a supertype of U' in which x is guaranteed not free. This is achieved by the **Promotion** operation, indicated by the third premise $\Gamma \vdash_{D_{<:S}} U' \Uparrow_x U$.

Definition 4.5. *The* ***Promotion*** *and* ***Demotion*** *operations are mutually inductively defined in Figure 4.5.*

For a judgment of ***Promotion*** $\Gamma \vdash_{D_{<:S}} S \Uparrow_x U$*,* Γ*,* S *and* x *are the inputs and* U *is the output. A* ***Demotion*** *judgment* $\Gamma \vdash_{D_{<:S}} S \Downarrow_x U$ *is defined similarly.*

Demotion is the dual operation of **Promotion**. Both operations need to be mutually defined, because the parameter types of functions and the lower bounds of type declarations are in contravariant positions, so there needs to be a dual operation of **Promotion** to find a subtype which does not have some variable free.

Since path types are the only types referring to variables, PD-Var, P-Sel1 and P-Sel2 are the base cases for **Promotion** (the situation in **Demotion**, namely D-Sel1 and D-Sel2, can be seen by duality and I omit the discussion in favor of conciseness).

1. (PD-Var) If a path type does not refer to the variable to be removed, then there is no need to do anything with the path type.

2. (P-Sel1) If a path refers to the variable to be removed, then first apply **Exposure** to $\Gamma(x)$, and if the result is a type declaration $\{A : S..U\}$, then U is the result. Notice that in a well-founded context, U is guaranteed to not contain x so there is no need for further recursion.

3. (P-Sel2) If **Exposure** gives $\bot$, then the result type can just be $\bot$.

4. (P-Top) If **Exposure** gives any other type, then the result type is $\top$.

The PD-All rule describes **Promotion** and **Demotion** of function types. In this rule, the recursive call for the body type U is *not* in an extended context, but in the original context Γ. This is correct, because a fresh variable can always be chosen due to α-conversion, and therefore the variable is guaranteed not to be the same as x. Imagine the free variable is y and $y \neq x$. If $y.A$ is used in U, PD-Var is guaranteed to apply, so there is no need to extend the context. A possible doubt could be that now the operations are operating on non-closed types. This is true, but since these non-closed types are always handled by the PD-Var rule, it corresponds to reflexivity in the declarative form and does not prevent the soundness proof from being established.

The following lemma shows that **Promotion** and **Demotion** achieve the intention.

Lemma 4.8. *[Nieto, 2017] (removal of free occurrences) If $\Gamma \vdash_{D_{<:S}} S \Uparrow_x (\Downarrow_x) U$, then $x \notin fv(U)$.*

Lemma 4.9. *[Nieto, 2017] (soundness of **Promotion** and **Demotion**)*

1. *If $\Gamma \vdash_{D_{<:S}} S \Uparrow_x U$ then $\Gamma \vdash_{D_{<:}} S <: U$.*
2. *If $\Gamma \vdash_{D_{<:S}} S \Downarrow_x U$ then $\Gamma \vdash_{D_{<:}} U <: S$.*

From here, we can see that step typing is a sound algorithm.

Theorem 4.10. *[Nieto, 2017] (soundness of step typing) If $\Gamma \vdash_{D_{<:S}} t : T$, then $\Gamma \vdash_{D_{<:}} t : T$.*

Theorem 4.11. *[Nieto, 2017] When viewed as an algorithm, step typing terminates.*

Proof. Every recursive call occurs on a strict sub-term of the input term. □

As we can see, step typing is quite intuitive and easy to understand. Indeed, the main difficulties of the type assignment problem come from the subtyping problem, and the large section of decidability reasoning of subtyping in the previous chapter also agrees with this claim. The same method of step typing can be adapted to larger calculi straightforwardly. In the rest of the chapter, I will focus more on the discussion about subtyping problem.

4.4 A Note on Execution of Step Subtyping

One might have noticed that the rules of step subtyping are not syntax-directed[1]. In particular, when both input types are path types, there are potentially three applicable rules: S-VREFL, S-SEL1 and S-SEL2. In this section, I will discuss and resolve this subtlety. Among the three, it is clear that the S-VREFL rule is the most preferable one: if both path types are syntactically identical, this is the best rule to apply. More subtleties come when two path types are not the same:

$$\Gamma \vdash_{D_{<:S}} x.A <: y.A, \text{where } x \neq y$$

One can apply either the S-SEL1 or S-SEL2 rules in this case.

This shows that step subtyping is indeed not syntax-directed. However, this problem can be resolved by Algorithm 1.

Algorithm 1 Step subtyping when input types are both path types

```
Input: $\Gamma, x.A, y.A$
 1: $S \leftarrow x.A$
 2: $U \leftarrow y.A$
 3: $Paths_1, Paths_2 \leftarrow \emptyset, \emptyset$                 ▷ sets of path types
 4: while $S$ is path type do
 5:     $Paths_1 \leftarrow Paths_1 \cup \{S\}$
 6:     $S \leftarrow$ apply Upcast to $\Gamma$ and $S$
 7: end while
 8: while $U$ is path type do
 9:     $Paths_2 \leftarrow Paths_2 \cup \{U\}$
10:     $U \leftarrow$ apply Downcast to $\Gamma$ and $U$
11: end while                           ▷ Now $S$ and $U$ are not path types
12: if $Paths_1 \cap Paths_2 = \emptyset$ then
13:     Return the result of the recursive call $\Gamma \vdash_{D_{<:S}} S <: U$.
14: else
15:     Subtyping is admitted.
16: end if
```

For ease of presentation, the algorithm is imperative, but rewriting it to be purely functional is straightforward. The algorithm keeps applying **Upcast** to S and **Downcast**

[1]For every input there is at most one rule which can be selected.

to U, until both of them are not path types. During the iterations, two sets of types, $Paths_1$ and $Paths_2$, are used to remember what path types S and U have ever become.

If $Paths_1 \cap Paths_2$ is not empty, then the judgment is concluded by the S-VREFL rule because there exists some order of applications of the S-SEL1 and S-SEL2 rules to turn S and U the same path type. Though the algorithm does not remember what order of applications is, the existence of such order implies that the comparison is admissible by S-VREFL. Otherwise, the judgment cannot be concluded by S-VREFL, so the eventual non-path types S and U become are recursively compared.

To conclude, though the algorithmic rules are not syntax-directed, there is a resolution to the execution order. In later chapters, there are many versions of algorithmic subtyping with the same trait. I will just omit the discussion on which path type related rule to apply first.

4.5 Kernel $D_{<:}$ and Step Subtyping

In this section, I will examine the properties of kernel $D_{<:}$ and its connection with step subtyping. The first easy property to show about kernel $D_{<:}$ is reflexivity.

Lemma 4.12. *Kernel $D_{<:}$ is reflexive.*

$$\Gamma \vdash_{D_{<:}K} T <: T$$

In Chapter 3, we have seen that in terms of expressiveness, transitivity is equivalent to bad bounds. In kernel $D_{<:}$, since the BB rule is removed, one should not expect transitivity to hold in general. However, transitivity on $\top$ and $\bot$ continues to hold.

Lemma 4.13. *If $\Gamma \vdash_{D_{<:}K} \top <: U$, then $\Gamma \vdash_{D_{<:}K} S <: U$.*

Since U can be a path type, the proof needs to be aware of the higher dimensions.

Lemma 4.14. *If $\Gamma \vdash_{D_{<:}K} T <: tdh(\{A : \top..U\}, l)$, then $\Gamma \vdash_{D_{<:}K} T <: tdh(\{A : S..U\}, l)$.*

Proof of Lemma 4.13 and Lemma 4.14. Mutual induction on the subtype derivations. □

The case for $\bot$ can also be proved in a similar way.

Lemma 4.15. *If $\Gamma \vdash_{D_{<:}K} S <: \bot$, then $\Gamma \vdash_{D_{<:}K} S <: U$.*

Now let us consider the soundness of step subtyping w.r.t. kernel $D_{<:}$. In step subtyping, there are two layers of operations: the first layer is step subtyping itself; the second one is **Exposure** which handles path types. To show soundness, the proofs need to go in the opposite direction by first connecting **Exposure** with kernel $D_{<:}$.

Lemma 4.16. *If $\Gamma \vdash_{D_{<:S}} S \Uparrow T$ and $\Gamma \vdash_{D_{<:K}} T <: U$, then $\Gamma \vdash_{D_{<:K}} S <: U$.*

Proof. Induction on the derivation of **Exposure**. □

Theorem 4.17. *(soundness of step subtyping w.r.t. kernel $D_{<:}$)*

If $\Gamma \vdash_{D_{<:S}} S <: U$, then $\Gamma \vdash_{D_{<:K}} S <: U$.

Proof. The proof begins by induction on the derivation of step subtyping. Most of the cases are easy except for the S-SEL1 and S-SEL2 cases; I only discuss the S-SEL2 case. Furthermore, in this case, I do a case split on **Upcast** and only discuss the UC-BND case. That gives a proof context with the following antecedents:

1. $\Gamma \vdash_{D_{<:S}} U' <: U$, and
2. $\Gamma_1 \vdash_{D_{<:S}} T \Uparrow \{A : S..U'\}$, where $\Gamma = \Gamma_1; x : T; \Gamma_2$.

The goal is to conclude $\Gamma \vdash_{D_{<:K}} x.A <: U$ via K-SEL2 which requires the premise $\Gamma \vdash_{D_{<:K}} T <: \{A : \bot..U\}$.

Since Γ is well-formed, T is closed w.r.t. Γ_1 so the second antecedent is equivalent to $\Gamma \vdash_{D_{<:S}} T \Uparrow \{A : S..U'\}$. From the first antecedent and the K-BND rule, $\Gamma \vdash_{D_{<:K}} \{A : S..U'\} <: \{A : \bot..U\}$ is constructed and by Lemma 4.16 $\Gamma \vdash_{D_{<:K}} T <: \{A : \bot..U\}$ is obtained as well the goal. □

Completeness is more difficult to prove than soundness. In the soundness proof, the proof is achieved by two steps representing two layers in step subtyping; the completeness proof, on the other hand, requires to construct the two layers in one go. This is reflected in the proof by the strengthened inductive hypothesis.

Theorem 4.18. *(completeness of step subtyping w.r.t. kernel $D_{<:}$)*

If $\Gamma \vdash_{D_{<:K}} S <: U$, then $\Gamma \vdash_{D_{<:S}} S <: U$.

Proof. The proof requires an intricate strengthening of the induction hypothesis: if $\Gamma \vdash_{D_{<:K}} S <: U$ and this derivation contains n steps, then $\Gamma \vdash_{D_{<:S}} S <: U$, and if U is of the form $\{A : T_1..T_2\}$, then $\Gamma \vdash_{D_{<:S}} S \Uparrow S'$ for some S', and either

1. $S' = \bot$ or
2. $S' = \{A : T_1'..T_2'\}$ for some T_1' and T_2' such that
   (a) $\Gamma \vdash_{D_{<:S}} T_1 <: T_1'$ and
   (b) $\Gamma \vdash_{D_{<:K}} T_2' <: T_2$, and the number of steps in the derivation of $\Gamma \vdash_{D_{<:K}} T_2' <: T_2$ is less than or equal to n.

The proof is by strong induction on n.

To prove $\Gamma \vdash_{D_{<:S}} S <: U$, the non-trivial cases are the K-SEL1 and K-SEL2 cases; we discuss the latter. In this case, $S = x.A$ for some x and the antecedent is $\Gamma \vdash_{D_{<:K}} \Gamma(x) <: \{A : \bot..U\}$. The goal is to show $\Gamma \vdash_{D_{<:S}} x.A <: U$ via the S-SEL2 rule, which requires two premises: $\Gamma \vdash_{D_{<:S}} \Gamma(x) \nearrow U'$ and $\Gamma \vdash_{D_{<:S}} U' <: U$.

On the other hand, the original inductive hypothesis only gives $\Gamma \vdash_{D_{<:S}} \Gamma(x) <: \{A : \bot..U\}$ which cannot be used to establish the premises, so we need the strengthened version. By case analyzing the strengthened inductive hypothesis, both required premises can be established and hence the final goal.

It remains to prove the strengthened part of the inductive hypothesis. The type U can have the specified form $\{A : T_1..T_2\}$ in the conclusions of three rules: K-BOT, K-BND and K-SEL2. Only the K-SEL2 case is interesting. The conclusion of this rule forces $S = y.A$ for some y, and the antecedent is $\Gamma \vdash_{D_{<:K}} \Gamma(y) <: \{A : \bot..\{A : T_1..T_2\}\}$. Applying the induction hypothesis to this antecedent leads to two cases:

1. When $\Gamma \vdash_{D_{<:S}} \Gamma(y) \Uparrow \bot$, the goal $\Gamma \vdash_{D_{<:S}} y.A \Uparrow \bot$ follows by EXP-BOT.
2. Otherwise, for some T_1' and T_2', we obtain additional antecedents:
   (a) $\Gamma \vdash_{D_{<:S}} \Gamma(y) \Uparrow \{A : T_1'..T_2'\}$,
   (b) $\Gamma \vdash_{D_{<:S}} \bot <: T_1'$, and
   (c) $\Gamma \vdash_{D_{<:K}} T_2' <: \{A : T_1..T_2\}$ by a derivation with strictly fewer that n steps.

   The intention is to apply the EXP-BND rule, but this rule requires an **Exposure** on T_2' as well. This can be achieved by applying the inductive hypothesis to the third antecedent again. This yields $\Gamma \vdash_{D_{<:S}} T_2' \Uparrow T_2''$ for some T_2'' and this case is concluded, so we can apply EXP-BND to obtain $\Gamma \vdash_{D_{<:S}} y.A \Uparrow T_2''$, where T_2'' satisfies the properties that the strengthened induction hypothesis requires of S'.

□

Now I have shown that step subtyping induces the same language as kernel $D_{<:}$ defines.

4.6 Strong Kernel $D_{<:}$

Previously, I defined a decidable fragment of $D_{<:}$, kernel $D_{<:}$. Despite its decidability, it comes with obvious disadvantages. Consider the example in Section 4.1:

$$x : \{A : \top..\top\} \vdash_{D_{<:}} \forall(y : x.A)\top <: \forall(y : \top)\top$$

This judgment is admissible in full $D_{<:}$ but not in kernel $D_{<:}$ because $x.A$ and $\top$ are not syntactically identical. However, notice that x binds to the type declaration $\{A : \top..\top\}$, which is quite special: the upper bound and lower bound are the same. This special status of the type declaration makes $x.A$ an *alias* of $\top$. Forcing parameter types to be syntactically the same disables any usage of aliasing, while in Scala (or generally in many other languages) aliasing of types is a desirable feature.

The inspiration for the new calculus comes from writing out the typing context *twice* in a subtyping derivation. For example, recall that the existing All rule is:

$$\frac{\Gamma \vdash_{D_{<:}} S' <: S \quad \Gamma; x : S' \vdash_{D_{<:}} U_x <: U'_x}{\Gamma \vdash_{D_{<:}} \forall(x : S)U <: \forall(x : S')U'} \textsc{All}$$

Let us write the contexts twice for this rule:

$$\frac{\Gamma \vdash_{D_{<:}} S' <: S \dashv \Gamma \quad \Gamma; x : S' \vdash_{D_{<:}} U_x <: U'_x \dashv \Gamma; x : S'}{\Gamma \vdash_{D_{<:}} \forall(x : S)U <: \forall(x : S')U' \dashv \Gamma} \textsc{All-TwoContexts}$$

Now do the same for the kernel version too:

$$\frac{\Gamma; x : S \vdash_{D_{<:}} U_x <: U'_x \dashv \Gamma; x : S}{\Gamma \vdash_{D_{<:}} \forall(x : S)U <: \forall(x : S)U' \dashv \Gamma} \textsc{K-All-TwoContexts}$$

So far, both copies of the context have been the same, so the second copy is redundant. However, comparing these two rules for a moment, we can see some potential for improvement. In the premise comparing $U_x <: U'_x$, the only difference is the primes on S in the typing contexts: the first rule uses S' on both sides, while the second rule uses S on both sides. Since U_x comes from a universal type where x has type S, and U'_x from one where x has type S', what if we took the middle ground between the two rules, and added S to the left context and S' to the right context?

$$\frac{\Gamma \vdash_{D_{<:}} S' <: S \dashv \Gamma \quad \Gamma; x : S \vdash_{D_{<:}} U_x <: U'_x \dashv \Gamma; x : S'}{\Gamma \vdash_{D_{<:}} \forall(x : S)U <: \forall(x : S')U' \dashv \Gamma} \textsc{All-AsymmetricContexts}$$

$$\frac{}{\Gamma_1 \vdash_{D_{<:SK}} T <: \top \dashv \Gamma_2}\ \textsc{Sk-Top} \qquad \frac{}{\Gamma_1 \vdash_{D_{<:SK}} \bot <: T \dashv \Gamma_2}\ \textsc{Sk-Bot}$$

$$\frac{}{\Gamma_1 \vdash_{D_{<:SK}} x.A <: x.A \dashv \Gamma_2}\ \textsc{Sk-VRefl}$$

$$\frac{\Gamma_1 \vdash_{D_{<:SK}} S_1 >: S_2 \dashv \Gamma_2 \quad \Gamma_1 \vdash_{D_{<:SK}} U_1 <: U_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{D_{<:SK}} \{A : S_1..U_1\} <: \{A : S_2..U_2\} \dashv \Gamma_2}\ \textsc{Sk-Bnd}$$

$$\frac{\Gamma_1 \vdash_{D_{<:SK}} S_1 >: S_2 \dashv \Gamma_2 \quad \Gamma_1; x : S_1 \vdash_{D_{<:SK}} U_1 <: U_2 \dashv \Gamma_2; x : S_2}{\Gamma_1 \vdash_{D_{<:SK}} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2 \dashv \Gamma_2}\ \textsc{Sk-All}$$

$$\frac{\Gamma_1 \vdash_{D_{<:SK}} \{A : S..\top\} >: \Gamma_2(x) \dashv \Gamma_2}{\Gamma_1 \vdash_{D_{<:SK}} S <: x.A \dashv \Gamma_2}\ \textsc{Sk-Sel1}$$

$$\frac{\Gamma_1 \vdash_{D_{<:SK}} \Gamma_1(x) <: \{A : \bot..U\} \dashv \Gamma_2}{\Gamma_1 \vdash_{D_{<:SK}} x.A <: U \dashv \Gamma_2}\ \textsc{Sk-Sel2}$$

Figure 4.6: Definition of strong kernel $D_{<:}$

The new rule enables the contexts to be different, so it justifies maintaining both contexts. But how will a calculus with this hybrid rule behave? Will it be strictly in between the decidable kernel $D_{<:}$ and the undecidable full $D_{<:}$ in expressiveness (assuming the BB rule is also removed)? Will it be decidable? I will show that the answer to both questions is yes. The new hybrid rule allows comparison of function types with *different* parameter types, and the return types are compared in two different contexts. In particular, it admits the example judgement with the aliased parameter types in the beginning of this section.

Definition 4.6. *Strong kernel $D_{<:}$ is defined in Figure 4.6.*

This calculus is called strong kernel $D_{<:}$, because it is more expressive than kernel $D_{<:}$. To show this, let us first examine that kernel $D_{<:}$ is sound w.r.t. strong kernel $D_{<:}$. The proof will require the following lemma.

Lemma 4.19. *(reflexivity of strong kernel $D_{<:}$)*

$$\Gamma_1 \vdash_{D_{<:SK}} T <: T \dashv \Gamma_2$$

$$\frac{}{\bullet \subseteq_{<:} \bullet}\ \textsc{Ope-Nil} \qquad \frac{\Gamma \subseteq_{<:} \Gamma'}{\Gamma; x : T \subseteq_{<:} \Gamma'}\ \textsc{Ope-Drop} \qquad \frac{\Gamma \subseteq_{<:} \Gamma' \quad \Gamma \vdash_{D_{<:}} S <: U}{\Gamma; x : S \subseteq_{<:} \Gamma'; x : U}\ \textsc{Ope-Keep}$$

Figure 4.7: Definition of $OPE_{<:}$

Theorem 4.20. *If* $\Gamma \vdash_{D_{<:K}} S <: U$*, then* $\Gamma \vdash_{D_{<:SK}} S <: U \dashv \Gamma$.

Proof. The proof is done by induction on the derivation of kernel $D_{<:}$. In the K-All case, reflexivity of strong kernel is needed to admit the comparison of parameter types, which is a provable property of strong kernel. $\square$

To show strong kernel is strictly stronger, it is sufficient to show that there is a subtyping relation admissible in strong kernel but not in kernel. If we start with the same context on both sides, the aliasing example in the beginning of the section is admitted by the following derivation:

let $\Gamma = x : \{A : \top..\top\}$

$$\dfrac{\dfrac{\text{reflexivity}}{\Gamma \vdash_{D_{<:SK}} x.A >: \top \dashv \Gamma}\ \textsc{Sk-Sel1} \qquad \dfrac{}{\Gamma; y : x.A \vdash_{D_{<:SK}} \top <: \top \dashv \Gamma; y : \top}\ \textsc{Sk-Top}}{\Gamma \vdash_{D_{<:SK}} \forall(y : x.A)\top <: \forall(y : \top)\top \dashv \Gamma}\ \textsc{Sk-All}$$

The next step is to show that strong kernel is sound w.r.t. full $D_{<:}$.

Theorem 4.21. *If* $\Gamma \vdash_{D_{<:SK}} S <: U \dashv \Gamma$ *then* $\Gamma \vdash_{D_{<:}} S <: U$.

An additional relation between two contexts is needed before proving this theorem.

Definition 4.7. *The order preserving sub-environment relation between two contexts, or* $OPE_{<:}$*, is defined in Figure 4.7.*

Intuitively, if $\Gamma \subseteq_{<:} \Gamma'$, then Γ is a more “informative” context than Γ'. $OPE_{<:}$ is a combination of the narrowing and weakening properties. The following properties of $OPE_{<:}$ to confirm this intuition.

Lemma 4.22. $OPE_{<:}$ *is reflexive.*

$$\Gamma \subseteq_{<:} \Gamma$$

Lemma 4.23. *$OPE_{<:}$ is transitive.*

$$\textit{If } \Gamma_1 \subseteq_{<:} \Gamma_2 \textit{ and } \Gamma_2 \subseteq_{<:} \Gamma_3, \textit{ then } \Gamma_1 \subseteq_{<:} \Gamma_3.$$

Theorem 4.24. *(respectfulness) Full $D_{<:}$ subtyping is preserved by $OPE_{<:}$.*

If $\Gamma \subseteq_{<:} \Gamma'$ and $\Gamma' \vdash_{D_{<:}} S <: U$, then $\Gamma \vdash_{D_{<:}} S <: U$.

$OPE_{<:}$ is used to express the inductive invariant required in Theorem 4.21. This is expressed by the following theorem:

Theorem 4.25. *If $\Gamma_1 \vdash_{D_{<:}SK} S <: U \dashv \Gamma_2$, $\Gamma \subseteq_{<:} \Gamma_1$ and $\Gamma \subseteq_{<:} \Gamma_2$, then $\Gamma \vdash_{D_{<:}} S <: U$.*

Proof. By induction on the strong kernel subtyping derivation. □

Then Theorem 4.21 follows from reflexivity of $OPE_{<:}$.

The Sk-All rule is strictly weaker than its full counterpart, the All rule. This can be seen from the following judgment:

$$\dfrac{\dfrac{\text{straightforward}}{\vdash_{D_{<:}} \{A : \bot..\bot\} <: \{A : \bot..\top\}}\ \textsc{Bnd} \qquad \dfrac{\text{straightforward}}{x : \{A : \bot..\bot\} \vdash_{D_{<:}} x.A <: \bot}\ \textsc{Sel2}}{\vdash_{D_{<:}} \forall(x : \{A : \bot..\top\})x.A <: \forall(x : \{A : \bot..\bot\})\bot}\ \textsc{All}$$

This judgment is rejected by strong kernel $D_{<:}$ because the comparison of the returned types relies on the parameter type to the right of <:, which is not possible in strong kernel $D_{<:}$. Notice that this example uses aliasing information from the right parameter type (i.e. that $x.A$ is an alias of $\bot$) to reason about the left return type (i.e. that $x.A$ is a subtype of $\bot$), which is something that strong kernel $D_{<:}$ cannot do.

4.7 Stare-at Subtyping

In this section, I will introduce the novel algorithmic subtyping rules, *stare-at subtyping*, which are a decision procedure for strong kernel $D_{<:}$. The name of this set of rules comes from its notation $\Gamma_1 \gg S <: U \ll \Gamma_2$. If we see $\gg$ and $\ll$ as eyes and <: as a nose, then the notation looks like a face, and the two eyes are staring at the nose.

In this section, I will begin with pointing out two shortcomings of step subtyping. Then I will define stare-at subtyping and informally explain why both shortcomings are lifted. I will leave the formal reasoning to Section 4.8.

4.7.1 Limitations of Step Subtyping

Stare-at subtyping is an improvement on step subtyping. Overall, stare-at subtyping resolves two shortcomings of step subtyping:

1. The parameter types of functions are required to be (syntactically) identical.
2. The termination proof as an algorithm is driven by semantics, instead of syntax.

The first shortcoming is inherited from kernel $D_{<:}$ as step subtyping is a complete algorithm. The second shortcoming requires some more explanations.

The second shortcoming is somewhat more technical, but is not negligible in formal proofs, as a semantic termination proof requires a significant amount of proof engineering. The following is the definition of the measure function for step subtyping described in Nieto [2017].

Definition 4.8. *The measure of a type T in a context Γ for step subtyping is computed as follows.*

$$
\begin{aligned}
W_\Gamma(\top) &= 1 \\
W_\Gamma(\bot) &= 1 \\
W_\Gamma(\{A : S..U\}) &= 1 + \max(W_\Gamma(S), W_\Gamma(U)) \\
W_\Gamma(x.A) &= 1 + W_{\Gamma_1}(T), && \textit{where } \Gamma = \Gamma_1; x : T; \Gamma_2 \\
W_\Gamma(\forall(x : S)U) &= 1 + W_{\Gamma;x:S}(U)
\end{aligned}
$$

The first three cases are syntactic: they are either base cases or simply recur down to the smaller syntactic subterms. However, the path type and function type cases are semantic.

1. In the path type case, the measure of $x.A$ relies on the measure of T, the type x binds to. This is not syntactic, because the measure depends on the meaning of x based on the context.
2. The function type case is also semantic, as the result depends on an extended context. The measure of $x.A$ is computed only in this extended context, so the measure of a function type always depends on the context.

$$\frac{}{\Gamma_1 \gg T <: \top \ll \Gamma_2}\ \text{SA-Top} \qquad \frac{}{\Gamma_1 \gg \bot <: T \ll \Gamma_2}\ \text{SA-Bot}$$

$$\frac{\begin{array}{c}\Gamma_2 \vdash_{D_{<:}S} x.A \searrow T \dashv \Gamma_2' \\ \Gamma_1 \gg S <: T \ll \Gamma_2'\end{array}}{\Gamma_1 \gg S <: x.A \ll \Gamma_2}\ \text{SA-Sel1} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash_{D_{<:}S} x.A \nearrow T \dashv \Gamma_1' \\ \Gamma_1' \gg T <: U \ll \Gamma_2\end{array}}{\Gamma_1 \gg x.A <: U \ll \Gamma_2}\ \text{SA-Sel2}$$

$$\frac{}{\Gamma_1 \gg x.A <: x.A \ll \Gamma_2}\ \text{SA-VRefl} \qquad \frac{\Gamma_1 \gg S >: S' \ll \Gamma_2 \quad \Gamma_1 \gg U <: U' \ll \Gamma_2}{\Gamma_1 \gg \{A : S..U\} <: \{A : S'..U'\} \ll \Gamma_2}\ \text{SA-Bnd}$$

$$\frac{\Gamma_1 \gg S >: S' \ll \Gamma_2 \quad \Gamma_1; x : S \gg U <: U' \ll \Gamma_2; x : S'}{\Gamma_1 \gg \forall(x : S)U <: \forall(x : S')U' \ll \Gamma_2}\ \text{SA-All}$$

Figure 4.8: Definition of stare-at subtyping

Moreover, this measure function is *partial*: in the path type case, x does not have to be bound in the context in general. This situation tends to be omitted in informal analysis, due to the basic assumption of well-formedness of contexts, but making this assumption explicit in formal proof assistants requires substantial engineering effort.

$D_{<:}$ is still a small calculus, so this engineering effort might not consume much time, but as the feature set increases in later chapters, this problem could quickly scale up and become very hard to manage. Stare-at subtyping provides a minimal solution to this situation.

4.7.2 Definition

Since judgments in strong kernel $D_{<:}$ have two contexts, as its intended decision algorithm, stare-at subtyping also operates on two contexts.

Definition 4.9. *Stare-at subtyping is defined in Figure 4.8.*

For a stare-at subtyping judgment $\Gamma_1 \gg S <: U \ll \Gamma_2$, all four places are the inputs, and it outputs true if inputs satisfy its definition.

One can think of stare-at subtyping as a collaborative game between two players, Alice and Bob. Alice is responsible for the context and type to the left of $<:$ or $>:$, while Bob is responsible for the other side. In particular, Alice and Bob are completely independent and do not need to see the contexts or types held by their collaborator. Most of the rules are just straightforward extensions of the corresponding rules of step subtyping with two contexts, except for three cases: SA-ALL, SA-SEL1 and SA-SEL2.

Notice that in SA-ALL, the two parameter types are allowed to be different. The first premise compares them in the original context. In the second premise, the parameter types are added to the contexts held by Alice and Bob respectively. This obviously has overcome the limitation of identical parameter types described previously.

For the path types, three cases are needed for the very same reasons as in step subtyping: SA-SEL1, SA-SEL2 and SA-VREFL. What is different in SA-SEL1 and SA-SEL2 are the **Upcast** and **Downcast** operations. They now return not only a type but also a context. These two operations depend on another operation similar to **Exposure**: **Revealing**.

Definition 4.10. *The definitions of **Revealing**, **Upcast** and **Downcast** operations are in Figure 4.9.*

1. *A **Revealing** judgment $\Gamma \vdash_{D_{<:S}} S \Uparrow U \dashv \Gamma'$ has Γ and a type S as inputs and Γ' and a type U as outputs.*

2. *An **Upcast** judgment $\Gamma \vdash_{D_{<:S}} x.A \nearrow U \dashv \Gamma'$ has Γ and a variable x as inputs and Γ' and a type U as outputs.*

3. *A **Downcast** judgment $\Gamma \vdash_{D_{<:S}} x.A \searrow U \dashv \Gamma'$ is defined similarly as **Upcast**.*

Notice that in the SA-SEL1 and SA-SEL2 rules, Alice and Bob use the contexts returned from **Upcast** and **Downcast** in the recursive calls.

The **Revealing** operation not only finds a non-path supertype of a path type as **Exposure** does, but also returns a prefix of the input context which contains all free variables of the output type. Returning this additional prefix context turns the termination proof purely syntactic. The observation is that in the termination measure of step subtyping (Definition 4.8), it is ultimately the context lookup in the path type case that makes the proof semantic. However, the very same context lookup has occurred once in the derivation of the **Exposure** operation, so the measure function effectively *simulates* **Exposure** one more time. This simulation is necessary in **Exposure** because the context is shared between Alice and Bob in step subtyping. But with stare-at subtyping, Alice and Bob work on two independent contexts. This allows them to more actively manipulate their own contexts, so that the contexts can directly reflect the results of context look-ups. Let me analyze the rules of **Revealing** one by one.

Revealing

$$\frac{T \text{ is not a path}}{\Gamma \vdash_{D_{<:S}} T \Uparrow T \dashv \Gamma} \text{ Rv-Stop} \qquad \frac{}{\Gamma \vdash_{D_{<:S}} T \Uparrow \top \dashv \bullet} \text{ Rv-Top*}$$

$$\frac{\Gamma_1 \vdash_{D_{<:S}} T \Uparrow \bot \dashv \Gamma_1'}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \Uparrow \bot \dashv \bullet} \text{ Rv-Bot} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash_{D_{<:S}} T \Uparrow \{A : S..U\} \dashv \Gamma_1' \\ \Gamma_1' \vdash_{D_{<:S}} U \Uparrow U' \dashv \Gamma_1''\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \Uparrow U' \dashv \Gamma_1''} \text{ Rv-Bnd}$$

Upcast/ Downcast

$$\frac{}{\Gamma \vdash_{D_{<:S}} x.A \nearrow \top \dashv \bullet} \text{ U-Top*} \qquad \frac{}{\Gamma \vdash_{D_{<:S}} x.A \searrow \bot \dashv \bullet} \text{ D-bot*}$$

$$\frac{\Gamma_1 \vdash_{D_{<:S}} T \Uparrow \bot \dashv \Gamma_1'}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \nearrow \bot \dashv \bullet} \text{ U-Bot} \qquad \frac{\Gamma_1 \vdash_{D_{<:S}} T \Uparrow \bot \dashv \Gamma_1'}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \searrow \top \dashv \bullet} \text{ D-Top}$$

$$\frac{\Gamma_1 \vdash_{D_{<:S}} T \Uparrow \{A : S..U\} \dashv \Gamma_1'}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \nearrow U \dashv \Gamma_1'} \text{ U-Bnd} \qquad \frac{\Gamma_1 \vdash_{D_{<:S}} T \Uparrow \{A : S..U\} \dashv \Gamma_1'}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \searrow S \dashv \Gamma_1'} \text{ D-Bnd}$$

Figure 4.9: Definition of **Revealing** and new definitions of **Upcast** and **Downcast**

1. (Rv-Stop) The same as **Exposure**, **Revealing** attempts to find a non-path supertype, so this case is the base case and has achieved the goal.
2. (Rv-Bot) If the input type is a path type, then apply **Revealing** to T. If **Revealing** returns $\bot$, the result is just $\bot$. Additionally, the returned context is empty. This is because $\bot$ has no free variables so there is no need to return any content in the context.
3. (Rv-Bnd) Similar to **Exposure**, when the **Revealing** of T returns a type declaration $\{A : S..U\}$, the second recursive call is there to make sure U becomes a non-path type.
4. (Rv-Top) If **Revealing** of T returns any other type, then this rule is here to make sure the overall function is total.

Upcast and **Downcast** are in the same positions as in step subtyping, and follow the same adjustments and serve as shallow wrappers over **Revealing**.

Reflexivity of stare-at subtyping can already be established based on its definition.

Theorem 4.26. *(reflexivity) Stare-at subtyping is reflexive.*

$$\Gamma_1 \gg T <: T \ll \Gamma_2$$

Proof. Perform induction on T. □

4.8 Properties of Operations in Stare-at Subtyping

Context manipulation is new in the subtyping decision procedure, so let us examine some properties of the operations to help understand how the operations behave. First, let us quantify the relation between the input and output contexts of **Revealing**.

Definition 4.11. *A context Γ' is a prefix of Γ, if there is another context Γ'', and $\Gamma = \Gamma'; \Gamma''$.*

Lemma 4.27. *(**Revealing** gives prefixes) If $\Gamma \vdash_{D_{<:S}} S \Uparrow U \dashv \Gamma'$, then Γ' is a prefix of Γ.*

From the rules, we can even be sure that contexts get strictly shorter if S is a path type.

In the previous section, I mentioned that **Revealing** is intended to replace **Exposure**, so it should achieve the properties of **Exposure**.

Lemma 4.28. *(**Revealing** returns no path) If $\Gamma \vdash_{D_{<:S}} S \Uparrow U \dashv \Gamma'$, then U is not a path.*

Lemma 4.29. *(soundness of **Revealing**) If $\Gamma \vdash_{D_{<:S}} S \Uparrow U \dashv \Gamma'$, then $\Gamma \vdash_{D_{<:}} S <: U$.*

Notice that the subtyping relation is witnessed in the original context Γ, not in the returned context Γ'. In general, the returned context is not expected to witness the subtyping relation.

Since the context is shrunk, the well-formedness condition is not as obvious as it used to be.

Lemma 4.30. *If Γ' is a prefix of Γ and Γ is well-formed, then Γ' is well-formed.*

Lemma 4.31. *(well-formedness condition) If $\Gamma \vdash_{D_{<:S}} S \Uparrow U \dashv \Gamma'$, Γ is well-formed and $fv(S) \subseteq dom(\Gamma)$, then $fv(U) \subseteq dom(\Gamma')$.*

Combining both lemmas, we can see that the well-formedness condition is recovered.

Similar lemmas can be established for **Upcast** and **Downcast**. They are much easier to prove because **Upcast** and **Downcast** are not even recursive.

Lemma 4.32. *The following hold.*

1. *If* $\Gamma \vdash_{D_{<:}S} x.A \nearrow (\searrow) T \dashv \Gamma'$, *then* Γ' *is a prefix of* Γ.
2. *If* $\Gamma \vdash_{D_{<:}S} x.A \nearrow T \dashv \Gamma'$, *then* $\Gamma \vdash_{D_{<:}} x.A <: T$.
3. *If* $\Gamma \vdash_{D_{<:}S} x.A \searrow T \dashv \Gamma'$, *then* $\Gamma \vdash_{D_{<:}} T <: x.A$.
4. *If* $\Gamma \vdash_{D_{<:}S} x.A \nearrow (\searrow) T \dashv \Gamma'$, Γ *is well-formed and* $x \in dom(\Gamma)$, *then* $fv(T) \subseteq dom(\Gamma')$.

To prove stare-at subtyping is sound, I use the $OPE_{<:}$ relation defined in Section 4.6. $OPE_{<:}$ is needed in order to describe the situations in SA-Sel1 and SA-Sel2 where the weakening part is needed, and the situations in SA-All where the narrowing part is needed. With this relation, the soundness of stare-at subtyping can be shown.

Theorem 4.33. *(soundness of stare-at subtyping) If* $\Gamma_1 \gg S <: U \ll \Gamma_2$, $\Gamma \subseteq_{<:} \Gamma_1$ *and* $\Gamma \subseteq_{<:} \Gamma_2$, *then* $\Gamma \vdash_{D_{<:}} S <: U$.

If Alice and Bob start with the same context, then stare-at subtyping is a partial decision algorithm for subtyping.

Theorem 4.34. *If* $\Gamma \gg S <: U \ll \Gamma$, *then* $\Gamma \vdash_{D_{<:}} S <: U$.

Next, I will show the termination proofs of the operations. It is easy to see that **Revealing** terminates.

Lemma 4.35. ***Revealing*** *terminates as an algorithm.*

Proof. The measure is the length of the input context. □

Then I will show that the termination argument of stare-at subtyping is purely syntactic. Consider the measure of the syntactic size of types and contexts.

Definition 4.12. *The measure M of types and contexts is defined by the following equations.*

$$
\begin{aligned}
M(\top) &= 1\\
M(\bot) &= 1\\
M(x.A) &= 2\\
M(\forall(x:S)U) &= 1 + M(S) + M(U)\\
M(\{A : S..U\}) &= 1 + M(S) + M(U)
\end{aligned}
$$

$$M(\Gamma) = \sum_{x:T\in\Gamma} M(T)$$

Comparing this definition with Definition 4.8, M simply measures the syntactic sizes of types and contexts and is a purely syntactic measure. Moreover, M is clearly a total function.

The following lemma can be shown easily by induction.

Lemma 4.36. *If $\Gamma \vdash_{D_{<:S}} S \Uparrow U \dashv \Gamma'$, then $M(\Gamma) + M(S) \geq M(\Gamma') + M(U)$.*

If $\Gamma \vdash_{D_{<:S}} x.A \nearrow (\searrow) U \dashv \Gamma'$, then $M(\Gamma) + M(x.A) > M(\Gamma') + M(U)$.

In the SA-Sel1 and SA-Sel2 cases, this lemma is used to argue size decrement, and the other rules clearly decrease by structure. Therefore, the termination of stare-at subtyping can also be concluded.

Theorem 4.37. *Stare-at subtyping terminates as an algorithm.*

Proof. For a judgment $\Gamma_1 \gg S <: U \ll \Gamma_2$, the overall measure is $M(\Gamma_1)+M(S)+M(U)+M(\Gamma_2)$. Use the previous lemma to show that **Upcast** and **Downcast** strictly decrease the measure. □

Consider the solution again from Alice and Bob's point of view. What happens effectively is Alice and Bob treat their own copies of contexts as scrap papers. Since their modifications to the contexts are independent, there is no need for any coordination between them, as long as the types and contexts they hold remain well-formed. The operations are designed to maintain $OPE_{<:}$ as an invariant, which grants Alice and Bob freedom to actively modify their contexts and eventually leads to the soundness and termination proofs.

4.9 Strong Kernel $D_{<:}$ and Stare-at Subtyping

The soundness and completeness proofs greatly resemble the situations in kernel $D_{<:}$. There are subtleties due to the two contexts but it would be clear enough if I just list the statements of the lemmas and theorems.

Lemma 4.38. *If* $\Gamma_1 \vdash_{D_{<:}SK} \top <: U \dashv \Gamma_2$*, then* $\Gamma_1 \vdash_{D_{<:}SK} S <: U \dashv \Gamma_2$*.*

Lemma 4.39. *If* $\Gamma_1 \vdash_{D_{<:}SK} S <: \bot \dashv \Gamma_2$*, then* $\Gamma_1 \vdash_{D_{<:}SK} S <: U \dashv \Gamma_2$*.*

Lemma 4.40. *If* $\Gamma_1 \vdash_{D_{<:}S} S \Uparrow T \dashv \Gamma_1'$ *and* $\Gamma_1 \vdash_{D_{<:}SK} T <: U \dashv \Gamma_2$*, then* $\Gamma_1 \vdash_{D_{<:}SK} S <: U \dashv \Gamma_2$*.*

In this last lemma, Γ_1' is not used in the rest of the statement. The intuition is that strong kernel $D_{<:}$ does not shrink the context like stare-at subtyping does, so the returned context from **Revealing** is simply dropped.

Theorem 4.41. *(soundness of stare-at subtyping w.r.t. strong kernel $D_{<:}$)*

If $\Gamma_1 \gg S <: U \ll \Gamma_2$*, then* $\Gamma_1 \vdash_{D_{<:}SK} S <: U \dashv \Gamma_2$*.*

Proof. By induction on the derivation of stare-at subtyping. The proof is very similar to the version of step subtyping. □

The completeness proof is slightly trickier than the one of step subtyping, because in the SA-Sel1 and SA-Sel2 cases, Alice and Bob work on prefix contexts in the recursive calls. In contrast, in the Sk-Sel1 and Sk-Sel2 rules of strong kernel $D_{<:}$, the subtyping judgements in the premises use the same full contexts as the conclusions. Therefore, we need to make sure that working on smaller contexts will not change the outcome.

Theorem 4.42. *(strengthening of stare-at subtyping) If* $\Gamma_1; \Gamma_1'; \Gamma_1'' \gg S <: U \ll \Gamma_2; \Gamma_2'; \Gamma_2''$*,* $fv(S) \subseteq dom(\Gamma_1; \Gamma_1'')$ *and* $fv(U) \subseteq dom(\Gamma_2; \Gamma_2'')$*, then* $\Gamma_1; \Gamma_1'' \gg S <: U \ll \Gamma_2; \Gamma_2''$*.*

Proof. Perform an induction on the derivation of stare-at subtyping. □

Now we can prove the completeness of stare-at subtyping.

Theorem 4.43. *(completeness of stare-at subtyping w.r.t. strong kernel $D_{<:}$) If* $\Gamma_1 \vdash_{D_{<:}SK} S <: U \dashv \Gamma_2$*, then* $\Gamma_1 \gg S <: U \ll \Gamma_2$*.*

Proof. The proof is similar to the one of Theorem 4.18. We also need to strengthen the inductive hypothesis to the following: if $\Gamma_1 \vdash_{D_{<:SK}} S <: U \dashv \Gamma_2$ and this derivation contains n steps, then $\Gamma_1 \gg S <: U \ll \Gamma_2$ and if U is of the form $\{A : T_1..T_2\}$, then $\Gamma_1 \vdash_{D_{<:S}} S \Uparrow S' \dashv \Gamma'_1$, and either

1. $S' = \bot$, or
2. $S' = \{A : T'_1..T'_2\}$ for some T'_1 and T'_2, such that
   (a) $\Gamma_1 \gg T_1 <: T'_1 \ll \Gamma_2$ and
   (b) $\Gamma_1 \vdash_{D_{<:SK}} T'_2 <: T_2 \dashv \Gamma_2$, and the number of steps in this derivation is less than or equal to n.

The Sk-Sel1 and Sk-Sel2 cases are trickier. After invoking the inductive hypothesis, due to the well-formedness condition of **Upcast** and **Downcast**, we apply Theorem 4.42 so that the eventual derivation of stare-at subtyping works in prefix contexts. □

Therefore, we conclude that strong kernel and stare-at subtyping are the same language.

Completeness may seem somewhat surprising since stare-at subtyping truncates the typing contexts in the SA-Sel1 and SA-Sel2 cases while strong kernel subtyping does not. Technically, the truncation is justified by Theorem 4.42. Intuitively, since the prefixes of the typing contexts cover the free variables of the relevant type, they do include all of the information necessary to reason about that type. However, it is important to keep in mind that this is possible only because we have removed the BB rule. In a calculus with the BB rule, it is possible that $\Gamma \vdash S <: U$ is false in some context Γ that binds all free variables of S and U, but that if we further extend the context with some Γ', that can make $\Gamma; \Gamma' \vdash S <: U$ true due to new subtyping relationships introduced in Γ' by the BB rule.

4.10 Discussions

4.10.1 Convergence of Theories

Up to the soundness and completeness of step subtyping and stare-at subtyping proved w.r.t. their corresponding kernel calculi, the theories related to $D_{<:}$ have beautifully converged. Tracing back to the root of all critical observations, $D_{<:}$ normal form makes most of the contributions. So far this thesis has shown that calculi in normal form give many advantages.

Reviewing the algorithmic rules and the kernel calculi, it is not difficult to notice a certain pattern exists. I call the transformation from a normal form calculus to its kernel form *kernelization*, and the transformation to strong kernel form *strong kernelization*. Their commonalities are

1. removal of bad bounds, and
2. identical parameter types in kernel or two contexts in strong kernel.

Clearly, strong kernelization is a stronger approach, and therefore, in the rest of the thesis, I will only consider the strong kernel form. In Chapter 5, I will show that a calculus extending $D_{<:}$ with intersection types also has a strong kernel form. However, with recursive types, how the same can be done becomes unclear.

4.10.2 Properties of Kernel Calculi

Previously, one difficulty of $D_{<:}$ was from the bad bounds. Due to bad bounds, a number of helpful properties are no longer true. For example, in $D_{<:}$, one might want the following property:

$$\text{If } \Gamma \vdash_{D_{<:}} \forall(x : S)U <: \forall(x : S')U', \text{then } \Gamma \vdash_{D_{<:}} S' <: S \text{ and } \Gamma; x : S' \vdash_{D_{<:}} U <: U'.$$

This property is a inversion property, and is *not true* in general.

Due to bad bounds, this very helpful property is rejected. As shown in Chapter 3, having bad bounds is equivalent to having transitivity. This implies two desired properties cannot coexist in $D_{<:}$, and transitivity makes the calculus harder to reason about. On the other hand, in (strong) kernel $D_{<:}$, the inversion property *holds*; one can verify it by simply looking at the rules and K-All/Sk-All is the only possible constructor. Therefore, study of kernel calculi is desirable, and their decidability further motivates that.

There are other aspects to study for the kernel calculi.

For example, in $F_{<:}$, one can define a *minimal* type T that satisfies a certain predicate P, if the following property holds:

$$\text{If } \Gamma \vdash_{F_{<:}} S <: T \text{ and } P(S), \text{ then } S = T$$

This concept is very useful. For example, Pierce [2002, 1997] Pierce showed that **Exposure** finds the minimal non-variable supertype of type variables in both $F_{<:}$ and $F_{<:}$ with $\bot$. In $D_{<:}$, however, it is challenging to informally state how a minimal type should be defined and this kind of property is unlikely to be true due to bad bounds.

For example, consider the following context:

$$x : \{A : \bot..\forall(z : \top)\top\}$$

It is tempting to say that the minimal supertype of $x.A$ that is not a path is $\forall(z : \top)\top$. However, this can be changed if the context is extended. Consider the following context extended with y:

$$x : \{A : \bot..\forall(z : \top)\top\}; y : \{A : x.A..\forall(z : \top)\bot\}$$

Now the situation is far less clear. In fact, within this context, the minimal supertype of $x.A$ that is not a path becomes $\forall(z : \top)\bot$.

The subtyping relations between these types can be represented by the following diagram.

$$\begin{tikzcd}
\bot \arrow[r] & x.A \arrow[rrr] \arrow[dr] & & & \forall(z : \top)\top \\
& & y.A \arrow[r] & \forall(z : \top)\bot \arrow[ur] &
\end{tikzcd}$$

In the diagram, $S \longrightarrow U$ means $\Gamma \vdash_{D_{<:}} S <: U$. As indicated by the second row, a new subtyping relation is injected into the diagram after the extension.

In (strong) kernel $D_{<:}$, the situation is slightly different.

$$\begin{tikzcd}
\bot \arrow[r] \arrow[drrr, bend right] & x.A \arrow[rrr, dotted] \arrow[dr, dotted] & & & \forall(z : \top)\top \\
& & y.A \arrow[r, dotted] & \forall(z : \top)\bot \arrow[ur] &
\end{tikzcd}$$

The above diagram represents the situation in (strong) kernel $D_{<:}$. Though all arrows denote subtyping relation, there are now different kinds of arrows. In particular, an arrow can be solid or half solid. Subtyping is not transitive through nodes connected by the dotted ends of arrows. For example, $y.A$ is one such kind of nodes:

$$\begin{tikzcd}
x.A \arrow[r, dotted] & y.A \arrow[r, dotted] & \forall(z : \top)\bot
\end{tikzcd}$$

In the diagram, $\bot \longrightarrow x.A$, $\bot \longrightarrow \forall(z : \top)\bot$ and $\forall(z : \top)\bot \longrightarrow \forall(z : \top)\top$ are solid because these relations are defined to hold. All other arrows are half solid because they are imposed by path dependent types. A lower bound has a dotted end into the path type and a upper bound has a dotted end out of the path type. Rejecting transitivity through a node connected by dotted ends is a diagrammatic representation of rejecting the BB rule.

For example, $\Gamma \vdash_{D<:} y.A <: \forall(z : \top)\bot <: \forall(z : \top)\top$ is justified by following the arrows starting from $y.A$, then to $\forall(z : \top)\bot$ and finally to $\forall(z : \top)\top$. On the other hand, $\Gamma \vdash_{D<:} x.A <: \forall(z : \top)\bot$ is *rejected*, because it would require to go through the arrow from $x.A$ to $y.A$ but subtyping is not transitive on $y.A$.

In this diagram, one can now claim that $\forall(z : \top)\top$ remains $x.A$'s minimal non-path supertype, even when y is in the context; $\forall(z : \top)\bot$ is not a supertype of $x.A$. Therefore, subtyping in kernel calculi is more stable relative to extensions of contexts.

If minimal typing is defined, then there will be more work to do on the operations. For example, I expect that **Exposure** and **Revealing** return the minimal non-path supertypes of the input types.

The requirement of distinguishing subtyping introduced by path types or not seems to indicate a certain stratification of the subtyping relation, which can be a very interesting direction to explore. This, however, will not be explored in this thesis and is left as future work.

Chapter 5

$D_\wedge$

The previous two chapters have introduced and analyzed various aspects of $D_{<:}$. However, $D_{<:}$ is the simplest calculus of the family and not the eventual goal. In this chapter, I will introduce an extension of $D_{<:}$, $D_\wedge$. $D_\wedge$ extends $D_{<:}$ with data fields and intersection types. To generalize the calculus, the numbers of type member labels and data field member labels in $D_\wedge$ are countably infinite. Compared to DOT, $D_\wedge$ only has recursive types missing.

In this chapter, I will start with the $D_\wedge$ normal form: as shown previously, non-normal form is much harder to analyze while not more expressive. I will prove that $D_\wedge$ sutyping is undecidable. I will show strong kernel $D_\wedge$ as an extension of strong kernel $D_{<:}$. I will then adapt stare-at subtyping in $D_{<:}$ to $D_\wedge$ and show that the algorithm is sound and complete in strong kernel $D_\wedge$. The situation in $D_\wedge$ greatly resembles the one in $D_{<:}$ and the theories of $D_\wedge$ also converge very nicely.

In this chapter, I only study the subtyping relation, and type assignment will be discussed in greater details in Chapter 6 and Chapter 7.

5.1 Definition of $D_\wedge$

Definition 5.1. *The definition of $D_\wedge$ is shown in Figure 5.1.*

Recall that in $D_{<:}$, there is no data field member and there is only one type member label which is A. In contrast to $D_{<:}$, there are infinitely many type member labels (A, B, C, etc) in $D_\wedge$. In addition, $D_\wedge$ also has field members. This is reflected in types by the

x, y, z	**Variable**
a, b, c	**Term member**
A, B, C	**Type member**

$S, T, U ::=$		**Type**
	$\top$	top type
	$\bot$	bottom type
	$\{A : S..U\}$	type declaration
	$\{a : T\}$	field declaration
	$x.A$	path type
	$\forall(x : S)U_x$	function
	$S \wedge U$	intersection

$$\frac{}{\Gamma \vdash_{D_\wedge} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{D_\wedge} \bot <: T}\ \textsc{Bot} \qquad \frac{}{\Gamma \vdash_{D_\wedge} T <: T}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash_{D_\wedge} S_2 <: S_1 \quad \Gamma \vdash_{D_\wedge} U_1 <: U_2}{\Gamma \vdash_{D_\wedge} \{A : S_1..U_1\} <: \{A : S_2..U_2\}}\ \textsc{Bnd} \qquad \frac{\Gamma \vdash_{D_\wedge} T_1 <: T_2}{\Gamma \vdash_{D_\wedge} \{a : T_1\} <: \{a : T_2\}}\ \textsc{Fld}$$

$$\frac{\Gamma \vdash_{D_\wedge} S_2 <: S_1 \quad \Gamma; x : S_2 \vdash_{D_\wedge} U_1 <: U_2}{\Gamma \vdash_{D_\wedge} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2}\ \textsc{All}$$

$$\frac{\Gamma \vdash_{D_\wedge} \Gamma(x) <: \{A : S..\top\} \quad \Gamma \vdash_{D_\wedge} \Gamma(x) <: \{A : \bot..U\}}{\Gamma \vdash_{D_\wedge} S <: U}\ \text{BB}$$

$$\frac{\Gamma \vdash_{D_\wedge} \Gamma(x) <: \{A : S..\top\}}{\Gamma \vdash_{D_\wedge} S <: x.A}\ \textsc{Sel1} \qquad \frac{\Gamma \vdash_{D_\wedge} \Gamma(x) <: \{A : \bot..U\}}{\Gamma \vdash_{D_\wedge} x.A <: U}\ \textsc{Sel2}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{D_\wedge} T <: S \\ \Gamma \vdash_{D_\wedge} T <: U\end{array}}{\Gamma \vdash_{D_\wedge} T <: S \wedge U}\ \textsc{And-I} \qquad \frac{\Gamma \vdash_{D_\wedge} S <: T}{\Gamma \vdash_{D_\wedge} S \wedge U <: T}\ \textsc{And-E1} \qquad \frac{\Gamma \vdash_{D_\wedge} U <: T}{\Gamma \vdash_{D_\wedge} S \wedge U <: T}\ \textsc{And-E2}$$

Figure 5.1: Definition of $D_\wedge$

field declaration $\{a : T\}$, as shaded. Moreover, $D_\wedge$ has intersection types, as indicated by its name. Otherwise the types are the same as in $D_{<:}$.

The subtyping rules have already been in normal form and the unshaded rules are the same as $D_{<:}$ normal form (Figure 3.11). Notice that transitivity is replaced by the BB rule, as in $D_{<:}$ normal form. The FLD rule is added to define an intuitive subtyping relation between field declarations, which indicates the types in them are in covariant position. Relations on intersection types are expressed by AND-I, AND-E1 and AND-E2.

In Section 3.11.2, I discussed why small-step subtyping does not work well with intersection types. Nonetheless, the AND-E1 and AND-E2 rules are results of direct engineering, and a similar definition can be found in $F_\wedge$ defined in Pierce [1991]. This definition of $D_\wedge$ is justified by proving the admissibility of transitivity. The statement of the theorem is the same as the one of $D_{<:}$ normal form.

Theorem 5.1. *For any type T and two subtyping derivations in $D_\wedge$, $\mathcal{D}_1$ and $\mathcal{D}_2$, the following hold:*

(1) (transitivity) If $\mathcal{D}_1$ concludes $\Gamma \vdash_{D_\wedge} S <: T$ and $\mathcal{D}_2$ concludes $\Gamma \vdash_{D_\wedge} T <: U$, then $\Gamma \vdash S <: U$.

(2) (narrowing) If $\mathcal{D}_1$ concludes $\Gamma \vdash_{D_\wedge} S <: T$ and $\mathcal{D}_2$ concludes $\Gamma; x : T; \Gamma' \vdash_{D_\wedge} S' <: U'$, then $\Gamma; x : S; \Gamma' \vdash_{D_\wedge} S' <: U'$.

(3) If $\mathcal{D}_1$ concludes $\Gamma \vdash_{D_\wedge} T' <: tdh(\{A : S'..T\}, l)$ and $\mathcal{D}_2$ concludes $\Gamma \vdash_{D_\wedge} T <: U$, then $\Gamma \vdash_{D_\wedge} T' <: tdh(\{A : S'..U\}, l)$.

(4) If $\mathcal{D}_1$ concludes $\Gamma \vdash_{D_\wedge} S <: T$ and $\mathcal{D}_2$ concludes $\Gamma \vdash_{D_\wedge} T' <: tdh(\{A : T..U'\}, l)$, then $\Gamma \vdash_{D_\wedge} T' <: tdh(\{A : S..U'\}, l)$.

The same as in $D_{<:}$, transitivity relies on narrowing and two other statements about $\text{tdh}(\cdot)$. It appears that the proof steps in $D_\wedge$ are almost the same as $D_{<:}$, other than necessary adaptations for intersection types. I will only discuss those cases.

Proof. The proof is almost the same as the one of Theorem 3.20. In the case split of transitivity, there are more cases due to intersection types.

The AND-E1-*any* case: In this case, $S = S_0 \wedge S_1$ for some S_0 and S_1 and the antecedents are:

1. $\Gamma \vdash_{D_\wedge} S_0 <: T$, and

2. $\Gamma \vdash_{D_\wedge} T <: U$.

By the inductive hypothesis, $\Gamma \vdash_{D_\wedge} S_0 <: U$ is concluded and by And-E1, the goal $\Gamma \vdash_{D_\wedge} S_0 \wedge S_1 <: U$ is shown. The dual case And-E2-*any* is shown by the same argument.

The *any*-And-I case: In this case, $U = U_0 \wedge U_1$ for some U_0 and U_1 and the antecedents are:

1. $\Gamma \vdash_{D_\wedge} S <: T$,
2. $\Gamma \vdash_{D_\wedge} T <: U_0$, and
3. $\Gamma \vdash_{D_\wedge} T <: U_1$.

By the inductive hypothesis, $\Gamma \vdash_{D_\wedge} S <: U_0$ and $\Gamma \vdash_{D_\wedge} S <: U_1$ are obtained and then by And-I, $\Gamma \vdash_{D_\wedge} S <: U_0 \wedge U_1$ is concluded as desired.

The And-I-And-E1 case: In this case, $T = T_0 \wedge T_1$ and the antecedents are:

1. $\Gamma \vdash_{D_\wedge} S <: T_0$,
2. $\Gamma \vdash_{D_\wedge} S <: T_1$, and
3. $\Gamma \vdash_{D_\wedge} T_0 <: U$.

Notice that by the inductive hypothesis with the first and third antecedents, the goal $\Gamma \vdash_{D_\wedge} S <: U$ can already be concluded. That means the second antecedent is completely dropped. The And-I-And-E2 case is proved similarly.

Other cases are exactly the same as the cases in $D_{<:}$ so I omit the discussion here. □

From the normal form, it is straightforward to show the undecidability of $D_\wedge$.

Theorem 5.2. *$D_\wedge$ subtyping is undecidable.*

Proof. The target theorem is the following equivalence.

$$\Gamma \vdash_{F^-_{<:}} S <: U \text{ iff } \langle\!\langle \Gamma \rangle\!\rangle \vdash_{D_\wedge} [\![S]\!] <: [\![U]\!]$$

The interpretation functions are the ones defined in Definition 3.5. The proof is almost the same as the one of Theorem 3.22. In the only if direction, since intersection types do not appear in the image of $[\![\cdot]\!]$, all cases of And-E1, And-E2 and And-I are discharged by contradiction. □

5.2 Strong Kernel $D_\wedge$

Similar to $D_{<:}$, $D_\wedge$ also has a strong kernel variant. In this section, I will define strong kernel $D_\wedge$ and discuss its relation with $D_\wedge$ (or full $D_\wedge$ following the convention), and in the next section, I will adapt stare-at subtyping and prove that it decides strong kernel $D_\wedge$. One can also define kernel $D_\wedge$ but as explained previously, kernel calculi require identical parameter types when comparing dependent function types and this restriction becomes too much as the feature set grows. Hence I will omit the discussion on step subtyping and kernel $D_\wedge$.

Definition 5.2. *Strong kernel $D_\wedge$ is defined in Figure 5.2.*

Strong kernel $D_\wedge$ simply extends strong kernel $D_{<:}$ with the additional features. The extra rules in strong kernel $D_{<:}$ are shaded. These rules can be seen as the two-contextual version of the corresponding rules in full $D_\wedge$.

Many properties of strong kernel $D_{<:}$ hold in strong kernel $D_\wedge$ as well.

Lemma 5.3. *Strong kernel $D_{<:}$ is reflexive.*

$$\Gamma_1 \vdash_{D_\wedge SK} T <: T \dashv \Gamma_2$$

Lemma 5.4. *If $\Gamma_1 \vdash_{D_\wedge SK} \top <: U \dashv \Gamma_2$, then $\Gamma_1 \vdash_{D_\wedge SK} S <: U \dashv \Gamma_2$.*

Lemma 5.5. *If $\Gamma_1 \vdash_{D_\wedge SK} S <: \bot \dashv \Gamma_2$, then $\Gamma_1 \vdash_{D_\wedge SK} S <: U \dashv \Gamma_2$.*

The soundness w.r.t. full $D_\wedge$ is proved in the same way as in strong kernel $D_{<:}$. Recall that the soundness proof relies on a concept $OPE_{<:}$ which is defined in Definition 4.7.

Theorem 5.6. *If $\Gamma_1 \vdash_{D_{<:}SK} S <: U \dashv \Gamma_2$, $\Gamma \subseteq_{<:} \Gamma_1$ and $\Gamma \subseteq_{<:} \Gamma_2$, then $\Gamma \vdash_{D_{<:}} S <: U$.*

The soundness theorem uses reflexivity of $OPE_{<:}$ to specialize the theorem to the same context on both sides.

Theorem 5.7. *If $\Gamma \vdash_{D_{<:}SK} S <: U \dashv \Gamma$, then $\Gamma \vdash_{D_{<:}} S <: U$.*

Due to its decidability to be shown in Section 5.6, strong kernel $D_\wedge$ is not complete w.r.t. full $D_\wedge$.

$$\frac{}{\Gamma_1 \vdash_{D_\wedge SK} T <: \top \dashv \Gamma_2}\ \textsc{Sk-Top} \qquad \frac{}{\Gamma_1 \vdash_{D_\wedge SK} \bot <: T \dashv \Gamma_2}\ \textsc{Sk-Bot}$$

$$\frac{}{\Gamma_1 \vdash_{D_\wedge SK} x.A <: x.A \dashv \Gamma_2}\ \textsc{Sk-VRefl}$$

$$\frac{\Gamma_1 \vdash_{D_\wedge SK} S_1 >: S_2 \dashv \Gamma_2 \quad \Gamma_1 \vdash_{D_\wedge SK} U_1 <: U_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{D_\wedge SK} \{A : S_1..U_1\} <: \{A : S_2..U_2\} \dashv \Gamma_2}\ \textsc{Sk-Bnd}$$

$$\frac{\Gamma_1 \vdash_{D_\wedge SK} T_1 <: T_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{D_\wedge SK} \{a : T_1\} <: \{a : T_2\} \dashv \Gamma_2}\ \textsc{Sk-Fld}$$

$$\frac{\Gamma_1 \vdash_{D_\wedge SK} S_1 >: S_2 \dashv \Gamma_2 \quad \Gamma_1; x : S_1 \vdash_{D_\wedge SK} U_1 <: U_2 \dashv \Gamma_2; x : S_2}{\Gamma_1 \vdash_{D_\wedge SK} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2 \dashv \Gamma_2}\ \textsc{Sk-All}$$

$$\frac{\Gamma_1 \vdash_{D_\wedge SK} \{A : S..\top\} >: \Gamma_2(x) \dashv \Gamma_2}{\Gamma_1 \vdash_{D_\wedge SK} S <: x.A \dashv \Gamma_2}\ \textsc{Sk-Sel1}$$

$$\frac{\Gamma_1 \vdash_{D_\wedge SK} \Gamma_1(x) <: \{A : \bot..U\} \dashv \Gamma_2}{\Gamma_1 \vdash_{D_\wedge SK} x.A <: U \dashv \Gamma_2}\ \textsc{Sk-Sel2} \qquad \frac{\Gamma_1 \vdash_{D_\wedge SK} S <: T \dashv \Gamma_2}{\Gamma_1 \vdash_{D_\wedge SK} S \wedge U <: T \dashv \Gamma_2}\ \textsc{Sk-And-E1}$$

$$\frac{\Gamma_1 \vdash_{D_\wedge SK} U <: T \dashv \Gamma_2}{\Gamma_1 \vdash_{D_\wedge SK} S \wedge U <: T \dashv \Gamma_2}\ \textsc{Sk-And-E2}$$

$$\frac{\Gamma_1 \vdash_{D_\wedge SK} T <: S \dashv \Gamma_2 \quad \Gamma_1 \vdash_{D_\wedge SK} T <: U \dashv \Gamma_2}{\Gamma_1 \vdash_{D_\wedge SK} T <: S \wedge U \dashv \Gamma_2}\ \textsc{Sk-And-I}$$

Figure 5.2: Definition of strong kernel $D_\wedge$

5.3 Revealing in $D_\wedge$

Starting from this section, I will discuss the algorithmic subtyping of $D_\wedge$. I will adapt stare-at subtyping to $D_\wedge$ and show that it decides strong kernel $D_\wedge$.

Recall that stare-at subtyping relies on a core operation called **Revealing**, and two wrappers **Upcast** and **Downcast**. Intersection types also complicate these operations. Consider the following subtyping relation.

$$\Gamma \vdash_{D_\wedge} x.A \wedge T <: x.A$$

This relation is obviously true. To derive this conclusion, first $x.A$ in $x.A \wedge T$ needs to be chosen by And-E1, and then reflexivity applies to derive $\Gamma \vdash_{D_\wedge} x.A <: x.A$. That implies there needs to be an operation to select types from the intersection types.

Definition 5.3. *$\wedge$-**Traversal** is defined in Figure 5.3.*

For a judgment $S \mapsto U$, S is the input type and U is the output type.

Intuitively, the operation non-deterministically selects a non-intersection type from a tree of intersections. The non-determinism is introduced by the At-Left or At-Right rules. This to some degree is expected. In the previous example, At-Left applies to retrieve $x.A$; if the subtyping problem is changed slightly to $T \wedge x.A <: x.A$, then At-Right should apply. In general, the right choice is unknown until both rules are tried.

Luckily, since syntactical constructs are always finite, there are only a finite number of non-deterministic choices to make and therefore can be dealt with by exhaustive search (and therefore the problem remains decidable). This also means the implementation needs to remember each point of these non-deterministic choices, and perform backtracking searches whenever subsequent operations fail. This will be discussed in greater details in Section 5.4.

Revealing, **Upcast** and **Downcast** need to be extended with $\wedge$-**Traversal**.

Definition 5.4. ***Revealing**, **Upcast** and **Downcast** are defined in Figure 5.3.*

Similar to $D_{<:}$,

1. *A **Revealing** judgment $\Gamma \vdash_{D_{<:S}} S \Uparrow U \dashv \Gamma'$ has Γ and a type S as inputs and Γ' and a type U as outputs.*
2. *An **Upcast** judgment $\Gamma \vdash_{D_{<:S}} x.A \nearrow U \dashv \Gamma'$ has Γ, a variable x and a type member label A as inputs and Γ' and a type U as outputs.*
3. ***Downcast** judgment $\Gamma \vdash_{D_{<:S}} x.A \searrow U \dashv \Gamma'$ is defined similarly as **Upcast**.*

The main complication of **Revealing** comes from the Rv-Bot and Rv-Bnd rules.

$\wedge$-Traversal

$$\frac{T \text{ is not } \wedge}{T \mapsto T}\ \textsc{At-Found} \qquad \frac{S \mapsto S'}{S \wedge U \mapsto S'}\ \textsc{At-Left} \qquad \frac{U \mapsto U'}{S \wedge U \mapsto U'}\ \textsc{At-Right}$$

Revealing

$$\frac{T \text{ is not a path}}{\Gamma \vdash_{D_\wedge S} T \Uparrow T \dashv \Gamma}\ \textsc{Rv-Stop} \qquad \frac{}{\Gamma \vdash_{D_\wedge S} T \Uparrow \top \dashv \bullet}\ \textsc{Rv-Top}^*$$

$$\frac{T \mapsto T_0 \quad \Gamma_1 \vdash_{D_\wedge S} T_0 \Uparrow T_1 \dashv \Gamma_1' \quad T_1 \mapsto \bot}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_\wedge S} x.A \Uparrow \bot \dashv \bullet}\ \textsc{Rv-Bot}$$

$$\frac{\begin{array}{c} T \mapsto T_0 \quad \Gamma_1 \vdash_{D_\wedge S} T_0 \Uparrow T_1 \dashv \Gamma_1' \\ T_1 \mapsto \{A : S..U\} \quad U \mapsto U_0 \quad \Gamma_1' \vdash_{D_\wedge S} U_0 \Uparrow U' \dashv \Gamma_1'' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_\wedge S} x.A \Uparrow U' \dashv \Gamma_1''}\ \textsc{Rv-Bnd}$$

Upcast/ Downcast

$$\frac{}{\Gamma \vdash_{D_\wedge S} x.A \nearrow \top \dashv \bullet}\ \textsc{U-Top}^* \qquad \frac{}{\Gamma \vdash_{D_\wedge S} x.A \searrow \bot \dashv \bullet}\ \textsc{D-Bot}^*$$

$$\frac{T \mapsto T_0 \quad \Gamma_1 \vdash_{D_\wedge S} T_0 \Uparrow T_1 \dashv \Gamma_1' \quad T_1 \mapsto \bot}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_\wedge S} x.A \nearrow \bot \dashv \bullet}\ \textsc{U-Bot}$$

$$\frac{T \mapsto T_0 \quad \Gamma_1 \vdash_{D_\wedge S} T_0 \Uparrow T_1 \dashv \Gamma_1' \quad T_1 \mapsto \{A : S..U\}}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_\wedge S} x.A \nearrow U \dashv \Gamma_1'}\ \textsc{U-Bnd}$$

$$\frac{T \mapsto T_0 \quad \Gamma_1 \vdash_{D_\wedge S} T_0 \Uparrow T_1 \dashv \Gamma_1' \quad T_1 \mapsto \bot}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_\wedge S} x.A \searrow \top \dashv \bullet}\ \textsc{D-Top}$$

$$\frac{T \mapsto T_0 \quad \Gamma_1 \vdash_{D_\wedge S} T_0 \Uparrow T_1 \dashv \Gamma_1' \quad T_1 \mapsto \{A : S..U\}}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_\wedge S} x.A \searrow S \dashv \Gamma_1'}\ \textsc{D-Bnd}$$

Figure 5.3: Definition of **Revealing** and new definitions of **Upcast** and **Downcast**

1. First the rules both obtain the type T which x binds to. However, T might be an intersection type, and therefore **$\wedge$-Traversal** is used to obtain a T_0 that is guaranteed to not be an intersection (it might or might not be another path type).

2. After the first recursive call, a new type T_1 is returned. However, this type might also be an intersection. In particular, if it hides a $\bot$ in it, then the operation should stop here (Rv-Bot); or if it has a type declaration, then a second recursive call needs to be made (Rv-Bnd). One might think there can be a third choice: **$\wedge$-Traversal** can select a path type, but this choice is not necessary, as I will explain later.

3. In Rv-Bnd, U does not directly get passed into **Revealing**, but first is processed by another **$\wedge$-Traversal** to get a non-intersection component type U_0. The result of **Revealing** of U_0 is the final result.

Upcast and **Downcast** remain moderate wrappers of **Revealing**, so their extensions follow **Revealing**'s tightly. Essentially, **$\wedge$-Traversal** is inserted into these operations to ensure these operations do not have to handle the complication of intersection types directly.

Previously, I left a question behind: in **Revealing**, why is there not a rule for $T_1 \mapsto y.B$? I claim this case is not necessary, because if this choice was the right one, then it would have been chosen by the premise $U \mapsto U_0$ in the Rv-Bnd rule already!

To understand this, let us consider an example with the following context,

$$\begin{aligned} U &= \{C : \bot..\{A : \bot..\forall(w : \top)\top\}\} \\ S &= \{B : \bot..z.C \wedge \top\} \\ \Gamma &= z : \top \wedge U; y : S; x : y.B \end{aligned}$$

The goal is to apply **Revealing** to $x.A$ in this context and obtain $\forall(w : \top)\top$ as the result. The high-level intention is to show that x has type $\{A : \bot..\forall(w : \top)\top\}$ and therefore $x.A$ is a subtype of the upper bound of the type declaration. This is given by the following derivation:

$$\dfrac{\begin{array}{c} y.B \mapsto y.B \quad z : \top \wedge U; y : S \vdash_{D_\wedge S} y.B \Uparrow \{A : \bot..\forall(w : \top)\top\} \dashv \bullet \\ \{A : \bot..\forall(w : \top)\top\} \mapsto \{A : \bot..\forall(w : \top)\top\} \\ \forall(w : \top)\top \mapsto \forall(w : \top)\top \quad \bullet \vdash_{D_\wedge S} \forall(w : \top)\top \Uparrow \forall(w : \top)\top \dashv \bullet \end{array}}{\Gamma \vdash_{D_\wedge S} x.A \Uparrow \forall(w : \top)\top \dashv \bullet}\ \text{Rv-Bnd}$$

In this derivation, all $\wedge$-**Traversal** are trivially constructed by AT-FOUND and the second **Revealing** is also trivial. Therefore, the first **Revealing** is critical to this derivation. The first **Revealing** of $y.B$ is given by the following derivation:

$$\dfrac{\begin{array}{c} S \mapsto S \quad z : \top \wedge U \vdash_{D_\wedge S} S \Uparrow S \dashv z : \top \wedge U \\ S \mapsto S \quad z.C \wedge \top \mapsto z.C \quad z : \top \wedge U \vdash_{D_\wedge S} z.C \Uparrow \{A : \bot..\forall(w : \top)\top\} \dashv \bullet \end{array}}{z : \top \wedge U; y : S \vdash_{D_\wedge S} y.B \Uparrow \{A : \bot..\forall(w : \top)\top\} \dashv \bullet} \text{RV-BND}$$

Since $S = \{B : \bot..z.C \wedge \top\}$ is not an intersection, all premises before the shaded premise $z.C \wedge \top \mapsto z.C$ are trivial. In this shaded premise, $z.C$ is selected by $\wedge$-**Traversal** which is subsequently passed in the second recursive call of **Revealing**.

In this example, it is necessary to apply **Revealing** to $z.C$ at some point to achieve the high-level intention of showing x of type $\{A : \bot..\forall(w : \top)\top\}$, because this type declaration is hidden inside of the type z binds to. Explicitly defining a rule in **Revealing** handling the case of $T_1 \mapsto y.B$ is possible, but as shown in the example above, it simply does not add power to the operation while increases the complexity of the formal proofs. This design of **Revealing** will be justified by the strengthened inductive hypothesis of the completeness of stare-at subtyping in strong kernel $D_\wedge$, as will be shown later.

5.4 Stare-at Subtyping

Having defined **Revealing**, **Upcast** and **Downcast**, now I can extend stare-at subtyping. Compared to **Revealing**, stare-at subtyping itself is just changed a little in terms of rules but its execution becomes far more complicated than it looks.

Definition 5.5. *Stare-at subtyping is defined in Figure 5.4.*

For a stare-at subtyping judgment $\Gamma_1 \gg S <: U \ll \Gamma_2$, all four places are the inputs, and it outputs true if the inputs satisfy its definition.

The execution of these rules will be explained next.

The extension to stare-at subtyping in $D_{<:}$ is shaded. Since $D_\wedge$ has data field members, stare-at subtyping is augmented to handle them (SA-FLD). SA-LEFT1, SA-LEFT2 and SA-RIGHT correspond to AND-E1, AND-E2 and AND-I, except there is one extra predicate: T should not be an intersection in SA-LEFT1 or SA-LEFT2. This predicate is here to ensure that the SA-RIGHT rule always applies before SA-LEFT1 and SA-LEFT2. That

$$\frac{}{\Gamma_1 \gg T <: \top \ll \Gamma_2}\ \textsc{SA-Top} \qquad \frac{}{\Gamma_1 \gg \bot <: T \ll \Gamma_2}\ \textsc{SA-Bot}$$

$$\frac{\begin{array}{c}\Gamma_2 \vdash_{D_{<:S}} x.A \searrow T \dashv \Gamma_2' \\ \Gamma_1 \gg S <: T \ll \Gamma_2'\end{array}}{\Gamma_1 \gg S <: x.A \ll \Gamma_2}\ \textsc{SA-Sel1} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash_{D_{<:S}} x.A \nearrow T \dashv \Gamma_1' \\ \Gamma_1' \gg T <: U \ll \Gamma_2\end{array}}{\Gamma_1 \gg x.A <: U \ll \Gamma_2}\ \textsc{SA-Sel2}$$

$$\frac{}{\Gamma_1 \gg x.A <: x.A \ll \Gamma_2}\ \textsc{SA-VRefl} \qquad \frac{\Gamma_1 \gg S >: S' \ll \Gamma_2 \quad \Gamma_1 \gg U <: U' \ll \Gamma_2}{\Gamma_1 \gg \{A : S..U\} <: \{A : S'..U'\} \ll \Gamma_2}\ \textsc{SA-Bnd}$$

$$\frac{\Gamma_1 \gg T <: T' \ll \Gamma_2}{\Gamma_1 \gg \{a : T\} <: \{a : T'\} \ll \Gamma_2}\ \textsc{SA-Fld}$$

$$\frac{\Gamma_1 \gg S >: S' \ll \Gamma_2 \quad \Gamma_1; x : S \gg U <: U' \ll \Gamma_2; x : S'}{\Gamma_1 \gg \forall(x : S)U <: \forall(x : S')U' \ll \Gamma_2}\ \textsc{SA-All}$$

$$\frac{\begin{array}{l}T \text{ is not an intersection} \\ \quad \Gamma_1 \gg S <: T \ll \Gamma_2\end{array}}{\Gamma_1 \gg S \wedge U <: T \ll \Gamma_2}\ \textsc{SA-Left1} \qquad \frac{\begin{array}{l}T \text{ is not an intersection} \\ \quad \Gamma_1 \gg U <: T \ll \Gamma_2\end{array}}{\Gamma_1 \gg S \wedge U <: T \ll \Gamma_2}\ \textsc{SA-Left2}$$

$$\frac{\Gamma_1 \gg T <: S \ll \Gamma_2 \quad \Gamma_1 \gg T <: U \ll \Gamma_2}{\Gamma_1 \gg T <: S \wedge U \ll \Gamma_2}\ \textsc{SA-Right}$$

Figure 5.4: Definition of stare-at subtyping

is, if Bob holds an intersection type, the algorithm should always apply SA-Right until Bob holds a non-intersection type, before considering applying SA-Left1 or SA-Left2.

Let us consider a more complicated situation when Alice holds an intersection type and Bob holds a path type.

Consider Figure 5.5. On the left, it shows a diagram that asserts subtyping between any type T and the intersection of S and U. In the diagram, a solid line denotes a subtyping judgment as a premise, and a dotted line denotes a subtyping judgment as a conclusion.

$$\begin{tikzcd}
S & & U \\
& S \wedge U \arrow[ul] \arrow[ur] & \\
& T \arrow[uul] \arrow[uur] \arrow[u, dotted] &
\end{tikzcd}
\qquad
\begin{tikzcd}
& x.A & \\
S & & U \\
& S \wedge U \arrow[ul] \arrow[ur] \arrow[uu] & \\
& T \arrow[uul] \arrow[uur] \arrow[uuu, dotted, bend right] &
\end{tikzcd}$$

Figure 5.5: Subtyping of intersection types and path types

The diagram on the left denotes the AND-I rule.

$$\frac{\Gamma \vdash_{D_\wedge} T <: S \quad \Gamma \vdash_{D_\wedge} T <: U}{\Gamma \vdash_{D_\wedge} T <: S \wedge U}\ \text{AND-I}$$

The diagram on the right, then, denotes the following fact.

$$\frac{\Gamma \vdash_{D_\wedge} S \wedge U <: x.A \quad \Gamma \vdash_{D_\wedge} T <: S \quad \Gamma \vdash_{D_\wedge} T <: U}{\Gamma \vdash_{D_\wedge} T <: x.A}$$

From the conclusion, however, there is no indication that some other S and U might be involved in the derivation! Moreover, there is not always a relation between S, U and $x.A$. Consider a general situation in which applying **Downcast** to $x.A$ yields $S \wedge U$. This proves $\Gamma \vdash_{D_\wedge} S \wedge U <: x.A$. Since S, U and $x.A$ can be unrelated, the subtyping relation $\Gamma \vdash_{D_\wedge} T <: x.A$ cannot be alternatively established by SA-LEFT1, SA-LEFT2 or SA-RIGHT; it is necessary to apply SA-SEL1 so **Downcast** of $x.A$ obtains $S \wedge U$, so that SA-RIGHT can be used to draw the conclusion. That is the following derivation:

$$\frac{\Gamma_2 \vdash_{D_\wedge S} x.A \searrow S \wedge U \dashv \Gamma_2' \qquad \dfrac{\Gamma_1 \gg T <: S \ll \Gamma_2' \quad \Gamma_1 \gg T <: U \ll \Gamma_2'}{\Gamma_1 \gg T <: S \wedge U \ll \Gamma_2'}\ \text{SA-RIGHT}}{\Gamma_1 \gg T <: x.A \ll \Gamma_2}$$

Therefore, for stare-at subtyping to decide strong kernel, the execution must handle this case correctly.

As analyzed above, the execution of stare-at subtyping in $D_\wedge$ is significantly more complicated than $D_{<:}$ due to intersection types. To summarize, when executing stare-at subtyping in $D_\wedge$, the rules are tried in the following order:

1. If Bob holds an intersection type, then apply SA-Right until he no longer holds one.

2. If Bob holds a path type, then try the following rules until the first success.

   (a) SA-VRefl.

   (b) SA-Sel1.

   (c) SA-Sel2.

   (d) SA-Left1 and SA-Left2.

3. If Bob holds a non-path and non-intersection type and Alice holds an intersection type, then try SA-Left1 and SA-Left2 until the first success.

4. If all previous cases do not apply, then the rest of the rules apply accordingly. At this point, Alice and Bob are guaranteed to not hold intersection types, so the problem just regresses to the easier situation in $D_{<:}$.

5. If a trial fails, then a correct implementation should backtrack to the latest branching point. Notice that **$\wedge$-Traversal** also creates branching points, so backtracking needs to get into **Revealing**, **Upcast** and **Downcast**, so all possible branches introduced by intersection types are exhausted. A subtyping problem is admitted if and only if one trial (the first one) is successful, and if all trials fail, then the problem is rejected.

Although the algorithm requires an exhaustive search with backtracking, the search space of each problem is still finite so the algorithm is guaranteed to terminate.

Definition 5.6. *The measure M of types and contexts is defined by the following equations.*

$$\begin{aligned}
M(\top) &= 1 \\
M(\bot) &= 1 \\
M(x.A) &= 2 \\
M(\forall(x:S)U) &= 1 + M(S) + M(U) \\
M(\{A : S..U\}) &= 1 + M(S) + M(U) \\
M(\{a : T\}) &= 1 + M(T) \\
M(S \wedge U) &= 1 + M(S) + M(U)
\end{aligned}$$

$$M(\Gamma) = \sum_{x:T\in\Gamma} M(T)$$

The measure M of $D_\wedge$ just slightly extends the measure of $D_{<:}$ and it simply measures the syntactical sizes of types and contexts.

The following lemma can be proved quite straightforwardly by induction.

Lemma 5.8. *If $S \mapsto U$, then $M(S) \geq M(U)$.*

If $\Gamma \vdash_{D_{<:}S} S \Uparrow U \dashv \Gamma'$, then $M(\Gamma) + M(S) \geq M(\Gamma') + M(U)$.

If $\Gamma \vdash_{D_{<:}S} x.A \nearrow (\searrow) U \dashv \Gamma'$, then $M(\Gamma) + M(x.A) > M(\Gamma') + M(U)$.

Theorem 5.9. *stare-at subtyping terminates as a non-deterministic algorithm.*

Looking at the rules for stare-at subtyping, each recursive case in the rules has a strictly smaller input. Though there can be a huge amount of backtracking, each problem only looks at a finite number of subproblems and each subproblem has a strictly smaller input size than the original input. Therefore stare-at subtyping has only a finite search space and the problem is decidable.

Despite its visual similarity to $D_{<:}$, stare-at subtyping has become subtly more complicated. Problems in the *DOT* family are very often in a seemingly simple disguise but in fact very difficult to tackle and get right. We must be very careful about informal conclusions, which are the fastest way to fall into the traps of the family.

5.5 Properties of Stare-at Subtyping and Properties

After the termination argument, the next step is to examine various properties of stare-at subtyping. The first one is reflexivity.

Theorem 5.10. *Stare-at subtyping is reflexive.*

$$\Gamma_1 \gg T <: T \ll \Gamma_2$$

This property can no longer be proved by induction on T. It requires the following strengthened characterization of how well stare-at subtyping is capable of handling intersection types.

Theorem 5.11. *Stare-at subtyping handles commutations and associations of intersection types.*

$$\textit{If forall } T, U \mapsto T \textit{ implies } S \mapsto T,$$
$$\textit{then } \Gamma_1 \gg S <: U \ll \Gamma_2.$$

This theorem says that if S and U are intersection types and all non-intersection types in U can be found in S, then $\Gamma_1 \gg S <: U \ll \Gamma_2$ holds. This theorem says that stare-at subtyping does not care how intersection types are concretely organized.

Proof of Theorem 5.10 and Theorem 5.11. Perform mutual induction on both input types between these two theorems. □

Similar to $D_{<:}$, the following properties of the operations can be shown.

Lemma 5.12. *(**Revealing** gives prefixes) If $\Gamma \vdash_{D_\wedge S} S \Uparrow U \dashv \Gamma'$, then Γ' is a prefix of Γ.*

Lemma 5.13. *(**Revealing** returns no path) If $\Gamma \vdash_{D_\wedge S} S \Uparrow U \dashv \Gamma'$, then U is not a path.*

Lemma 5.14. ***Revealing** terminates as a non-deterministic algorithm.*

Proof. Its non-determinism comes from **∧-Traversal**. However, each subproblem decreases by the length of the input context. □

Lemma 5.15. *(soundness of **Revealing**) If $\Gamma \vdash_{D_\wedge S} S \Uparrow U \dashv \Gamma'$, then $\Gamma \vdash_{D_\wedge} S <: U$.*

Lemma 5.16. *The following hold.*

1. *If $\Gamma \vdash_{D_\wedge S} x.A \nearrow (\searrow) T \dashv \Gamma'$, then Γ' is a prefix of Γ.*
2. *If $\Gamma \vdash_{D_\wedge S} x.A \nearrow T \dashv \Gamma'$, then $\Gamma \vdash_{D_\wedge} x.A <: T$.*
3. *If $\Gamma \vdash_{D_\wedge S} x.A \searrow T \dashv \Gamma'$, then $\Gamma \vdash_{D_\wedge} T <: x.A$.*

Then soundness can be concluded.

Theorem 5.17. *(soundness of stare-at subtyping) If $\Gamma_1 \gg S <: U \ll \Gamma_2$, $\Gamma \subseteq_{<:} \Gamma_1$ and $\Gamma \subseteq_{<:} \Gamma_2$, then $\Gamma \vdash_{D_\wedge} S <: U$.*

Theorem 5.18. *If $\Gamma \gg S <: U \ll \Gamma$, then $\Gamma \vdash_{D_\wedge} S <: U$.*

5.6 Strong Kernel $D_\wedge$ and Stare-at Subtyping

In this section, I will connect strong kernel $D_\wedge$ and stare-at subtyping by showing the algorithm actually decides strong kernel.

Compared to stare-at subtyping in $D_{<:}$, stare-at subtyping in $D_\wedge$ has one more layer: $\wedge$-**Traversal**. Following the same method in $D_{<:}$, the following lemma is to connect $\wedge$-**Traversal** with strong kernel $D_\wedge$.

Lemma 5.19. *If $S \mapsto T$ and $\Gamma_1 \vdash_{D_\wedge SK} T <: U \dashv \Gamma_2$, then $\Gamma_1 \vdash_{D_\wedge SK} S <: U \dashv \Gamma_2$.*

Then this lemma can be used to connect **Revealing** with strong kernel:

Lemma 5.20. *If $\Gamma_1 \vdash_{D_\wedge S} S \Uparrow T \dashv \Gamma_1'$ and $\Gamma_1 \vdash_{D_\wedge SK} T <: U \dashv \Gamma_2$, then $\Gamma_1 \vdash_{D_\wedge SK} S <: U \dashv \Gamma_2$.*

By applying this lemma, soundness is ready to be established.

Theorem 5.21. *(soundness of stare-at subtyping w.r.t. strong kernel $D_\wedge$)*

If $\Gamma_1 \gg S <: U \ll \Gamma_2$, then $\Gamma_1 \vdash_{D_\wedge SK} S <: U \dashv \Gamma_2$.

The same as in strong kernel $D_{<:}$, completeness is harder to establish than soundness. In the case of strong kernel $D_\wedge$, it is even more difficult because the strengthened inductive hypothesis needs to mention $\wedge$-**Traversal**.

Theorem 5.22. *(completeness of stare-at subtyping w.r.t. strong kernel $D_{<:}$) If $\Gamma_1 \vdash_{D_\wedge SK} S <: U \dashv \Gamma_2$, then $\Gamma_1 \gg S <: U \ll \Gamma_2$.*

Proof. The proof requires the following strengthened inductive hypothesis: if $\Gamma_1 \vdash_{D_\wedge SK} S <: U \dashv \Gamma_2$ and this derivation contains n steps, then $\Gamma_1 \gg S <: U \ll \Gamma_2$ and if U is of the form $\{A : T_1..T_2\}$, then $\Gamma_1 \vdash_{D_\wedge S} S \Uparrow S' \dashv \Gamma_1'$, and there are two types S_0 and S_1, such that

1. $S \mapsto S_0$, and
2. $\Gamma_1 \vdash_{D_\wedge S} S_0 \Uparrow S_1 \dashv \Gamma_1'$, and
3. (a) $S_1 \mapsto \bot$, or
   (b) $S_1 \mapsto \{A : T_1'..T_2'\}$ for some types T_1' and T_2', such that

i. $\Gamma_1 \gg T_1 <: T_1' \ll \Gamma_2$, and

ii. $\Gamma_1 \vdash_{D_\wedge SK} T_2' <: T_2 \dashv \Gamma_2$, and the number of steps in this derivation is less than or equal to n.

In the Sk-And-E1 and Sk-And-E2 cases, there is a complication, because the SA-Left1 and SA-Left2 rules require Bob to hold a non-intersection type, while this is not guaranteed to hold in the antecedents. This can be resolved by a nested induction.

Other cases are straightforward adaptations of the ones in Theorem 4.43 so I omit the discussion here. □

The strengthened inductive hypothesis also provides a formal justification of the completeness of the **Revealing** operation. Recall that in Section 5.3, I informally explained why there is not a case selecting $T_1 \mapsto y.B$. In the strengthened inductive hypothesis, there are only two returned types from **Revealing** to consider: $\bot$ or a type declaration. In particular, there is no case for path types. In addition, the strengthened inductive hypothesis concludes that an **$\wedge$-Traversal** always applies before **Revealing**, which is exactly how **Revealing** is designed. These observations have led to the confidence in the design of operations in stare-at subtyping.

Chapter 6

$\mu DART$

In the previous chapter, I studied the interaction between path dependent types and intersection types. Recall that there are three core features in DOT, and the last one is recursive types (or μ types). Recursive types are used to model objects as in object oriented languages. Fields in the same object definition can refer to each other. Since $D_\wedge$ is already quite complicated, it would be beneficial to take a step back and consider the interaction between path dependent types and recursive types only, leaving intersection types out, which motivates μ-Dependent And Record Types, or $\mu DART$.

In this chapter, I will focus on describing the interaction between path dependent types and recursive types and explaining why they are difficult to deal with. I will show how stare-at subtyping can continue to be adapted so that it handles recursive types. When handling recursive types, the idea of separating subtyping contexts to Alice and Bob starts to really thrive: it is hard to see how to achieve the same with only one single context.

In the previous chapters, the type assignment problem is considered straightforward, because most of the difficulties lie in the subtyping problem. However, due to the introduction of μ types, type assignment becomes more subtle and requires more attention. Therefore, I will also describe a set of bi-directional type assignment rules just for *variables*. It turns out that μ types allow variables to have richer typing behaviors than any other kinds of terms.

Unlike $D_{<:}$ and $D_\wedge$ which have been shown undecidable and have kernel forms, I am not able to prove $\mu DART$ undecidable or derive its kernel forms. This is due to the difficulties of unravelling typing and subtyping rules described in Section 3.11.2 and is left as a future problem.

Γ	**Context**
a, b, c	**Data mamber**
A, B, C	**Type member**
x, y, z	**Free variable**
$s, t, u ::=$	**Term**
x	variable
v	value
$x.a$	data access
$x\ y$	application
$\text{let } x : T? = s \text{ in } t$	let binding
$v ::=$	**Value**
$\nu(x : DS_x)\{ds_x\}$	definitions
$\lambda(x : T).t$	function
$ds ::= d*$	**Definitions**

$d ::=$	**Definition**
$\{a = t\}$	data definition
$\{A = T\}$	type definition
$DS ::= D*$	**Declarations**
$D ::=$	**Declaration**
$\{a : T\}$	data field
$\{A : S..T\}$	type field
$S, T, U ::=$	**Type**
$\top$	top
$\bot$	bottom
$\forall(x : S)T$	function
$x.A$	type projection
DS	record type
$\mu(x : DS_x)$	object type

Figure 6.1: $\mu DART$ syntax

6.1 Definition of $\mu DART$

$\mu DART$ extends $D_{<:}$ with data field members and recursive types (or μ types), so that the language is capable of expressing objects. Compared to DOT, $\mu DART$ can express the same set of values. The only difference is that the typing system in $\mu DART$ cannot express intersection types so it is not as expressive.

Definition 6.1. *The abstract syntax of $\mu DART$ is defined in Figure A.8. The subtyping rules are defined in Figure 6.2 and the typing rules are defined in Figure 6.3.*

The $\mu DART$ calculus extends $D_{<:}$ in several ways. First, it adds an optional type declaration in let bindings. Second, it extends objects to allow multiple members, both data field and type members. There are countably infinitely many data field member and type member labels in $\mu DART$, while $D_{<:}$ has only one which is A. Third, $\mu DART$ makes objects self-recursive: in an object $\nu(x : DS_x)\{ds_x\}$, the type DS_x and the body ds_x of

Subtyping

$$\frac{}{\Gamma \vdash_{\mu DART} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{\mu DART} \bot <: T}\ \textsc{Bot} \qquad \frac{}{\Gamma \vdash_{\mu DART} T <: T}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash_{\mu DART} S_2 <: S_1 \quad \Gamma \vdash_{\mu DART} U_1 <: U_2}{\Gamma \vdash_{\mu DART} \{A : S_1..U_1\} <: \{A : S_2..U_2\}}\ \textsc{Bnd} \qquad \frac{\Gamma \vdash_{\mu DART} T_1 <: T_2}{\Gamma \vdash_{\mu DART} \{a : T_1\} <: \{a : T_2\}}\ \textsc{Fld}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mu DART} S_2 <: S_1 \\ \Gamma; x : S_2 \vdash_{\mu DART} U_1 <: U_2\end{array}}{\Gamma \vdash_{\mu DART} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2}\ \textsc{All} \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mu DART} S <: T \\ \Gamma \vdash_{\mu DART} T <: U\end{array}}{\Gamma \vdash_{\mu DART} S <: U}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash_{\mu DART} x : \{A : S..U\}}{\Gamma \vdash_{\mu DART} S <: x.A}\ \textsc{Sel1} \qquad \frac{\Gamma \vdash_{\mu DART} x : \{A : S..U\}}{\Gamma \vdash_{\mu DART} x.A <: U}\ \textsc{Sel2}$$

$$\frac{|DS_1| \neq 0}{\Gamma \vdash_{\mu DART} \{DS_1; DS_2\} <: DS_2}\ \textsc{Drop1} \qquad \frac{|DS_2| \neq 0}{\Gamma \vdash_{\mu DART} \{DS_1; DS_2\} <: DS_1}\ \textsc{Drop2}$$

$$\frac{\Gamma \vdash_{\mu DART} \{DS\} <: \{DS_1\} \quad \Gamma \vdash_{\mu DART} \{DS\} <: \{DS_2\}}{\Gamma \vdash_{\mu DART} \{DS\} <: DS_1; DS_2}\ \textsc{Merge}$$

Figure 6.2: Subtyping rules of $\mu DART$

the object may depend on x, the object self-reference. Accordingly, $\mu DART$ adds the self-recursive object type $\mu(x : DS_x)$.

$\mu DART$ adds several rules to the subtyping rules to handle the added features. The FLD rule handles data members. The DROP1, DROP2 and MERGE rules express the subtyping between record types. In the DROP1 and DROP2 rules, it is required that $|DS| \neq 0$. This means the length of DS shall be positive. This is reasonable because these rules should not be applied pointlessly. Similar to DOT, μ types do not participate in subtyping.

$\mu DART$ adds the LET2 rule to handle type-ascribed let binding. In the OBJ-I rule, notice that it is the μ type being inserted into the context, instead of a record type $\{DS_x\}$ like DOT's OBJ-I rule. That is, the DOT rule exposes the self reference and relies on the free variable lookup in the context to discover the same x. Unfortunately, this approach introduces a cycle into the typing environment ($\{DS_x\}$ refers to its binder x), and this

Type Assignment

$$\frac{}{\Gamma \vdash_{\mu DART} x : \Gamma(x)}\ \textsc{Var} \qquad \frac{\Gamma \vdash_{\mu DART} t : S \quad \Gamma \vdash_{\mu DART} S <: U}{\Gamma \vdash_{\mu DART} t : U}\ \textsc{Sub}$$

$$\frac{\Gamma; x : S \vdash_{\mu DART} t : U}{\Gamma \vdash_{\mu DART} \lambda(x : S)t : \forall(x : S)U}\ \textsc{All-I} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mu DART} x : \forall(z : S)U_z \\ \Gamma \vdash_{\mu DART} y : S\end{array}}{\Gamma \vdash_{\mu DART} x\ y : U_y}\ \textsc{All-E}$$

$$\frac{\begin{array}{c}\Gamma; x : \mu(z : DS_z) \vdash_{\mu DART} ds_x : DS_x \\ dom(ds) \text{ is unique}\end{array}}{\Gamma \vdash_{\mu DART} \nu(z : DS_z)\{ds_z\} : \mu(z : DS_z)}\ \textsc{Obj-I} \qquad \frac{\Gamma \vdash_{\mu DART} x : \{a : T\}}{\Gamma \vdash_{\mu DART} x.a : T}\ \textsc{Obj-E}$$

$$\frac{\Gamma \vdash_{\mu DART} x : \{DS_x\}}{\Gamma \vdash_{\mu DART} x : \mu(z : DS_z)}\ \textsc{Rec-I} \qquad \frac{\Gamma \vdash_{\mu DART} x : \mu(z : DS_z)}{\Gamma \vdash_{\mu DART} x : \{DS_x\}}\ \textsc{Rec-E}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mu DART} t : S \quad x \notin fv(U) \\ \Gamma; x : S \vdash_{\mu DART} u : U\end{array}}{\Gamma \vdash_{\mu DART} \text{let } x = t \text{ in } u : U}\ \textsc{Let1} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mu DART} t : S \quad x \notin fv(U) \\ \Gamma; x : S \vdash_{\mu DART} u : U\end{array}}{\Gamma \vdash_{\mu DART} \text{let } x : S = t \text{ in } u : U}\ \textsc{Let2}$$

Object Definition Type Assignment

$$\frac{\Gamma \vdash_{\mu DART} t : T}{\Gamma \vdash_{\mu DART} \{a = t\} : \{A : T\}}\ \textsc{Def-Trm} \qquad \frac{}{\Gamma \vdash_{\mu DART} \{A = T\} : \{A : T..T\}}\ \textsc{Def-Typ}$$

$$\frac{}{\Gamma \vdash_{\mu DART} \{\} : \{\}}\ \textsc{Def-Nil} \qquad \frac{\Gamma \vdash_{\mu DART} \{d\} : \{D\} \quad \Gamma \vdash_{\mu DART} \{ds\} : \{DS\}}{\Gamma \vdash_{\mu DART} \{d; ds\} : \{D; DS\}}\ \textsc{Def-Cons}$$

Figure 6.3: Typing rules of $\mu DART$

disagrees with the definition of well-formed contexts in Definition 2.4. In contrast, in $\mu DART$, the context is extended with the recursive type $x : \mu(z : DS_z)$, leaving the cycle encapsulated inside the recursive μ type. This makes μ types indicators of cycles in the context, while keeping the context itself syntactically cycle-free. Moreover, there is no loss of expressiveness: Rec-E can be used to unpack the μ type to obtain a record type. This treatment simplifies the static properties of the calculus without compromising any expressiveness.

As discussed in Section 3.11.2, since I am not able to transform calculi with μ types to normal form, the undecidability of $\mu DART$ is technically unknown. However, looking at the complexity of its definition, I conjecture that it is undecidable.

Conjecture 5. *$\mu DART$ typing and subtyping are undecidable.*

This conjecture justifies the incomplete algorithmic rules to be presented in this chapter.

6.2 Difficulties of μ Types

In this section, I will describe why μ types deserve a special look. From a high level, μ types enable mutual references of path dependent types in their bounds. In the previous treatments, **Exposure**/ **Revealing** simply follows the upper bounds to find a supertype that is not a path. This method no longer works with the presence of μ types. Consider the following subtyping problem.

$$\Gamma = y : \mu(z : \{A : \bot..z.B; B : \bot..z.A\}); x : y.A$$
$$\Gamma \vdash_{\mu DART} x.A <:? \bot$$

If the same strategy is applied to this problem, then to find a non-path supertype of $x.A$, it is necessary to find a non-path supertype of $y.A$ first. But the following can be shown:

$$\dfrac{\dfrac{}{\Gamma \vdash_{\mu DART} y : \mu(z : (A : \bot..z.B; B : \bot..z.A))}\textsc{Var}}{\Gamma \vdash_{\mu DART} y : \{A : \bot..y.B; B : \bot..y.A\}}\textsc{Rec-E}$$

So a naive design of **Exposure** or **Revealing** will have the following search sequence.

$$y.A \to y.B \to y.A \to y.B \to y.A \to y.B \to ...$$

Clearly the algorithm no longer terminates. What is worse, cycles do not have to be this explicit. It is not difficult to engineer cycles that loop in the subtyping algorithm instead of within **Revealing**.

$$\begin{aligned} & x : \mu(x : \{A : \bot..x.B; B : \bot..\forall(z : \bot)x.A\}) \\ & ; y : \mu(y : \{A : y.B..\top; B : \forall(z : \bot)y.A..\top\}) \\ \vdash_{\mu DART} & \ x.A <:? y.A \end{aligned}$$

Though $x.A$ can **Upcast** to $\forall(z:\bot)x.A$ and $y.A$ can **Downcast** to $\forall(z:\bot)y.A$, now the subtyping algorithm starts to loop.

$$\begin{array}{lccll}
 & x.A & <:? & y.A & \\
\Rightarrow & \forall(z:\bot)x.A & <:? & \forall(z:\bot)y.A & \\
\Rightarrow & x.A & <:? & y.A & \Rightarrow \ldots
\end{array}$$

With μ types, currently both **Revealing** and stare-at subtyping will loop for some inputs. The approach I take here is to understand why these two components loop during the design of the algorithm.

6.3 Stare-at Subtyping for $\mu DART$

From the previous examples, the observation is that the ultimate cause of looping is the interactions between path types and μ types. On the other hand, stare-at subtyping does not handle path types directly. Looking at the definition of stare-at subtyping of $D_{<:}$, defined in Figure 4.8, the rules simply get down to substructures and attempt to decide subtyping relation for subproblems and problems on path types are passed over to **Revealing**. So let us leave the looping problem alone for a moment, and first set up stare-at subtyping in $\mu DART$.

Definition 6.2. *Stare-at subtyping is defined in Figure 6.4.*

For a stare-at subtyping judgment $\Gamma_1 \gg S <: U \ll \Gamma_2$, all four places are the inputs, and it outputs true if inputs satisfy its definition.

The rules are extended with the SA-Mu and SA-Rcd rules compared to $D_{<:}$. SA-Mu only admits subtyping between identical μ types. This is the case because the calculus itself does not define subtyping between μ types beyond reflexivity. One might expect at least to be able to permute the declarations in μ types. Indeed, it is desirable, but the definition of the calculus does not permit it. I will show how to modify the calculus to achieve more in Chapter 7.

SA-Rcd checks the subtyping relation between two record types. It simply delegates the check to another set of subtyping rules, $\Gamma_1 \gg \{DS_1\} <:_D \{DS_2\} \ll \Gamma_2$. This set of subtyping rules loops over the record type held by Bob and for each declaration on Bob's side, it is required to have a matching declaration on Alice's side and satisfy the subtyping

$$\frac{}{\Gamma_1 \gg T <: \top \ll \Gamma_2}\ \textsc{SA-Top} \qquad \frac{}{\Gamma_1 \gg \bot <: T \ll \Gamma_2}\ \textsc{SA-Bot}$$

$$\frac{\begin{array}{c}\Gamma_2 \vdash_{\mu DARTS} x.A \searrow T \dashv \Gamma_2' \\ \Gamma_1 \gg S <: T \ll \Gamma_2'\end{array}}{\Gamma_1 \gg S <: x.A \ll \Gamma_2}\ \textsc{SA-Sel1} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mu DARTS} x.A \nearrow T \dashv \Gamma_1' \\ \Gamma_1' \gg T <: U \ll \Gamma_2\end{array}}{\Gamma_1 \gg x.A <: U \ll \Gamma_2}\ \textsc{SA-Sel2}$$

$$\frac{}{\Gamma_1 \gg x.A <: x.A \ll \Gamma_2}\ \textsc{SA-VRefl} \qquad \frac{\begin{array}{c}\Gamma_1 \gg S >: S' \ll \Gamma_2 \\ \Gamma_1; x : S \gg U <: U' \ll \Gamma_2; x : S'\end{array}}{\Gamma_1 \gg \forall(x : S)U <: \forall(x : S')U' \ll \Gamma_2}\ \textsc{SA-All}$$

$$\frac{}{\Gamma_1 \gg \mu(x : DS_x) <: \mu(x : DS_x) \ll \Gamma_2}\ \textsc{SA-Mu}$$

$$\frac{\Gamma_1 \gg \{DS_1\} <:_D \{DS_2\} \ll \Gamma_2}{\Gamma_1 \gg \{DS_1\} <: \{DS_2\} \ll \Gamma_2}\ \textsc{SA-Rcd}$$

Declarations subtyping

$$\frac{\begin{array}{c}\{A : S..U\} \in \{DS_1\} \\ \Gamma_1 \gg S >: S' \ll \Gamma_2 \quad \Gamma_1 \gg U <: U' \ll \Gamma_2 \quad \Gamma_1 \gg \{DS_1\} <:_D \{DS_2\} \ll \Gamma_2\end{array}}{\Gamma_1 \gg \{DS_1\} <:_D \{A : S'..U'; DS_2\} \ll \Gamma_2}\ \textsc{SAR-Bnd}$$

$$\frac{\{a : T\} \in \{DS_1\} \quad \Gamma_1 \gg T <: T' \ll \Gamma_2 \quad \Gamma_1 \gg \{DS_1\} <:_D \{DS_2\} \ll \Gamma_2}{\Gamma_1 \gg \{DS_1\} <:_D \{a : T'; DS_2\} \ll \Gamma_2}\ \textsc{SAR-Fld}$$

$$\frac{}{\Gamma_1 \gg \{DS\} <:_D \{\} \ll \Gamma_2}\ \textsc{SAR-Nil}$$

Figure 6.4: Definition of stare-at subtyping

relation correctly (SAR-Bnd and SAR-Fld). The base case is when Bob holds an empty record (SAR-Nil), which is a supertype of all record types.

We can show reflexivity holds without any further consideration.

Lemma 6.1. *Stare-at subtyping is reflexive.*

$$\Gamma_1 \gg T <: T \ll \Gamma_2$$

Surely the rules would still loop for the reasons explained in the previous section, but having defined the rules allows to pinpoint the actual reasons for looping. Since only the SA-Sel1 and SA-Sel2 rules involve path types, they are the rules to blame. In both rules, **Upcast** and **Downcast** are used, which are just wrappers of **Revealing**. **Revealing** is responsible for searching non-path supertypes of path types and is regarded as the core operation. For this reason, it becomes clear that this operation needs to be well engineered so that the two kinds of loops in the examples can be eliminated.

6.4 Revealing Reconsidered

Since **Revealing** is the core operation of path type handling, it is beneficial to reconsider how this operation should be structured.

The first idea is to augment the operation to handle μ types. One candidate is the following rule, mimicking Rv-Bnd.

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uuparrow \mu(z : DS_z) \dashv \Gamma'_1 \quad \{A : S..U_z\} \in DS_z \quad \Gamma'_1 \vdash_{\mu DARTS} U_x \Uuparrow U' \dashv \Gamma''_1}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uuparrow U' \dashv \Gamma''_1}$$

When a μ type is encountered, the type member label A is looked up in the type declarations in the μ type, and **Revealing** is applied to the upper bound with self reference z substituted with the variable x in the path type. However, this rule is problematic. It is U_x that is passed into the second recursive call and U_x may refer to x, whereas x is guaranteed to not exist in Γ'_1!

Imagine $U_z = z.B$, then $U_x = x.B$. In the second **Revealing**, $x.B$ needs to be resolved, but since x is no longer in the context, the only type that can be returned and surely correct is $\top$, which makes this operation too weak. This just means the input context of the second recursive call must know something about x. This motivates two other candidates.

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uuparrow \mu(z : DS_z) \dashv \Gamma'_1 \quad \{A : S..U_z\} \in DS_z \quad \Gamma'_1; x : T \vdash_{\mu DARTS} U_x \Uuparrow U' \dashv \Gamma''_1}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uuparrow U' \dashv \Gamma''_1}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mu DARTS} T \Uuparrow \mu(z : DS_z) \dashv \Gamma'_1 \\ \{A : S..U_z\} \in DS_z \quad \Gamma'_1; x : \mu(z : DS_z) \vdash_{\mu DARTS} U_x \Uuparrow U' \dashv \Gamma''_1\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uuparrow U' \dashv \Gamma''_1}$$

These two candidates only differ in the second recursive **Revealing**. In the first candidate, $x : T$, namely the original binding, is added back to Γ_1', the returned context from the first recursive call, while in the second candidate, $x : \mu(z : DS_z)$, namely the resolved object type, is added to Γ_1'. A bit of thinking leads to the conclusion that the second candidate is better. This is because **Revealing** removes bindings from contexts; so what if T refers to another variable which has been removed in Γ_1'? From the rule, there is no obvious guarantee to prevent this. On the other hand, in the second candidate, it is quite expected that $\mu(z : DS_z)$ remains closed in Γ_1'. Therefore at least the second candidate operates on a well-formed input context.

Despite getting closer, the second candidate fails to handle all looping situations. Consider the situation in which T is already $\mu(z : DS_z)$. Then the first recursive **Revealing** does not change the type. Let us further assume $\Gamma_2 = \bullet$ and $U_z = z.B$. Then the rule is reduced to

$$\frac{\{A : S..z.B\} \in DS_z \quad \Gamma_1; x : \mu(z : DS_z) \vdash_{\mu DARTS} x.B \Uparrow U' \dashv \Gamma_1'}{\Gamma_1; x : \mu(z : DS_z) \vdash_{\mu DARTS} x.A \Uparrow U' \dashv \Gamma_1'}$$

The overall inputs are $\Gamma_1; x : \mu(z : DS_z)$ and $x.A$. The input context of the recursive call remains the same and the input type is changed slightly to $x.B$. It is not obvious at all why $x.B$ is smaller than $x.A$ by any measure! In fact, if $x.A$ and $x.B$ are upper bounds of each other, then this rule will not terminate. Even if in some situation $x.A$ and $x.B$ can be distinguished by some measure, this measure is likely to perform non-trivial analysis, which turns the termination argument semantic again.

Looking from afar, when a μ type is encountered, there are two kinds of path types: a path type that points back to the same μ type, or a path type referring to a variable on its left in the context. Without μ types, the situation is always the second case. That is why in $D_{<:}$, returning a prefix context in **Revealing** is sufficient. To tackle the first case, a special treatment is motivated, and this results in another operation, **Exposure**$^{\mu}$.

6.5 μ Types as Cyclic Contexts

Consider the form of a μ type, e.g.

$$\mu(x : \{A : \bot..x.B; B : \bot..x.A\})$$

A μ type looks like a context: the type member labels are just like variables and each binds to two types: the lower bound and the upper bound (while field member labels are

$$\frac{A \notin dom(DS_x)}{DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} \top \dashv DS_x}\ \textsc{Em-Top*} \qquad \frac{\{A : S..y.B\} \in DS_x \quad x \neq y}{DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} y.B \dashv DS_x \backslash A}\ \textsc{Em-Sel}$$

$$\frac{\{A : S..T\} \in DS_x \quad T \text{ is not a path}}{DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} T \dashv DS_x \backslash A}\ \textsc{Em-Stop}$$

$$\frac{\{A : S..x.B\} \in DS_x \quad DS_x \backslash A \vdash_{\mu DARTS} x.B \Uparrow^{\mu} T \dashv DS'_x}{DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} T \dashv DS'_x}\ \textsc{Em-Recur}$$

Figure 6.5: Definition of **Exposure**$^{\mu}$

ignored here because they are not involved in path type handling). This "context" differs from the regular typing context in its cyclic nature: the bounds can refer to other type member labels defined in the same object. Therefore, resolving a path type in an object is the same as handling a cyclic context. A second difference is that this cyclic context comes with a variable as the self reference, used to tell whether a path type is pointing back to the same μ type. Thus an operation, **Exposure**$^{\mu}$, is defined to handle this problem by considering μ types as cyclic context-like components in the typing contexts. The $\vdash$ symbol is used to emphasize the contextual nature of μ types, and the self reference variable is identified as a subscript.

Definition 6.3. ***Exposure***$^{\mu}$ *is defined in Figure 6.5.*

For a ***Exposure***$^{\mu}$ *judgment* $DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} T \dashv DS'_x$*, declarations* DS_x*, variable* x *and type member label* A *are inputs and type* T *and declaations* DS'_x *are outputs.*

Recall that the operation **Exposure** is defined in step subtyping and is used to find a supertype of a path type that is not a path type. **Exposure**$^{\mu}$ follows the same essence: it is used to find a supertype of a path type within a μ type which is either not a path type, or a path type but not pointing back to the same μ type (hence the variable must not be the self reference of the μ type).

In the definition, I make use of set theoretic notation $\backslash$ to mean removing the type declaration associated with the label from the declarations. For example, $DS\backslash A$ is guaranteed to not have label A anymore and all other labels remain.

To see how this operation works, let us look at the rules one by one.

1. (Em-Top) If the label is not found in the μ type, then $\top$ is the only safe result to return and the declarations remain untouched.

2. (Em-Sel) If the label is found and the upper bound is a path type, but its variable is different from the self reference, then the intention is achieved and the returned declarations are the same as the input except A is removed.

3. (Em-Stop) If the label is found and the upper bound is not a path, then it is also done.

4. (Em-Recur) If the label is found and the upper bound points back to the same μ type, then recursion needs to happen within the μ type. What if $x.B$ refers back to $x.A$? This is resolved by first removing A from DS as indicated by the input of the recursive call: $DS_x \backslash A$. If $x.B$ indeed has $x.A$ as upper bound, then this recursive call will fall to Em-Top and the whole algorithm will terminate.

As seen in the Em-Recur rule, the algorithm relies on removing type labels to ensure termination. This is illustrated by the following example. If $x.A$ and $x.B$ are the upper bounds of each other, it is the same as if they do not have upper bound. More concretely, consider the μ type in the beginning of this section:

$$\mu(x : \{A : \bot..x.B; B : \bot..x.A\})$$

A and B have full freedom to be instantiated to anything and therefore there is virtually no upper bound on them.

The following properties characterize the **Exposure**$^\mu$ operation.

Lemma 6.2. *(soundness) If* $\Gamma \vdash_{\mu DART} x : \{DS_x\}$ *and* $DS_x \vdash_{\mu DARTS} x.A \Uparrow^\mu T \dashv DS'_x$*, then* $\Gamma \vdash_{\mu DART} x.A <: T$*.*

Lemma 6.3. ***Exposure***$^\mu$ *returns either a non-path type or a path type that does not refer to the self reference of the* μ *type.*

Lemma 6.4. *Returned declarations* $\{DS'_x\}$ *are a subset of the input declarations* $\{DS_x\}$*. To use set theoretic notation,* $\{DS'_x\} \subseteq \{DS_x\}$*.*

Lemma 6.5. ***Exposure***$^\mu$ *terminates as an algorithm.*

Proof. The measure is the length of input declarations. □

This operation resolves the first problem posed in Section 6.2: since **Exposure**$^\mu$ is a terminating algorithm to handle path search, **Revealing** shall also terminate now. However, it still seems unclear how to ensure the termination of stare-at subtyping. The trick lies in the second return value of **Exposure**$^\mu$, namely the new declarations.

From the rules, we can see that for each type label the operation touches, it will be removed from the returned declarations. For example, in the Em-Sel and Em-Stop rules, $DS_x \backslash A$ is returned. In the Em-Recur rule, $DS_x \backslash A$ is the input of the recursive call, and therefore it can be inferred that DS'_x cannot have labels A and B. The removal is the secret to make the whole subtyping algorithm terminate.

6.6 Types Are Resources!

Recall that in Section 6.2, I described two reasons why stare-at subtyping loops. The first one loops within **Revealing**, which has been resolved by **Exposure**$^\mu$, as described in the previous section. The second loops in the stare-at subtyping itself, due to μ types. One more idea is needed to ensure termination of the algorithm.

Consider why a problem is undecidable. Generally speaking, there must be a part of the problem the scale of which extends to infinity. In the context of programming languages, due to the Curry-Howard correspondence, $\mu DART$ corresponds to some structural logic and one source of non-termination are the structural properties, especially contraction. From a logical point of view, contraction is the property to allow a variable to be referred to infinitely many times, and μ types directly enable that in subtyping. This leads to the loop in the examples. On the other hand, a decidable problem must have some finite structure. One observation is that resources must be finite, and therefore if the execution of an algorithm consumes resources, it must terminate.

Sometimes, one might use a natural number to constrain the depth of recursion to ensure termination. This can be considered as each recursion consuming 1 from the natural number and is a general solution to the problem, but it lacks elegance to build this treatment into the theory. In reality, this might cause unexpected problems: for example, wrapping the same definition in an object suddenly might cause type checking to fail. Moreover, it misleadingly suggests that all problems can be decided by simply increasing the depth, while this is theoretically not true.

Reviewing the definition of **Exposure**$^\mu$ in Figure 6.5 again, **Exposure**$^\mu$ removes a type label A whenever it is accessed. To state the same behavior in a different way, the operation *charges* or *consumes* the amount of the declaration associated with A if it is accessed. This leads to the observation that the syntax of types is resources! If the idea is extended to the whole stare-at subtyping, then the algorithm ought to terminate.

6.7 Revealing in $\mu DART$

To regard syntax of types as resources, it requires a consistent maintenance and the same source must not be consumed twice. Based on this observation, **Revealing** for $\mu DART$ can be defined.

Definition 6.4. ***Revealing***, ***Upcast*** *and* ***Downcast*** *is defined in Figure 6.6.*

Similar to $D_{<:}$,

1. *A* ***Revealing*** *judgment $\Gamma \vdash_{\mu DARTS} S \Uparrow U \dashv \Gamma'$ has Γ and a type S as inputs and Γ' and a type U as outputs.*
2. *An* ***Upcast*** *judgment $\Gamma \vdash_{\mu DARTS} x.A \nearrow U \dashv \Gamma'$ has Γ, a variable x and a type member label A as inputs and Γ' and a type U as outputs.*
3. ***Downcast*** *judgment $\Gamma \vdash_{\mu DARTS} x.A \searrow U \dashv \Gamma'$ works similarly as* ***Upcast****.*

Let me discuss the rules one by one.

1. Rv-Stop, Rv-Top and Rv-Bot are standard.
2. Rv-Bnd naturally adapts the similar situation in $D_{<:}$. It requires a lookup in $\{DS\}$ because now there are more type labels and the lookup is to make sure A is in $\{DS\}$. If A is not found, it falls back to Rv-Top. Recall that the asterisk next to the name denotes that it is a fallback rule.
3. Rv-Mu1 and Rv-Mu2 are more sophisticated. These rules are here to handle μ types. In these two rules, T is first revealed to a μ type $\mu(z : DS_z)$ and it is ensured that A is in the μ type. Then **Exposure**$^\mu$ is used to traverse within the μ type. Then two rules start to branch depending on what **Exposure**$^\mu$ returns. Let us look deeper into both rules.

Revealing

$$\frac{T \text{ is not a path}}{\Gamma \vdash_{\mu DARTS} T \Uparrow T \dashv \Gamma}\ \textsc{Rv-Stop} \qquad \frac{}{\Gamma \vdash_{\mu DARTS} T \Uparrow \top \dashv \bullet}\ \textsc{Rv-Top}^*$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \bot \dashv \Gamma_1'}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow \bot \dashv \bullet}\ \textsc{Rv-Bot}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \{DS\} \dashv \Gamma_1' \quad \{A : S..U\} \in \{DS\} \quad \Gamma_1' \vdash_{\mu DARTS} U \Uparrow U' \dashv \Gamma_1''}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow U' \dashv \Gamma_1''}\ \textsc{Rv-Bnd}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \mu(z : DS_z) \dashv \Gamma_1' \quad A \in dom(DS_z)\\ DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} U \dashv DS_x' \quad U \text{ is not a path}\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow U \dashv \Gamma_1'; x : \mu(z : DS_z')}\ \textsc{Rv-Mu1}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \mu(z : DS_z) \dashv \Gamma_1' \quad A \in dom(DS_z)\\ DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} y.B \dashv DS_x' \quad \Gamma_1' \vdash_{\mu DARTS} y.B \Uparrow U \dashv \Gamma_1''\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow U \dashv \Gamma_1''}\ \textsc{Rv-Mu2}$$

Upcast/ Downcast

$$\frac{}{\Gamma \vdash_{D_{<:S}} x.A \nearrow \top \dashv \bullet}\ \textsc{U-Top}^* \qquad \frac{}{\Gamma \vdash_{D_{<:S}} x.A \searrow \bot \dashv \bullet}\ \textsc{D-bot}^*$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \bot \dashv \Gamma_1'}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \nearrow \bot \dashv \bullet}\ \textsc{U-Bot} \qquad \frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \bot \dashv \Gamma_1'}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \searrow \top \dashv \bullet}\ \textsc{D-Top}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \{DS\} \dashv \Gamma_1'\\ \{A : S..U\} \in DS\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \nearrow U \dashv \Gamma_1'}\ \textsc{U-Bnd} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \{DS\} \dashv \Gamma_1'\\ \{A : S..U\} \in DS\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \searrow S \dashv \Gamma_1'}\ \textsc{D-Bnd}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \mu(z : DS_z) \dashv \Gamma_1' \quad \{A : S..U\} \in DS_x}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \nearrow U \dashv \Gamma_1'; x : \mu(z : DS_z \backslash A)}\ \textsc{U-Mu}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \mu(z : DS_z) \dashv \Gamma_1' \quad \{A : S..U\} \in DS_x}{\Gamma_1; x : T; \Gamma_2 \vdash_{D_{<:S}} x.A \searrow S \dashv \Gamma_1'; x : \mu(z : DS_z \backslash A)}\ \textsc{D-Mu}$$

Figure 6.6: Definition of **Revealing**, **Upcast** and **Downcast**

Rv-Mu1 returns a context in which the binding $x : T$ is replaced by $x : \mu(z : DS'_z)$ and Γ_2 is dropped. This is fine because U is in DS_x and nothing from Γ_2 will be mentioned by DS_x because the context is well-formed. Since U is not a path already, it is safe to return it as result. However, it might mention x, so the returned context must bind x consistently. Rebinding x to $\mu(z : DS'_z)$ allows other unused paths to remain interpretable, while $\mu(z : DS'_z)$ is closed in Γ'_1. This is because $\mu(z : DS_z)$ is closed in Γ'_1 (a lemma to be proved) and $\{DS'_z\} \subseteq \{DS_z\}$.

Rv-Mu2 recurs a second time because **Exposure**$^\mu$ still gives a path type. It returns the context that is returned by the second recursive call. Note that x is entirely forgotten in the result because **Exposure**$^\mu$ guarantees $x \neq y$, and therefore there is no reason to even remember x. In this case, the μ type behaves similarly to a record type in Rv-Bnd rule.

The **Upcast** and **Downcast** operations, again, remain moderate wrappers of **Revealing**. The added rules to handle μ types just return the right bounds and remove the declaration from the μ types, which follows **Revealing** in the same essential way.

From a high level point of view, these operations avoid resolving the same path type twice. This property is automatically true if a record type is encountered, because a well-formed context naturally prevents a record type from referring back to itself (or any other variables come after it); so only the case for μ types needs explicit maintenance. Now reconsider the second looping case and the example introduced in Section 6.2.

$$\begin{aligned} & x : \mu(x : \{A : \bot..x.B; B : \bot..\forall(z : \bot)x.A\}) \\ & ; y : \mu(y : \{A : y.B..\top; B : \forall(z : \bot)y.A..\top\}) \\ \vdash_{\mu DART} \ & x.A <:? \ y.A \end{aligned}$$

Once $x.A$ and $y.A$ are touched, they are removed from the context, and therefore the loop is broken. This is sufficient to make stare-at subtyping terminate with μ types. Consider SA-Sel2 again (the situation of SA-Sel1 follows by duality).

$$\frac{\Gamma_1 \vdash_{\mu DARTS} x.A \nearrow T \dashv \Gamma'_1 \quad \Gamma_1 \gg T <: U \ll \Gamma_2}{\Gamma_1 \gg x.A <: U \ll \Gamma_2}\ \text{SA-Sel2}$$

If $x : T'$ and T' is revealed to a μ type, then $x.A$ is guaranteed to be removed in Γ'_1 already, and even if T refers to $x.A$ in the future again, it will not be found anymore and just behave the same as $\top$ due to U-Top.

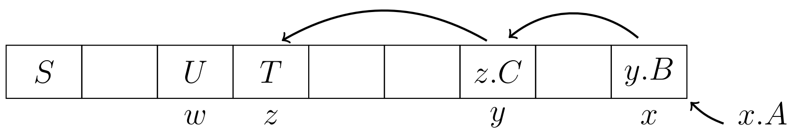

Figure 6.7: The original context of the example for **Revealing**

6.8 An Example of Revealing

The **Revealing** operation has become rather complicated and may be hard to understand. In this section, I will give an example of execution to show the **Revealing** operation in action. Figure 6.7 gives a typing context in which Alice applies **Revealing** to $x.A$. S is a well-formed type in the context and is not a path type. T is an abbreviation for following type:

$$T = \{C : \bot..\mu(s : \{B : \bot..s.D; D : \bot..\{A : \bot..w.E\}; F : \bot..S\})\}$$

and U is an abbreviation for following type:

$$U = \{E : \bot..S\}$$

Invoking **Revealing** in this context with $x.A$ will produce following steps.

1. Case for $z.C$:

$$z.C \Uuparrow \mu(s : \{B : \bot..s.D; D : \bot..\{A : \bot..w.E\}; F : \bot..S\}) \dashv \begin{array}{|c|c|c|}\hline S & & U \\ \hline \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{w} \end{array}$$

2. Case for $y.B$:

$$y.B \Uuparrow \{A : \bot..w.E\} \dashv \begin{array}{|c|c|c|c|}\hline S & & U & \mu(s : \{F : \bot..S\}) \\ \hline \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{w} & \multicolumn{1}{c}{y} \end{array}$$

3. Case for $x.A$ yields $w.E$, but then Alice needs to search $w.E$ in the context returned in Item 2:

$$w.E \Uuparrow S \dashv \begin{array}{|c|c|}\hline S & \\ \hline \end{array}$$

To resolve $x.A$, $y.B$ and then $z.C$ need to be resolved. Since z binds to a record type, the Rv-Bnd rule is used in the first step and it drops the part of the context starting at z. Note that the returned type is still closed in the smaller context. The second step uses the Rv-Mu1 rule, in which **Exposure**$^\mu$ is used to traverse the μ type returned in the first step. Type labels B and D are removed from the μ type because they are visited. Notice that the type of y has been replaced by the reduced μ type in the returned context. Since $y.B$ reveals to a record type, the previous call is the first recursive call in the Rv-Bnd rule. The third step is the second recursive call to resolve $w.E$, which further makes the context smaller. S is the final result of revealing $x.A$. We can see that indeed $x.A <: S$ is witnessed in the original context.

Revealing is not always able to reduce the number of bindings in the context when it encounters a path type. In the worst case, the number of bindings in the context remains the same. However, even in this case, **Revealing** is still able to consume some type members from a μ type in the context to make sure the overall syntactic size of types is smaller. This observation allows a very straightforward termination argument.

6.9 Properties of Operations and Stare-at Subtyping

Now the algorithm has been fully designed, so I will start formally investigating the properties of the operations and stare-at subtyping. In a **Revealing** judgment $\Gamma \vdash_{D_{<:S}} S \Uparrow U \dashv \Gamma'$ in $D_{<:}$, I have shown that Γ' is a prefix of Γ. Due to the introduction of μ types, this is no longer true due to the Rv-Mu1 rule. Their relation can be captured by the following notions.

Let us look at the μ types and the declarative rules defined in Figure 6.3 more closely. Though μ types are not directly connected by the subtyping relation due to lack of subtyping rules, some can still be converted from one to another via the Rec-I and Rec-E rules if bound to a variable. This complication is captured by the notion of *convertibility.*

Definition 6.5. *The Convertibility relation,* $\Gamma \vdash_{\mu DART} x : S \rightsquigarrow U$*, is defined in Figure 6.8.*

The intuition is that if S is convertible to U and $\Gamma \vdash x : S$, then $\Gamma \vdash x : U$. Convertibility is a necessary notion because μ types do not participate in the subtyping relation, while μ types are not entirely disconnected.

The following lemmas can be shown by induction.

Lemma 6.6. *If* $\Gamma \vdash_{\mu DART} x : T$ *and* $\Gamma \vdash_{\mu DART} x : T \rightsquigarrow U$*, then* $\Gamma \vdash_{\mu DART} x : U$*.*

Lemma 6.7. *If* $\{DS\} \subseteq \{DS'\}$*, then* $\Gamma \vdash_{\mu DART} x : \mu(z : DS'_z) \rightsquigarrow \mu(z : DS_z)$*.*

Recall that **Exposure**$^{\mu}$ removes labels from μ types whenever they are touched. Due to lack of subtyping relation between μ types, the μ types before and after can only be related by the convertibility relation.

$$\frac{\Gamma \vdash_{\mu DART} S <: U}{\Gamma \vdash_{\mu DART} x : S \rightsquigarrow U}\ \textsc{Cv-Sub} \qquad \frac{\Gamma \vdash_{\mu DART} \{DS_x\} <: \{DS'_x\}}{\Gamma \vdash_{\mu DART} x : \mu(z : DS_z) \rightsquigarrow \mu(z : DS'_z)}\ \textsc{Cv-Mu}$$

$$\frac{\Gamma \vdash_{\mu DART} x : S \rightsquigarrow T \quad \Gamma \vdash_{\mu DART} x : T \rightsquigarrow U}{\Gamma \vdash_{\mu DART} x : S \rightsquigarrow U}\ \textsc{Cv-Trans}$$

Figure 6.8: Two types are convertible w.r.t. a variable

$$\frac{}{\bullet \subseteq_{\rightsquigarrow} \bullet}\ \textsc{Opec-Nil} \qquad \frac{\Gamma \subseteq_{\rightsquigarrow} \Gamma'}{\Gamma; x : T \subseteq_{\rightsquigarrow} \Gamma'}\ \textsc{Opec-Drop}$$

$$\frac{\Gamma \subseteq_{\rightsquigarrow} \Gamma' \quad \Gamma \vdash_{\mu DART} x : S \rightsquigarrow U}{\Gamma; x : S \subseteq_{\rightsquigarrow} \Gamma'; x : U}\ \textsc{Opec-Keep}$$

Figure 6.9: The definition of order preserving convertible environment

Convertibility relates two types. In $D_{<:}$, a relation $OPE_{<:}$ is defined between two contexts. A similar notion is needed in $\mu DART$ to reflect the differences between subtyping and convertibility.

Definition 6.6. *The order preserving convertible environment, $OPE_{\rightsquigarrow}$, is defined in Figure 6.9.*

Generally speaking, if $\Gamma \subseteq_{\rightsquigarrow} \Gamma'$, then Γ is more informative than Γ'. In the Opec-Keep rule, the second premise requires the two types pushed into the contexts to be convertible. This is more permissive than the similar rule Ope-Keep in $OPE_{<:}$. Note that Γ' does not have to witness the convertibility relation.

Once the definitions are set up, properties can be proven.

Theorem 6.8. *($OPE_{\rightsquigarrow}$ is respectful) $OPE_{\rightsquigarrow}$ respects the declarative typing/subtyping rules.*

If $\Gamma \subseteq_{\rightsquigarrow} \Gamma'$ *and* $\Gamma' \vdash_{\mu DART} t : T$, *then* $\Gamma \vdash_{\mu DART} t : T$.
If $\Gamma \subseteq_{\rightsquigarrow} \Gamma'$ *and* $\Gamma' \vdash_{\mu DART} S <: U$, *then* $\Gamma \vdash_{\mu DART} S <: U$.

This theorem says that typing and subtyping are preserved as long as the information in the context is more precise.

Lemma 6.9. *If* $\Gamma' \vdash_{\mu DART} x : S \rightsquigarrow U$ *and* $\Gamma \subseteq_{\rightsquigarrow} \Gamma'$*, then* $\Gamma \vdash_{\mu DART} x : S \rightsquigarrow U$*.*

This lemma shows the interaction between convertibility and $OPE_{\rightsquigarrow}$, which is also quite intuitive.

Lemma 6.10. *($OPE_{\rightsquigarrow}$ is preorder) $OPE_{\rightsquigarrow}$ is reflexive and transitive.*

1. $\Gamma \subseteq_{\rightsquigarrow} \Gamma$
2. *If* $\Gamma \subseteq_{\rightsquigarrow} \Gamma'$ *and* $\Gamma' \subseteq_{\rightsquigarrow} \Gamma''$*, then* $\Gamma \subseteq_{\rightsquigarrow} \Gamma''$*.*

Since convertibility is general enough to express the connection between the input and output μ types of **Exposure**$^{\mu}$, $OPE_{\rightsquigarrow}$ is general enough to express the connection between the input and output contexts of **Revealing**, **Upcast** and **Downcast**.

Theorem 6.11. *If* $\Gamma \vdash_{\mu DARTS} S \Uparrow U \dashv \Gamma'$*, then* $\Gamma \subseteq_{\rightsquigarrow} \Gamma'$*.*

Theorem 6.12. *(soundness) If* $\Gamma \vdash_{\mu DARTS} S \Uparrow U \dashv \Gamma'$*, then* $\Gamma \vdash_{\mu DART} S <: U$*.*

Proof. The previous two theorems need to be proved by mutual induction. □

The following theorem needs to mention the well-formedness condition because it proves that the treatment in **Revealing** does not break this basic assumption.

Theorem 6.13. *(preservation of well-formedness) If* $\Gamma \vdash_{\mu DARTS} S \Uparrow U \dashv \Gamma'$*,* Γ *is well-formed and* S *is closed in* Γ*, then* Γ' *is also well-formed and* U *is closed in* Γ'*.*

Finally, **Revealing** returns a non-path type indeed.

Lemma 6.14. ***Revealing*** *returns a non-path type.*

Lemma 6.15. ***Revealing*** *terminates as an algorithm.*

Proof. The measure is the length of the input context. □

Amazingly, the statements of properties of **Revealing** in $\mu DART$ do not deviate too much from $D_{<:}$. This shows that stare-at subtyping is a very robust framework for subtyping decisions.

Similar lemmas can be proved for **Upcast** and **Downcast**. The situation in these two operations is simpler than **Revealing** and therefore the discussion is omitted.

Lemma 6.16. *The following hold.*

1. *If* $\Gamma \vdash_{\mu DARTS} x.A \nearrow (\searrow) T \dashv \Gamma'$*, then* $\Gamma \subseteq_{\leadsto} \Gamma'$.
2. *If* $\Gamma \vdash_{\mu DARTS} x.A \nearrow T \dashv \Gamma'$*, then* $\Gamma \vdash_{\mu DART} x.A <: T$.
3. *If* $\Gamma \vdash_{\mu DARTS} x.A \searrow T \dashv \Gamma'$*, then* $\Gamma \vdash_{\mu DART} T <: x.A$.
4. *If* $\Gamma \vdash_{\mu DARTS} x.A \nearrow (\searrow) T \dashv \Gamma'$*,* Γ *is well-formed and* $x \in dom(\Gamma)$*, then* T *is closed in* Γ'.

The statement of the soundness theorem of stare-at subtyping needs to be adjusted to reflect $OPE_{\leadsto}$.

Theorem 6.17. *(soundness of stare-at subtyping) If* $\Gamma_1 \gg S <: U \ll \Gamma_2$*,* $\Gamma \subseteq_{\leadsto} \Gamma_1$ *and* $\Gamma \subseteq_{\leadsto} \Gamma_2$*, then* $\Gamma \vdash_{\mu DART} S <: U$.

Then similarly, if Alice and Bob start with the same context, stare-at subtyping can be used as a subtyping decision procedure.

Theorem 6.18. *If* $\Gamma \gg S <: U \ll \Gamma$*, then* $\Gamma \vdash_{\mu DART} S <: U$.

At this point, soundness of stare-at subtyping is wrapped up. The next step is the termination proof. As hinted previously, the termination proof relies on the fact that no path type can be referred to more than once (or it will be the same as $\top$).

Definition 6.7. *The measures M of types, declarations and contexts are defined by the following equations.*

$$\begin{aligned} M(\top) &= 1 \\ M(\bot) &= 1 \\ M(x.A) &= 2 \\ M(\forall(x:S)U) &= 1 + M(S) + M(U) \\ M(\{DS\}) &= 1 + M(DS) \\ M(\mu(z:DS)) &= 2 + M(DS) \end{aligned} \qquad \begin{aligned} M(DS) &= \sum_{D \in DS} M(D) \\ M(\{A : S..U\}) &= 1 + M(S) + M(U) \\ M(\{a : T\}) &= 1 + M(T) \\ \\ M(\Gamma) &= \sum_{x:T \in \Gamma} M(T) \end{aligned}$$

Termination of the SA-SEL1 and SA-SEL2 cases requires the following lemmas.

Lemma 6.19. *If $DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} T \dashv DS'_x$ and $A \in dom(DS_x)$, then $M(DS'_x) + M(T) < M(DS_x)$.*

Lemma 6.20. *If $\Gamma \vdash_{\mu DARTS} S \Uparrow U \dashv \Gamma'$, then $M(\Gamma) + M(S) \geq M(\Gamma') + M(U)$. If $\Gamma \vdash_{\mu DARTS} x.A \nearrow (\searrow) U \dashv \Gamma'$, then $M(\Gamma) + M(x.A) > M(\Gamma') + M(U)$.*

This is enough to show the SA-SEL1 and SA-SEL2 recur on a strictly smaller problem.

Theorem 6.21. *Stare-at subtyping terminates as an algorithm.*

Proof. The complexity of this proof is no more than the one in $D_{<:}$. □

At this point, the algorithmic subtyping of a fragment of $\mu DART$ is handled.

6.10 How Large Is the Decidable Fragment?

Before discussing the bi-directional type assignment algorithm for variables, let me first discuss stare-at subtyping at a high level.

According to the previous discussion, the termination of stare-at subtyping is a direct consequence of the view of taking types as resources. Each visit of a path type consumes some types and a consistent global maintenance of this behavior leads to the final termination proof. Then the following two questions arise regarding the algorithm:

1. Is there a declarative form which is decided by the algorithm?
2. How much can this removal behavior reduce the decidable fragment?

I will attempt to give partial answers to both questions.

6.10.1 What is the language?

The first question asks if there can be a concise declarative definition of a language for which the algorithm is sound and complete. The goal is to mimic $D_{<:}$ in which kernel and strong kernel can be defined and their corresponding algorithms can be shown sound and complete. For $\mu DART$, how to achieve the same is far less clear, but this is not a fatal shortcoming either. In an actual implementation of the language, there are more optimizations that are not captured by the algorithmic rules (for various reasons, like performance or some desired but not admitted special cases). In an implementation, it is often not a part of the concern of compiler writers what declarative form the implementation represents. The algorithmic subtyping has defined a clear line of the design to follow and can be shown sound and terminating. These properties have given important guidance to compiler writers.

6.10.2 How does removal impact the decidable fragment?

This question is more important. It questions how usable the algorithm is. In the extreme case, for example, an algorithm can just reject *every* subtyping problem. This is no doubt a sound and terminating algorithm, but just not very useful! Surely stare-at subtyping does not fall into this category but it is still useful to at least informally understand what stare-at subtyping can do. I will take a two steps to analyze the algorithm.

The first step is to compare it with the stare-at subtyping in $D_{<:}$, defined in Figure 4.8. Reviewing the rules, it is quite clear that stare-at subtyping in $D_{<:}$ is just a special case of $\mu DART$. This can be seen by matching rules one by one, and from the fact that stare-at subtyping in $\mu DART$ syntactically extends stare-at subtyping in $D_{<:}$. This means stare-at subtyping in $\mu DART$ admits everything that stare-at subtyping in $D_{<:}$ admits. If an interesting program can be written in $D_{<:}$ and all subtyping problems are admitted by stare-at subtyping in $D_{<:}$, then there is no need to worry about its equivalence in $\mu DART$.

The second step is to understand in what situation the removal of a path type fails an admissible subtyping relation. Browsing through the rules again, defined in Figure 6.4, it is obvious that only the recursive calls in SA-SEL1 and SA-SEL2 rules can witness such impact. In any other rules, for example, in the SA-ALL rule, the same Γ_1 and Γ_2 are passed into two recursive calls.

$$\frac{\Gamma_1 \gg S >: S' \ll \Gamma_2 \quad \Gamma_1; x : S \gg U <: U' \ll \Gamma_2; x : S'}{\Gamma_1 \gg \forall(x : S)U <: \forall(x : S')U' \ll \Gamma_2} \text{SA-ALL}$$

If a path type is removed in the first recursive call, the second recursive call receives no impact.

To enter SA-SEL1 and SA-SEL2 and witness the impact, the only way is to input a path type, $x.A$, and have it **Upcast** or **Downcast** to other types that eventually use $x.A$. Without loss of generality, let us focus on SA-SEL2 and **Upcast**. F-bounded quantification is one of these situations. If $x.A$ is F-bounded quantified, then its upper bound refers back to $x.A$. Consider a stare-at subtyping problem $\Gamma_1 \gg x.A <: U \ll \Gamma_2$. If $x : \mu(z : \{A : \bot..F(z.A)\})$ in Γ_1 where $F(z.A)$ is a type expression that uses $z.A$ (and REC-E will make it use $x.A$), then $\Gamma_1 \vdash_{\mu DARTS} x.A \nearrow F(x.A) \dashv \Gamma_1'$. If $U = F(x.A)$, then by reflexivity, stare-at subtyping admits this relation, namely $\Gamma_1 \gg x.A <: F(x.A) \ll \Gamma_2$ is provable by stare-at subtyping. Therefore, F-bounded comparison is admissible in stare-at subtyping.

However, in $\Gamma_1 \gg x.A <: F(F(x.A)) \ll \Gamma_2$, it requires showing $\Gamma_1' \gg F(x.A) <: F(F(x.A)) \ll \Gamma_2$ and therefore comparing $x.A$ and $F(x.A)$ after $x.A$ is removed, which is not admitted by the algorithm anymore.

Generally speaking, if a subtyping problem requires resolution of the same path type twice, then stare-at subtyping is not able to handle it. On the other hand, that also implies the relation between path types is more complex than F-bounded quantification. F-bounded quantification is already not a frequently seen technique in reality, so if a piece of code requires a more sophisticated technique, then such a relation between types is probably already very hard for people to understand. In the example above, instead of comparing $x.A$ and $F(F(x.A))$, one could just simplify $F(F(x.A))$ to $F(x.A)$ to assist the subtyping procedure to make a desirable decision.

6.11 An Alternative Treatment

Recall that the treatment here to ensure termination of stare-at subtyping is to remove a component from the syntax of the contexts. One obvious alternative is to maintain a *counter* for each path type for all μ types and decrement the counter by one when a path type is visited, instead of actually removing types from the context. When the counter hits zero, the path type can no longer be visited and is regarded as $\top$. The counter is like an access permission of its path type. One can see that these two approaches are essentially the same, except that the actual proof differs. This alternative approach does not need convertibility and $OPE_{\leadsto}$, but just $OPE_{<:}$. This is because the actual content of the context remains the same after all and the accounting of the counters means nothing in terms of the subtyping relation.

Another advantage of this alternative approach is that the initial values of counters can be parameterized. The removal approach by nature is equivalent to the case in which the initial values are one. However, by directly maintaining the counter, the values can be two, five, or any other number. This provides a very interesting mathematical device to talk about expressive power of the language, because as the initial values increase the expressive power also increases. I expect that the language reaches some "normal form" if this initial value is taken to be infinity.

6.12 Variable Typing

In this section, I will discuss a bi-directional variable typing algorithm. A bi-directional typing algorithm [Pierce and Turner, 2000, Odersky et al., 2001] operates in two modes: *checking mode* and *synthesis mode*. The checking mode checks if a given term has a given type, and the synthesis mode infers a type for a given term.

In the technical work, I have designed a bi-directional term typing algorithm for $\mu DART$. However, I postpone the discussion on the general bi-directional term typing to Chapter 7 and only focus on variable typing in this section in order to expose a particular complication introduced by μ types. For completeness, the definition of bi-directional term typing can be found in Appendix A.

The reason why variable typing deserves special attention is that the Rec-I and Rec-E rules have put variables in a special position. In particular, μ types do not participate in subtyping, but if a μ type is bound to a variable, then Rec-I and Rec-E can convert the μ type to a record type and back, and record types indeed have subtyping. μ types then rely on variables to interact with other types in subtyping, so variables have more interactions with types than any other terms. In fact, the convertibility relation defined in Section 6.9 is also based on this observation. Now let us consider the definition of bi-directional variable typing concretely.

Definition 6.8. *The bi-directional variable typing rules are defined in Figure 6.10.*

A variable type check judgment, $\Gamma \vdash_{\mu DARTS} x \overset{\leftarrow}{:} T$, has all of Γ, x and T as the inputs, and outputs true if the inputs satisfy the definition.

A variable synthesis judgment, $\Gamma \vdash_{\mu DARTS} x / V \overset{\rightarrow}{:} T$, has Γ, x and V as the inputs and T as the output.

The algorithm relies on **Exposure** and its dual **Imposure**, both adapted to $\mu DART$. Recall that **Exposure** finds the supertype of a path type that is not a path. Dually,

$$\frac{\Gamma \vdash_{\mu DARTS} U \Downarrow \mu(z : DS_z) \quad \Gamma \vdash_{\mu DARTS} x \overset{\leftarrow}{:} \{DS_x\}}{\Gamma \vdash_{\mu DARTS} x \overset{\leftarrow}{:} U}\ \textsc{Chk-Mu} \qquad \frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \mu(z : DS_z) \quad \Gamma \gg \{DS_x\} <: U \ll \Gamma}{\Gamma \vdash_{\mu DARTS} x \overset{\leftarrow}{:} U}\ \textsc{Chk-BindMu}$$

$$\frac{\Gamma \gg \Gamma(x) <: U \ll \Gamma}{\Gamma \vdash_{\mu DARTS} x \overset{\leftarrow}{:} U}\ \textsc{Chk-Sub} \qquad \frac{fv(\Gamma(x)) \cap V = \emptyset}{\Gamma \vdash_{\mu DARTS} x/V \overset{\rightarrow}{:} \Gamma(x)}\ \textsc{Syn-Stop}$$

$$\frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \mu(z : DS_z) \quad \Gamma \vdash_{\mu DARTS} \{DS_x\} \Uparrow_{V \setminus x} \{DS'_x\}}{\Gamma \vdash_{\mu DARTS} x/V \overset{\rightarrow}{:} \mu(z : DS'_z)}\ \textsc{Syn-Mu} \qquad \frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow_V U}{\Gamma \vdash_{\mu DARTS} x/V \overset{\rightarrow}{:} U}\ \textsc{Syn-Bind}$$

Figure 6.10: Definition of variable type checking and type synthesis

Imposure finds the subtype of a path type that is not a path. The concrete definitions are too verbose in this discussion of variable typing, so I put them in the appendix. Moreover, the algorithm relies on the **Promotion** operation. This operation finds a supertype of a given type so that a set of variables does not occur free in the result type. The specifications of these operations are listed below. Again, the definitions are found in Appendix A.

1. A **Exposure** judgment, $\Gamma \vdash_{\mu DARTS} S \Uparrow U$, has Γ and S as inputs and U as output. If $\Gamma \vdash_{\mu DARTS} S \Uparrow U$, $\Gamma \vdash_{\mu DART} S <: U$ and U is guaranteed not a path type.

2. A **Imposure** judgment, $\Gamma \vdash_{\mu DARTS} S \Downarrow U$, has Γ and S as inputs and U as output. If $\Gamma \vdash_{\mu DARTS} S \Downarrow U$, $\Gamma \vdash_{\mu DART} U <: S$ and U is guaranteed not a path type.

3. A **Promotion** judgment, $\Gamma \vdash_{\mu DARTS} S \Uparrow_V U$, has Γ, S and a set of variables V as inputs and U as output. If $\Gamma \vdash_{\mu DARTS} S \Uparrow_V U$, $\Gamma \vdash_{\mu DART} S <: U$ and U does not have any free variables in V.

The algorithm needs to be careful about μ types. For example, the Chk-Mu rule handles a case in which **Imposure** of U finds a μ type $\mu(z : DS_z)$[1]. A direct subtyping comparison between $\Gamma(x)$ and $\mu(z : DS_z)$ might fail, because, again, μ types do not participate in subtyping. However, applying Rec-E turns $\mu(z : DS_z)$ to a record type $\{DS_x\}$ which is more likely to have relation with $\Gamma(x)$. Note that if $\Gamma \vdash_{\mu DARTS} x \overset{\leftarrow}{:} \{DS_x\}$

[1]Due to α conversion, DS does not have z occur free.

indeed holds, then the connection between $\Gamma(x)$ and U cannot be expressed by subtyping, as it at least requires Rec-E.

The Chk-BindMu rule is dual to the Chk-Mu rule. It handles the case in which **Exposure** of $\Gamma(x)$ returns a μ type.

In the synthesis direction, the idea is to use **Promotion** to find a supertype so that unwanted free variables are removed (Syn-Bind). The complication comes when **Exposure** of $\Gamma(x)$ returns a μ type $\mu(z : DS_z)$ (Syn-Mu). In this case, $\mu(z : DS_z)$ is turned into a record type $\{DS_x\}$ before being passed into **Promotion**.

Soundness properties are quite straightforward.

Lemma 6.22. *(soundness) If* $\Gamma \vdash_{\mu DARTS} x \overset{\leftarrow}{:} T$*, then* $\Gamma \vdash_{\mu DART} x : T$*.*

Lemma 6.23. *(soundness) If* $\Gamma \vdash_{\mu DARTS} x/V \overset{\rightarrow}{:} T$*, then* $\Gamma \vdash_{\mu DART} x : T$*.*

The synthesis direction has one more soundness condition, because it needs to make sure the returned type does not have any free occurrences of the variables in V.

Lemma 6.24. *If* $\Gamma \vdash_{\mu DARTS} x/V \overset{\rightarrow}{:} T$*, then* $fv(T) \cap V = \emptyset$*.*

As shown above, due to μ types, variable typing requires much more additional consideration that is not required when handling other terms. In Chapter 7, I will show a further extension of variable typing to enlarge the typeable fragment.

Chapter 7

jDOT

In the previous chapters, I studied the algorithmic (sub)typing of some decidable fragments of $D_{<:}$, $D_{\wedge}$ and $\mu DART$. The study of the algorithmic (sub)typing has revealed several techniques to tackle type checking and subtyping decision problems due to the features of DOT. This chapter is meant to bring all these techniques together and present an overall result.

I will begin with a high level review of the algorithmic techniques introduced previously. This review points out some diadvantages of the previous methods. Then I will also review the Wadlerfest DOT calculus. It turns out there are a number of problems in Wadlerfest DOT that are either overlooked or ignored in the discussions of its soundness. Aggregating all the information, I propose $jDOT$ to be a proper alternative of DOT(j for *just-right*[1]). $jDOT$ simplifies DOT on one hand, and better matches the human intuition on the other hand. $jDOT$ is designed to be not only a formalization of Scala, but also friendly with algorithmic study.

I will also informally discuss the expressiveness and semantics of $jDOT$ and explain why this is a better alternative. I will show that the covariant list example from Amin et al. [2016] can be properly encoded in $jDOT$. Additionally, the bi-directional type assignment algorithm also admits this encoding of lists.

Finishing the technical work of this chapter, I consider the algorithmic typing problem of Dependent Object Types has been gracefully resolved. However, the same as $\mu DART$, I am still not able to prove the undecidability of $jDOT$, which is left as a regret and a piece of future work for interested researchers.

[1]Or for Jason's.

7.1 A Step Back: Algorithmic (Sub)typing Reconsidered

In the previous chapters, I described some problems encountered when designing the algorithmic (sub)typing of corresponding calculi and some techniques to tackle them. There has been much information, so it would be a good idea to pause for a second, and review the methods and what they resolve.

7.1.1 Subtyping as communication

In Chapter 4, I described the motivation to make subtyping decisions using two contexts, and modelled the decision procedure as a communication game between Alice and Bob. Ultimately, the reason for two contexts is to obtain more power from the substitutive nature of subtyping between dependent functions (explained in Section 4.7.2). As a result, the SA-All rule in stare-at subtyping is designed.

7.1.2 Backtracking due to $\wedge$-Traversal

In Chapter 5, I investigated the interactions between path dependent types and intersection types. It turns out that in order to decide subtyping involving intersection types, operations in stare-at subtyping need to have some capabilities of backtracking. This backtracking needs to exist not only in stare-at subtyping itself, but also all the way to every **$\wedge$-Traversal** instance in **Revealing**, **Upcast** and **Downcast**. This is because intersection types introduce uncertainty into the types and the algorithm has to introduce non-determinism to potentially exhaust all the possibilities. The algorithm terminates because the number of choices every **$\wedge$-Traversal** can make is finite and there is a measure showing that each subproblem is strictly smaller than the input problem.

7.1.3 Types as resources

In Chapter 6, I showed how to handle the interactions between path dependent types and μ types. Essentially the method is to treat types as resources, and each access of path types consumes some "types" as the cost. Since the input is finite, the overall algorithm must terminate. The soundness of the algorithm requires some consistent maintenance and the full details have been shown in the chapter. Unlike $D_\wedge$, $\mu DART$ does not require backtracking to handle μ types.

The investigation in Chapter 6 is somewhat different from the ones in Chapter 4 and Chapter 5. In particular, the method in Chapter 6 is entirely operational, whereas both $D_{<:}$ and $D_\wedge$ are shown to have declarative strong kernels w.r.t. which the algorithms are sound and complete. In other words, since the theories in $D_{<:}$ and $D_\wedge$ have agreed so well, they do not seem to have much room for adjustment, while $\mu DART$ still shows much potential for improvement.

My first dissatisfaction with the work on $\mu DART$ is the large number of operations. To name them all, the operations are **Exposure**, **Imposure**, **Exposure**$^\mu$, **Imposure**$^\mu$, **Revealing**, **Upcast**, **Downcast**, **Promotion**, **Demotion**, **Promotion**$^\mu$, **Demotion**$^\mu$, stare-at subtyping and bi-directional type assignment. Among them, I consider **Revealing**, **Upcast**, **Downcast** and stare-at subtyping necessary for the subtyping framework and **Promotion**, **Demotion** and bi-directional type assignment necessary for the typing framework. The rest of the operations look unnecessary. They were designed as a result of lack of understanding of μ types. However, after actually solving the problem, the solution can be revised and re-engineered so that it can be more compact.

Secondly, in DOT, all three of path types, intersection types and μ types will start to interact, and **Exposure**$^\mu$, for example, is no longer capable of handling the situation. To see the limitation of **Exposure**$^\mu$, consider the following context.

$$\Gamma = z : \{C : \bot..\mu(w : \{A : \bot..\bot\})\}; y : \{B : \bot..\top\}; x : y.B \wedge z.C$$

Revealing $x.A$ in this context requires resolution of $z.C$, which gives $\mu(w : \{A : \bot..\bot\})$ and the overall result should be $\bot$. This situation fits the Rv-Mu1 rule in **Revealing** of $\mu DART$. However, this will not work well. Since $z.C$ will push the context to the left up until z, and y is to the right of z, so y will be forgotten. To enlarge the decidable fragment, Rv-Mu1 appends the remaining declarations of the μ type to the returned context. In this case, the overall returned context becomes

$$\Gamma' = x : y.B \wedge \mu(w : \top)$$

Notice that y no longer exists in the context! This context is not well-formed. That means the treatment in $\mu DART$ is quite limited and cannot directly work with intersection types. The core issue is that **Revealing** so far relies on dropping bindings from the contexts to ease the termination proof and it is a safe treatment for the previous calculi, while now with both recursive types and intersection types, bindings can no longer easily be dropped, otherwise the well-formedness of the contexts might not be maintained.

To wrap up, the summary shows that stare-at subtyping and **$\wedge$-Traversal** are reasonable operations and shall be kept. Types as resources is still a reasonable idea to guarantee termination but needs to be engineered in a different form to accommodate the interactions between intersection types and μ types.

7.2 A Second Step Back: A Short Review of *DOT*

The next thing to do is to review the *DOT* calculus (defined in Definition 2.10). To tackle its complexity, it is a good idea to review its source of complexity and understand if that source of complexity is necessary. This section is devoted to have a high level overview of the *DOT* calculus and it identifies the places that can be better defined.

7.2.1 Uninterpretable types with μ

Though μ types have been studied to some extent in Chapter 6, μ types can only wrap over record types in $\mu DART$, which makes the μ construct still controllable. In *DOT*, μ types can wrap over any types, generating types that do not correspond to any types in Scala. Since one purpose of *DOT* is to model the Scala language, each type in the calculus ought to find its corresponding interpretation in the Scala language. If a type in *DOT* does not correspond to any type in Scala, this type should not be a part of the consideration to begin with. Consider the following bindings.

$$\begin{aligned}
&x : \mu(z : \bot)\\
&x : \mu(z : \mu(w : \{A : \bot..\top\}))\\
&x : \mu(z : \{A : \bot..z.B\} \wedge \mu(w : \{B : \bot..w.A\}))\\
&x : \mu(z : \forall(y : \top)\top)
\end{aligned}$$

Let me analyze them one by one.

1. Due to bad bounds, if $x : \bot$ is in the context, then every type is a subtype of any other (a proof can be found in Section 3.1). Now consider $x : \mu(z : \bot)$. Recall that μ types do not participate in subtyping, so this judgment $\mu(z : \bot) <: \mu(z : \top)$ cannot be proven in general. But if this type can be found bound to a variable in a context, everything changes.

$$\dfrac{\dfrac{}{\Gamma \vdash_{DOT} x : \mu(z : \bot)}\ \textsc{Var}}{\Gamma \vdash_{DOT} x : \bot}\ \textsc{Rec-E}$$

This is a proof that can be used to show $\mu(z : \bot)$ behaves the same as $\bot$ once bound to a variable! In general it is true for $\bot$ nested arbitrarily deep in μ's: $\mu(y : \mu(z : \bot))$, $\mu(y : \mu(z : \mu(w : \bot)))$, etc. This gives more complications, as it hints that the algorithm needs to handle arbitrarily nested μ's, while these μ-wrapped $\bot$'s clearly correspond to nothing in Scala.

2. The binding $x : \mu(z : \mu(w : \{A : \bot..\top\}))$ shows another aspect of nested μ types. In order to find out what $x.A$ means, the algorithm needs to look deep into the μ's. Ideally, this type should be the same as $\mu(z : \{A : \bot..\top\})$, but there is no formal relation to support this intuition.

3. The binding $x : \mu(z : \{A : \bot..z.B\} \wedge \mu(w : \{B : \bot..w.A\}))$ is a more sophisticated example. Imagine **Revealing** is applied to $x.B$. It needs to first know that the type member label B is in an intersection in a nested μ type, and knows that $w.A$ can become $x.A$ by applying Rec-E twice. Moreover, it needs to look back to the outer μ type again to find $\{A : \bot..z.B\}$ and turn $z.B$ to $x.B$. Finally, **Revealing** stops here because $x.B$ has been referred to twice, so the result should be $\top$.

   Notice that these path type bounds use different self references to indirectly refer to each other! The algorithm needs to be very intelligent about the method to handle the self reference, but this complication is unseen in Scala.

4. $x : \mu(z : \forall(y : \top)\top)$ wraps a function type in a μ. A μ wrapped function type is clearly not an intended type in *DOT* but nonetheless a valid one, which further constitutes more complications because it becomes a legitimate function type once Rec-E is applied.

7.2.2 How are objects encoded?

In the existing literature [Amin et al., 2016, Rapoport et al., 2017, Rompf and Amin, 2016, Amin et al., 2012], several encodings of objects in the corresponding calculi are shown as examples, but this problem in fact has never been formally studied. There are many problems once this issue is analyzed. In Wadlerfest *DOT*, to encode the following Scala trait, there are many available options.

```
trait Exists { type A ; def elem : this.A }
```

All of the followings are valid encodings.

1. $\mu(w : \{A : \bot..\top\} \wedge \{elem : w.A\})$
2. $\mu(w : \{A : \bot..\top\}) \wedge \mu(w : \{elem : w.A\})$
3. $\{A : \bot..\top\} \wedge \mu(w : \{elem : w.A\})$

Here I only listed the encodings that are worth looking into. Arbitrarily many more trivial encodings can be generated by wrapping μ's and putting the intersection in different places, so I omit them in this discussion.

Based on the Obj-I rule, the first option is taken as the primal choice. However, this option has two problems.

1. $\mu(w : \{A : \bot..\top\} \wedge \{elem : w.A\})$ and $\mu(w : \{elem : w.A\} \wedge \{A : \bot..\top\})$ are not related by subtyping (notice that the order of two fields is exchanged). This is very counter-intuitive, because normally we do not regard exchanging the order of fields in an object definition to have a fundamental impact.
2. Dropping a field also does not make a supertype: $\mu(w : \{A : \bot..\top\})$. The same as the previous point, this is due to lack of subtyping between μ types.

On the other hand, the next two encodings seem to avoid these two problems, due to the subtyping properties of intersection types.

Having many encodings of the same concept is problematic as the algorithmic design might be expected to cover all encodings.

7.2.3 Unexpected recursive path types

Consider the following (invalid) definition of a class in Scala.

```
class A extends B { trait B }
```

In this definition, the two `B`'s are literally the same trait. Namely, the class `A` is inheriting another trait `B` defined in its own definition. Needless to say, this definition is definitely bizarre, and it is reasonable to expect it not to compile. However, with the excessive expressiveness of μ types in *DOT*, this can be encoded as

$$\mu(w : w.B \wedge \{B : \bot..\top\})$$

This type is well-formed: after all there is no free variable in this encoding! Wadlerfest *DOT* has no predicate to reject this kind of type.

From an algorithmic design point of view, then, all the previous examples yield the following puzzling question.

Question 8. *What are the programs a candidate algorithm is OK to give up on? Or to what extent an algorithm can give up?*

To approach the answer to this question, we need to understand what the definition is meant to reflect, and from there, a new calculus can be engineered as an improvement.

7.2.4 Soundness proof guided definition

The hints lie in the soundness proof of *DOT*. The soundness proofs of both Wadlerfest *DOT* and OOPSLA *DOT* fundamentally rely on some knowledge of the runtime information of typing contexts. For example, in Wadlerfest *DOT*, Obj-I is the *only* rule to introduce an object type, in which it forces the bounds of a type declaration to be syntactically the same, and all objects are encoded in the form of $\mu(z : T)$ where T is an intersection of type and field declarations. When a step in operational semantics is taken, structural properties like weakening, narrowing and substitution are used to transform the context so that the μ types in the typing context must be of that form.

This proof technique lowers the difficulties of reasoning by concentrating the languages to only the runtime portion, and leads to the success of establishing canonical form theorems and the eventual soundness proofs. Details on soundness proofs can be found in Rompf and Amin [2016], Amin et al. [2016], Rapoport et al. [2017], Amin and Rompf [2017]. Though the concrete presentations differ, the central idea is described here.

The technique of using runtime information of typing contexts gives a proper explanation of why all of the problematic types shown in the examples above did not become major issues in all previous investigations. Generally speaking, more syntactic elements can be added to the calculus. As long as the form of runtime information remains the same, the essential portion of the soundness proofs will remain more or less familiar.

Taking the same observation in another direction, the definition of the language should not be too much larger than what the runtime portion is. The soundness proofs hint that the intended language should have the following characteristics.

1. There should be at most one layer of μ.

2. Only type and field declarations need to interact with μ types. In particular, path dependent types should not be wrapped in μ's.

3. Rules similar to REC-I and REC-E are still needed to ensure the calculus maintains behaviors close to the *DOT* calculi we currently have.

4. The intersection types can be taken advantage of to obtain some "free" subtyping relation between encodings of object types.

Languages richer than what is described by these points are just needlessly big, unnecessary, and not intended. These points give a clear direction on how to refine *DOT* to a desired form.

7.3 The Definition of *jDOT*

In this section, I will introduce *jDOT*, an alternative based on the observations described in the previous sections. In some sense *jDOT* is simpler because μ types are more constrained; in another sense, *jDOT* is more concentrated, because it uses intersection types to allow a simple subtyping relation between object types, which gets much closer to the human intuition.

Definition 7.1. *The abstract syntax of jDOT is shown in Figure 7.1, the typing rules are shown in Figure 7.2 and the subtyping rules are shown in Figure 7.3.*

In Figure 7.1, I highlighted the difference between *DOT* and *jDOT*. The same as in *μDART*, let bindings in *jDOT* have an optional ascription. On the side of types, instead of having a general μ construct, *jDOT* only has μ declarations. μ declarations permit type and field declarations in which the type components have access to a self reference variable, represented by w in the definition.

In the typing rules defined in Figure 7.2, the rules are almost the same as *DOT*. LET2 is added to ensure the let binding with an explicit type ascription is well typed. Consider the OBJ-I rule. This rule is not the same as the counterpart in *DOT*. To explain this rule, let us first see how *jDOT* represents object types.

Recall that in the previous section, I discussed the ambiguity of encodings of object types into *DOT*. The following trait definition is the example.

```
trait Exists { type A ; def elem : this.A }
```

$s, t, u ::=$		**Term**
	x	variable
	v	value
	$x.a$	selection
	$x\ y$	application
	$\text{let } x : T? = t \text{ in } u_x$	let binding
$v ::=$		**Value**
	$\lambda(x : T)t_x$	lambda
	$\nu(x : T)d_x$	object
$d ::=$		**Definition**
	$\{a = t\}$	field definition
	$\{A = T\}$	type definition
	$d_1 \wedge d_2$	aggregation

$D ::=$		**Declaration**
	$\{A : S..U\}$	type declaration
	$\{a : T\}$	field declaration
$\mu D ::=$		μ **Declaration**
	$\mu\{w.A : S_w..U_w\}$	
	$\mu\{w.a : T_w\}$	
$S, T, U ::=$		**Type**
	$\top$	top type
	$\bot$	bottom type
	$x.A$	path type
	D	Declaration
	μD	μ Declaration
	$\forall(x : S)U_x$	function
	$S \wedge U$	intersection

Figure 7.1: Abstract syntax of $jDOT$

In $jDOT$, this trait is canonically represented by the following type (up to commutations and associations of intersection types).

$$\mu\{w.A : \bot..\top\} \wedge \mu\{w.elem : w.A\}$$

I will discuss the advantages of this representation in greater detail once the rules are fully discussed. In Obj-I, the type T in the ν shall be an intersection of μ declarations. Otherwise the object definition would not be type checked. Notice that the result type of an object is just the same intersection of μ declarations in the ν. This is not the same as DOT, in which the corresponding rule requires an α renaming of T before type checking the object body. The rule delegates the concrete type checking to the definition typing rules. Def-Trm, Def-Typ and Def-And are the rules and they require an extra argument to the predicate: x, the variable in the context which is bound to the type of the object. This variable is recorded as a subscript under the colon (:). In the definition typing rules, w is the self reference of the object, carried over from the Obj-I rule.

Type Assignment

$$\frac{}{\Gamma \vdash_{jDOT} x : \Gamma(x)}\ \textsc{Var} \qquad \frac{\Gamma \vdash_{jDOT} t : S \quad \Gamma \vdash_{jDOT} S <: U}{\Gamma \vdash_{jDOT} t : U}\ \textsc{Sub}$$

$$\frac{\Gamma; x : S \vdash_{jDOT} t : U}{\Gamma \vdash_{jDOT} \lambda(x : S)t : \forall(x : S)U}\ \textsc{All-I} \qquad \frac{\Gamma \vdash_{jDOT} x : \forall(z : S)U_z \quad \Gamma \vdash_{jDOT} y : S}{\Gamma \vdash_{jDOT} x\ y : U_y}\ \textsc{All-E}$$

$$\frac{\Gamma; x : T \vdash_{jDOT} d :_x T \quad dom(d) \text{ is unique}}{\Gamma \vdash_{jDOT} \nu(w : T)d_w : T}\ \textsc{Obj-I} \qquad \frac{\Gamma \vdash_{jDOT} x : \{a : T\}}{\Gamma \vdash_{jDOT} x.a : T}\ \textsc{Obj-E}$$

$$\frac{\Gamma \vdash_{jDOT} x : S \quad \Gamma \vdash x : U}{\Gamma \vdash_{jDOT} x : S \wedge U}\ \textsc{And} \qquad \frac{\Gamma \vdash_{jDOT} x : D_x}{\Gamma \vdash_{jDOT} x : \mu D}\ \textsc{Rec-I} \qquad \frac{\Gamma \vdash_{jDOT} x : \mu D}{\Gamma \vdash_{jDOT} x : D_x}\ \textsc{Rec-E}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{jDOT} t : S \quad x \notin fv(U) \\ \Gamma; x : S \vdash_{jDOT} u : U\end{array}}{\Gamma \vdash_{jDOT} \text{let } x = t \text{ in } u : U}\ \textsc{Let1} \qquad \frac{\begin{array}{c}\Gamma \vdash_{jDOT} t : S \quad x \notin fv(U) \\ \Gamma; x : S \vdash_{jDOT} u : U\end{array}}{\Gamma \vdash_{jDOT} \text{let } x : S = t \text{ in } u : U}\ \textsc{Let2}$$

Object Definition Type Assignment

$$\frac{\Gamma \vdash_{jDOT} t_x : T_x}{\Gamma \vdash_{jDOT} \{a = t_w\} :_x \mu\{w.A : T_w\}}\ \textsc{Def-Trm}$$

$$\frac{}{\Gamma \vdash_{jDOT} \{A = T_w\} :_x \mu\{w.A : T_w..T_w\}}\ \textsc{Def-Typ}$$

$$\frac{\Gamma \vdash_{jDOT} d_w :_x S \quad \Gamma \vdash_{jDOT} d'_w :_x U}{\Gamma \vdash_{jDOT} d_w \wedge d'_w :_x S \wedge U}\ \textsc{Def-And}$$

Figure 7.2: Typing rules of $jDOT$

In the object definition typing rules, Def-And just recurs down to subproblems. Def-Typ is routine: it sets the bounds in the declaration to be identical. Def-Trm is almost routine, except that before the definition of the field is type checked, the self reference w needs to be substituted by the variable x in the context. This slight complication is needed because ν directly remembers the result type of the object.

Subtyping

$$\frac{}{\Gamma \vdash_{jDOT} T <: \top}\ \textsc{Top} \qquad \frac{}{\Gamma \vdash_{jDOT} \bot <: T}\ \textsc{Bot} \qquad \frac{}{\Gamma \vdash_{jDOT} T <: T}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash_{jDOT} S_2 <: S_1 \quad \Gamma \vdash_{jDOT} U_1 <: U_2}{\Gamma \vdash_{jDOT} \{A : S_1..U_1\} <: \{A : S_2..U_2\}}\ \textsc{Bnd} \qquad \frac{\Gamma \vdash_{jDOT} T_1 <: T_2}{\Gamma \vdash_{jDOT} \{a : T_1\} <: \{a : T_2\}}\ \textsc{Fld}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{jDOT} S_2 <: S_1 \\ \Gamma; x : S_2 \vdash_{jDOT} U_1 <: U_2\end{array}}{\Gamma \vdash_{jDOT} \forall(x : S_1)U_1 <: \forall(x : S_2)U_2}\ \textsc{All} \qquad \frac{\begin{array}{c}\Gamma \vdash_{jDOT} S <: T \\ \Gamma \vdash_{jDOT} T <: U\end{array}}{\Gamma \vdash_{jDOT} S <: U}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash_{jDOT} x : \{A : S..U\}}{\Gamma \vdash_{jDOT} S <: x.A}\ \textsc{Sel1} \qquad \frac{\Gamma \vdash_{jDOT} x : \{A : S..U\}}{\Gamma \vdash_{jDOT} x.A <: U}\ \textsc{Sel2} \qquad \frac{\begin{array}{c}\Gamma \vdash_{jDOT} T <: S \\ \Gamma \vdash_{jDOT} T <: U\end{array}}{\Gamma \vdash_{jDOT} T <: S \wedge U}\ \textsc{And-I}$$

$$\frac{}{\Gamma \vdash_{jDOT} S \wedge U <: S}\ \textsc{And-E1} \qquad \frac{}{\Gamma \vdash_{jDOT} S \wedge U <: U}\ \textsc{And-E2}$$

Figure 7.3: Subtyping rules of *jDOT*

The subtyping rules are defined in Figure 7.3, and they are literally the same as *DOT* and therefore do not need more discussion.

As hinted previously, using intersections of μ declarations to represent object types overcomes a number of problems found in *DOT*.

1. Objects now have simple subtyping. Subtyping between objects is obtained freely due to intersection types. This means the object types now can freely commute, duplicate and re-associate their declarations, and obtain equivalent types. Dropping fields now obtains a supertype. This gets much closer to the human intuition. However, notice that there is no rule to capture subtyping between μ declarations, so subtyping between μ declarations remain impossible.

2. μ now is a well controlled language construct. Unlike the examples in Section 7.2, μ is restricted to have only one layer, and only interacts with type and field declarations. This simplifies the language and makes it much more understandable.

These observations give more confidence to work on *jDOT* instead of *DOT*. An interesting question to ask is whether this language is sound. I conjecture so.

Conjecture 6. *$jDOT$ is sound.*

Note that this conjecture is promising. Its soundness is not a direct consequence of the soundness of DOT, because $jDOT$ changes the primal representation of object types which means this language is not a syntactic subset of DOT. However, the essential components remain similar and therefore I speculate that following Rapoport et al. [2017] can easily establish its soundness proof. The actual verification is left as future work.

Moreover, for the same reason as $\mu DART$ and DOT, I am not able to establish the undecidability proof of $jDOT$. In the same spirit, its undecidability is conjectured.

Conjecture 7. *$jDOT$ typing and subtyping are undecidable.*

A rigorous proof is left as future work.

7.4 Refining Revealing

In Section 7.1, I drew the conclusion that there is not much room to improve what is in $D_\wedge$, but things in $\mu DART$ can be better reworked. The biggest problem I consider in $\mu DART$ is that the number of operations is too large. In this section, I will refine **Revealing** so that all unnecessary operations can be dropped.

Within the current design of stare-at subtyping, **Revealing** is no doubt in the core of all versions. Therefore, to improve the algorithm, **Revealing** is the first and the most important operation to look into. In $\mu DART$'s **Revealing** (defined in Figure 6.6), **Exposure**$^\mu$ is used because μ types are considered as cyclic contexts, and **Exposure**$^\mu$ is an operation motivated by that observation. This view of μ types has two problems.

1. It directly leads to four operations: **Exposure**$^\mu$, **Imposure**$^\mu$, **Promotion**$^\mu$ and **Demotion**$^\mu$.
2. It fails to adapt to the change in $jDOT$. A counterexample was given in Section 7.1, in which **Revealing** fails to maintain the well-formedness of the returned context.

This means regarding μ types as cyclic contexts is an approach limited to $\mu DART$, and that motivates another understanding of μ types that has not been thought of. This understanding is hidden because the μ in μ types means recursion and μ types are intended to represent object types. To see how μ types can be reinterpreted, Rec-I and Rec-E give the first hint (to make the idea more explicit, let us just consider a particular type declaration).

$$\frac{\Gamma \vdash_{jDOT} x : \{A : \bot..x.B\}}{\Gamma \vdash_{jDOT} x : \mu\{w.A : \bot..w.B\}}\ \textsc{Rec-I} \qquad \frac{\Gamma \vdash_{jDOT} x : \mu\{w.A : \bot..w.B\}}{\Gamma \vdash_{jDOT} x : \{A : \bot..x.B\}}\ \textsc{Rec-E}$$

These two rules come in a pairm and are forms of constructing and eliminating μ types. When a μ type is found bound to some binder x in the context, μ is removed and the self reference w is replaced by the binder x. This replacement makes the self reference w look like a *hole*, which is to be filled in by the binder of the μ declaration. The following declaration makes the hole explicit.

$$\{A : \bot..[\,].B\}$$

where [ ] denotes a hole.

The intention is to consider a template of type declarations in which there are holes of a variable. Being bound to a binder in the context grants the permission for the binder to fill in these holes, as denoted by the Rec-E rule. The Rec-I rule is the dual operation in which the binder *retreats* and holes are formed. The μ construct is just a wrapper to make sure that a declaration with holes is well-formed.

This view removes the special position of μ types in the subtyping procedure, and enables a thought different from one emphasizing its cyclic nature. The point here is just to make sure there are binders to fill in the holes in μ types. If **Revealing** can already achieve this, there is no need for **Exposure**$^\mu$ anymore. The solution is to strengthen **Revealing** so that it can achieve what is meant to be done by **Exposure**$^\mu$. Surely, well-formedness of returned contexts needs to be taken into account.

Definition 7.2. ***Revealing*** *and* $\wedge$***-Traversal*** *are defined in Figure 7.4.*

1. *For a judgment $S \mapsto U \,\&\, T$, S is the input type and U and T are the output types.*
2. *A* ***Revealing*** *judgment $\Gamma \vdash_{jDOTS} S \Uparrow U \dashv \Gamma'$ has Γ and a type S as inputs and Γ' and a type U as outputs.*

Definition 7.3. ***Upcast*** *and* ***Downcast*** *is defined in Figure 7.5.*

1. *An* ***Upcast*** *judgment $\Gamma \vdash_{jDOTS} x.A \nearrow U \dashv \Gamma'$ has Γ, a variable x and a type member label A as inputs and Γ' and a type U as outputs.*

∧-Traversal

$$\frac{T \text{ is not } \wedge}{T \mapsto T \mathbin{\&} \top}\ \textsc{At-Found} \qquad \frac{S \mapsto S' \mathbin{\&} T}{S \wedge U \mapsto S' \mathbin{\&} T \wedge U}\ \textsc{At-Left}$$

$$\frac{U \mapsto U' \mathbin{\&} T}{S \wedge U \mapsto U' \mathbin{\&} S \wedge T}\ \textsc{At-Right}$$

Revealing

$$\frac{T \text{ is not a path}}{\Gamma \vdash_{jDOTS} T \Uparrow T \dashv \Gamma}\ \textsc{Rv-Stop} \qquad \frac{}{\Gamma \vdash_{jDOTS} T \Uparrow \top \dashv \bullet}\ \textsc{Rv-Top}^*$$

$$\frac{T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uparrow T_2 \dashv \Gamma_1' \quad T_2 \mapsto \bot \mathbin{\&} T_3}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow \bot \dashv \Gamma_1'; x : T_1 \wedge T_3; \Gamma_2}\ \textsc{Rv-Bot}$$

$$\frac{\begin{array}{c} T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uparrow T_2 \dashv \Gamma_1' \\ T_2 \mapsto \{A : S..U\} \mathbin{\&} T_3 \quad U \mapsto U_0 \mathbin{\&} U_1 \quad \Gamma_1' \vdash_{jDOTS} U_0 \Uparrow U_2 \dashv \Gamma_1'' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow U' \dashv \Gamma_1''; x : T_1 \wedge T_3; \Gamma_2}\ \textsc{Rv-Bnd}$$

$$\frac{\begin{array}{c} T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uparrow T_2 \dashv \Gamma_1' \\ T_2 \mapsto \mu\{w.A : S..U_w\} \mathbin{\&} T_3 \quad U_x \mapsto U_0 \mathbin{\&} U_1 \quad \Gamma_1'; x : T_1 \wedge T_3 \vdash_{jDOTS} U_0 \Uparrow U_2 \dashv \Gamma_1'' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow U' \dashv \Gamma_1''; \Gamma_2}\ \textsc{Rv-Mu}$$

Figure 7.4: Definitions of **∧-Traversal** and **Revealing**

2. ***Downcast*** *judgment* $\Gamma \vdash_{jDOTS} x.A \searrow U \dashv \Gamma'$ *is defined similarly as* ***Upcast****.*

The definition of **Revealing** might appear to be almost the same as one of $D_\wedge$ but the points are subtle, so let us walk through the rules one by one. Before that, let us take a look at the new definition of **∧-Traversal**.

In *jDOT*, **∧-Traversal** not only selects a non-intersection type, like in $D_\wedge$, but also returns the rest of the intersection that has not been selected. From At-Left and At-Right, the unselected types in the recursive calls are intersected again before being returned. This treatment can ensure that **∧-Traversal** does not lose any information. This is guaranteed by the following lemma.

$$\frac{}{\Gamma \vdash_{jDOTS} x.A \nearrow \top \dashv \bullet}\ \textsc{U-Top}^* \qquad \frac{}{\Gamma \vdash_{jDOTS} x.A \searrow \bot \dashv \bullet}\ \textsc{D-Bot}^*$$

$$\frac{T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uuparrow T_2 \dashv \Gamma_1' \quad T_2 \mapsto \bot \mathbin{\&} T_3}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \nearrow \bot \dashv \Gamma_1'; x : T_1 \wedge T_3; \Gamma_2}\ \textsc{U-Bot}$$

$$\frac{T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uuparrow T_2 \dashv \Gamma_1' \quad T_2 \mapsto \bot \mathbin{\&} T_3}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \searrow \top \dashv \Gamma_1'; x : T_1 \wedge T_3; \Gamma_2}\ \textsc{D-Top}$$

$$\frac{T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uuparrow T_2 \dashv \Gamma_1' \quad T_2 \mapsto \{A : S..U\} \mathbin{\&} T_3}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \nearrow U \dashv \Gamma_1'; x : T_1 \wedge T_3; \Gamma_2}\ \textsc{U-Bnd}$$

$$\frac{T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uuparrow T_2 \dashv \Gamma_1' \quad T_2 \mapsto \{A : S..U\} \mathbin{\&} T_3}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \searrow S \dashv \Gamma_1'; x : T_1 \wedge T_3; \Gamma_2}\ \textsc{D-Bnd}$$

$$\frac{T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uuparrow T_2 \dashv \Gamma_1' \quad T_2 \mapsto \mu\{w.A : S..U_w\} \mathbin{\&} T_3}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \nearrow U_x \dashv \Gamma_1'; x : T_1 \wedge T_3; \Gamma_2}\ \textsc{U-Mu}$$

$$\frac{T \mapsto T_0 \mathbin{\&} T_1 \quad \Gamma_1 \vdash_{jDOTS} T_0 \Uuparrow T_2 \dashv \Gamma_1' \quad T_2 \mapsto \mu\{w.A : S_w..U\} \mathbin{\&} T_3}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \searrow S_x \dashv \Gamma_1'; x : T_1 \wedge T_3; \Gamma_2}\ \textsc{D-Mu}$$

Figure 7.5: Definitions of **Upcast** and **Downcast**

Lemma 7.1. *If $S \mapsto U \mathbin{\&} T$, the following are true.*

1. $\Gamma \vdash_{jDOT} S <: U \wedge T$.
2. $\Gamma \vdash_{jDOT} U \wedge T <: S$.

$\wedge$-Traversal remains non-deterministic and therefore requires backtracking as in $D_\wedge$ if regarded as an algorithm. This has been discussed in Chapter 5.

Knowing **$\wedge$-Traversal**, **Revealing** can be understood.

1. Rv-Stop and Rv-Top are routine.
2. (Rv-Bot) The same as $D_\wedge$, if **$\wedge$-Traversal** can select $\bot$ from the result type of the first recursive call, then the result can just be $\bot$. However, the returned context is no

longer empty; in fact the returned context still remembers all binders from the input context. **Revealing** no longer attempts to shrink the length of the context. The reason for this will become clear in the RV-MU rule. At the very least, **Revealing** consumes the $\bot$ type from the second $\wedge$**-Traversal** call, and just makes $x : T_1 \wedge T_3$ in the returned context. This aligns with the idea of types as resources: consuming types to ensure termination.

3. (RV-BND) Alternatively, $\wedge$**-Traversal** might select a type declaration from the result type of the first recursive call. Following $D_\wedge$, U is selected by $\wedge$**-Traversal** again to avoid having an additional case of selecting a path type from T_2. Then U_0 is revealed again, and the overall returned context also replaces x's type with $T_1 \wedge T_3$.

4. (RV-MU) This rule is the most complicated one. When $\wedge$**-Traversal** selects a μ declaration from T_2, it first fills in the hole in U_w by substituting w with x. Then $\wedge$**-Traversal** selects U_0 from U_x. The second recursive call is curious: since now U_0 might refer to x, to ensure U_0 is closed, $x : T_1 \wedge T_3$ is appended into the context. In the same spirit, the μ declaration where U_w resides has been consumed and is gone. Therefore, even if Γ_2 is empty, the input of the second recursive call is still smaller than the original input. Moreover, since **Revealing** does not remove bindings from the context anymore, x is guaranteed to exist in Γ_1'', and that is the reason why the overall returned context does not mention x.

By comparing RV-MU with **Exposure**$^\mu$, it is easier to understand how this rule is compact enough to entirely replace **Exposure**$^\mu$. Assuming $U_0 = x.B$, then the second recursive call needs to do more work to find a non-path supertype. Even further assume $x.B$ requires **Revealing** of $x.C$, and $x.C$ requires **Revealing** of $x.D$, and so forth. The second recursive call in RV-MU iterates and functions just like **Exposure**$^\mu$. Since every iteration of the second recursive call in the RV-MU rule consumes a part of the type bound to x, there must be an end of the iterations and the recursion either continues to the left of the context or stops. In either way, the operation is guaranteed to terminate.

In **Exposure**$^\mu$, on the other hand, this looping is encoded as a separate operation. This shows **Revealing** in *jDOT* is a more compact and powerful operation.

On the flip side, though, this compactness come with some cost. In all previous versions of **Revealing**, the termination argument was very straightforward, because each recursive call must decrease the number of bindings in the context and this serves as a clear indication of termination. This is no longer true in *jDOT*. In Section 7.6, I will show how the termination argument for this **Revealing** is established.

Once **Revealing** is understood, **Upcast** and **Downcast** are straightforward. They again remain moderate wrappers of **Revealing** and simply take a part of the logic from **Revealing** and find the result types in the right direction with consistent treatment in the returned contexts.

7.5 Examples of Revealing

Though I have informally discussed the compactness of **Revealing**, it might still not be clear why Rv-Mu is well engineered. The following are some examples for this purpose.

7.5.1 Maintenance of well-formedness

This is a simple example showing how **Revealing** maintains the well-formedness of the contexts. Consider the same example shown in Section 7.1:

$$\Gamma = z : \{C : \bot..\mu(w : \{A : \bot..\bot\})\}; y : \{B : \bot..\top\}; x : y.B \wedge z.C$$

I use $\Gamma[..y]$ to denote truncation of context up to the variable y (inclusive). For example, $\Gamma[..y]$ means Γ dropping x only. **Revealing** $x.A$ in this context gives

$$\dfrac{\begin{array}{c} y.B \wedge z.C \mapsto z.C \;\&\; y.B \wedge \top \\ \Gamma[..y] \vdash_{jDOTS} z.C \Uparrow \mu\{w.A : \bot..\bot\} \dashv z : \top \wedge \top; y : \{B : \bot..\top\} \\ \mu\{w.A : \bot..\bot\} \mapsto \mu\{w.A : \bot..\bot\} \;\&\; \top \end{array}}{\Gamma \vdash_{jDOTS} x.A \Uparrow \bot \dashv z : \top \wedge \top; y : \{B : \bot..\top\}; x : y.B \wedge \top \wedge \top}\ \text{Rv-Mu}$$

In the derivation above, there are many $\top$'s. This is the result of following the rules rigorously. The $\top$'s come from the second output of the At-Found rule. Both recursive calls should be clear and their derivations are omitted. In the returned context, x binds to $y.B \wedge \top \wedge \top$, but now y remains untouched in the returned context, and that fixes the previous problem of **Exposure**$^{\mu}$.

7.5.2 Handling object types

Recall that object types in *jDOT* are represented by intersections of μ declarations. Rv-Mu relies on the second recursion to loop within the object to take over **Exposure**$^{\mu}$'s responsibility as in $\mu DART$. An example is designed to demonstrate the behavior of this rule.

$$\Gamma = y : \{D : \bot..\mu\{w.B : \bot..w.C\}\}; x : y.D \wedge \mu\{w.A : \bot..w.B\} \wedge \mu\{w.C : \bot..\forall(z : \top)\top\}$$

Consider **Revealing** $x.A$ in this context. The intended output type is $\forall(z : \top)\top$. The problem is made a bit harder as the type of x refers to y. This pattern is not rare as it can be used to encode inheritance. I will use ? to represent unresolved return. The following is a partial execution trace. Since x has type $\mu\{w.A : \bot..w.B\}$, the second recursive call has $x.B$ as the input type.

$$\frac{\begin{array}{c}\Gamma(x) \mapsto \mu\{w.A : \bot..w.B\} \;\&\; y.D \wedge \top \wedge \mu\{w.C : \bot..\forall(z : \top)\top\} \\ \mu\{w.A : \bot..w.B\} \mapsto \mu\{w.A : \bot..w.B\} \;\&\; \top \quad x.B \mapsto x.B \;\&\; \top \\ \Gamma[..y]; x : y.D \wedge \top \wedge \mu\{w.C : \bot..\forall(z : \top)\top\} \wedge \top \vdash_{jDOTS} x.B \Uparrow ? \dashv ?\end{array}}{\Gamma \vdash_{jDOTS} x.A \Uparrow ? \dashv ?}\ \textsc{Rv-Mu}$$

Rv-Stop applies in the first recursive call, and is omitted for conciseness. The second recursive call needs to resolve $x.B$. The only type label B is in y, so **Revealing** needs to look into y.

Let $\Gamma' = \Gamma[..y]; x : y.D \wedge \top \wedge \mu\{w.C : \bot..\forall(z : \top)\top\} \wedge \top$

$$\frac{\begin{array}{c}\Gamma'(x) \mapsto y.D \;\&\; \top \wedge \top \wedge \mu\{w.C : \bot..\forall(z : \top)\top\} \wedge \top \\ \Gamma'[..y] \vdash_{jDOTS} y.D \Uparrow \mu\{w.B : \bot..w.C\} \dashv y : \top \wedge \top \ (\textsc{Rv-Bnd}) \\ \mu\{w.B : \bot..w.C\} \mapsto \mu\{w.B : \bot..w.C\} \;\&\; \top \quad x.C \mapsto x.C \;\&\; \top \\ y : \top \wedge \top; x : \top \wedge \top \wedge \mu\{w.C : \bot..\forall(z : \top)\top\} \wedge \top \wedge \top \vdash_{jDOTS} x.C \Uparrow ? \dashv ?\end{array}}{\Gamma' \vdash_{jDOTS} x.B \Uparrow ? \dashv ?}\ \textsc{Rv-Mu}$$

Notice that the input context in the second recursive call is almost wiped clean, except for the μ declaration with type label C. The visited labels are replaced by $\top$'s. It might seem these $\top$'s contribute to the size of the context, but this is not an issue. In the next section, I will show how to make the termination argument ignore these $\top$'s.

Now **Revealing** has reached $x.C$, which is still a path type. The next step is to retrieve

$$\frac{}{\bullet \preceq \bullet}\ \textsc{Se-Nil} \qquad \frac{\Gamma \preceq \Gamma' \quad \Gamma \vdash_{jDOT} S <: U}{\Gamma; x : S \preceq \Gamma'; x : U}\ \textsc{Se-Cons}$$

Figure 7.6: Definition of sub-environment

the result from the one last remaining μ declaration in the context.

$$\text{Let } \Gamma'' = y : \top \wedge \top; x : \top \wedge \top \wedge \mu\{w.C : \bot..\forall(z : \top)\top\} \wedge \top \wedge \top$$

$$\frac{\begin{array}{c}\Gamma''(x) \mapsto \mu\{w.C : \bot..\forall(z : \top)\top\}\ \&\ \top \wedge \top \wedge \top \wedge \top \wedge \top \\ \mu\{w.C : \bot..\forall(z : \top)\top\} \mapsto \mu\{w.C : \bot..\forall(z : \top)\top\}\ \&\ \top \\ \forall(z : \top)\top \mapsto \forall(z : \top)\top\ \&\ \top\end{array}}{\Gamma'' \vdash_{jDOTS} x.C \Uparrow \forall(z : \top)\top \dashv y : \top \wedge \top; x : \top \wedge \top \wedge \top \wedge \top \wedge \top \wedge \top}\ \textsc{Rv-Mu}$$

Finally, $x.C$ is revealed to a function type which is not a path type. The whole operation finishes and it is OK to rewind the call stack. Notice that now the context has been fully wiped clean but still contains all the binders from the original context.

$$\cfrac{\cfrac{\cfrac{\text{... as shown above}}{\Gamma'' \vdash_{jDOTS} x.C \Uparrow \forall(z : \top)\top \dashv y : \top \wedge \top; x : \top \wedge \top \wedge \top \wedge \top \wedge \top \wedge \top}\ \textsc{Rv-Mu}}{\Gamma' \vdash_{jDOTS} x.B \Uparrow \forall(z : \top)\top \dashv y : \top \wedge \top; x : \top \wedge \top \wedge \top \wedge \top \wedge \top \wedge \top}\ \textsc{Rv-Mu}}{\Gamma \vdash_{jDOTS} x.A \Uparrow \forall(z : \top)\top \dashv y : \top \wedge \top; x : \top \wedge \top \wedge \top \wedge \top \wedge \top \wedge \top}\ \textsc{Rv-Mu}$$

7.6 Properties of Operations

After informally discussing the behaviors of **Revealing**, it is time to formally examine the properties. Since **Revealing** has been entirely re-engineered, there are more properties that need to be proven to ensure its correctness.

First, a relation between contexts is needed to express the relation between the input and output contexts of **Revealing**.

Definition 7.4. *The sub-environment relation between contexts, denoted by $\Gamma \preceq \Gamma'$, is defined in Figure 7.6.*

Theorem 7.2. *(soundness) If* $\Gamma \vdash_{jDOTS} S \Uparrow U \dashv \Gamma'$*, then* $\Gamma \vdash_{jDOT} S <: U$*.*

Theorem 7.3. *If* $\Gamma \vdash_{jDOTS} S \Uparrow U \dashv \Gamma'$*, then* $\Gamma \preceq \Gamma'$*.*

Proof. These two theorems are mutually dependent. They need to be proven together. □

The following lemma ensures **Revealing** does not remove bindings from the contexts.

Lemma 7.4. *If* $\Gamma \preceq \Gamma'$*, then* $dom(\Gamma) = dom(\Gamma')$*.*

Lemma 7.5. *If* $\Gamma \vdash_{jDOTS} S \Uparrow U \dashv \Gamma'$*, then* U *is not a path type.*

The well-formedness of outputs can be shown by the next theorem.

Theorem 7.6. *If* $\Gamma \vdash_{jDOTS} S \Uparrow U \dashv \Gamma'$ *and* Γ *and* S *are well-formed, so are* Γ' *and* U*. In particular,* $fv(U) \subseteq dom(\Gamma') = dom(\Gamma)$*.*

Visually, one of obvious distinctions between **Revealing** in *jDOT* and other **Revealing**s is that **Revealing** in *jDOT* does not lose bindings in the contexts anymore. The reason why other **Revealing**s do this is to ease the termination argument. Specifically, all previous **Revealing**s can be shown to terminate by arguing the length of the input context shrinks. The termination argument in *jDOT* is no longer that straightforward, but still remains purely syntactical, following the highest level philosophy of this thesis.

To make the termination argument as smooth as possible for the future, I will need two slightly different measures, and take advantage of their combination.

Definition 7.5. *The accounting measure* M_A *of types and contexts, and the structural measure* M_S *of types are defined by the following equations.*

$$\begin{aligned}
M_A(\top) &= 0 & M_S(\top) &= 1\\
M_A(\bot) &= 0 & M_S(\bot) &= 1\\
M_A(S \wedge U) &= M_A(S) + M_A(U) & M_S(S \wedge U) &= 1 + M_S(S) + M_S(U)\\
M_A(x.A) &= 2 & M_S(x.A) &= 2\\
M_A(\forall(x : S)U) &= 1 + M_A(S) + M_A(U) & M_S(\forall(x : S)U) &= 1 + M_S(S) + M_S(U)\\
M_A(D) &= 1 + M_A(D) & M_S(D) &= 1 + M_S(D)\\
M_A(\mu D) &= 2 + M_A(D) & M_S(\mu D) &= 2 + M_S(D)\\
M_A(\{A : S..U\}) &= M_A(S) + M_A(U) & M_S(\{A : S..U\}) &= M_S(S) + M_S(U)\\
M_A(\{a : T\}) &= M_A(T) & M_S(\{a : T\}) &= M_S(T)
\end{aligned}$$

$$M_A(\Gamma) = \sum_{x:T\in\Gamma} M_A(T)$$

Comparing these two measures,

1. Note that there is no need to define the structural measure M_S for contexts.
2. These two measures are very close, except for the first three equations. Measures in previous chapters are closer to the structural measure. The accounting measure does not assign weights to $\top$ and $\bot$, and simply adds the weights of component types of intersections, while the structural measure gives additional weight to the intersection.

These characteristics make $M_S(T) \geq M_A(T)$, and the termination argument of stare-at subtyping succeeds by taking advantage of their difference. To show the termination of **Revealing**, M_S is still not needed. Its role becomes important once stare-at subtyping is discussed.

Definition 7.6. *Define the combined accounting measure*

$$M_A(\Gamma, T) = M_A(\Gamma) + M_A(T)$$

It can be shown that

Lemma 7.7. *If* $\Gamma \vdash_{jDOTS} S \Uparrow U \dashv \Gamma'$*, then* $M_A(\Gamma, S) \geq M_A(\Gamma', U)$*.*

Theorem 7.8. ***Revealing*** *terminates as a non-deterministic algorithm.*

Proof. The measure is $M_A(\Gamma, S)$. It is non-deterministic because of $\wedge$-**Traversal**, for the same reason as $D_\wedge$. This is OK because the choice to make is always finite so backtracking is guaranteed to terminate. $\square$

The properties of **Upcast** and **Downcast** are easier to prove than those of **Revealing**.

Lemma 7.9. *The following hold.*

1. *If* $\Gamma \vdash_{jDOTS} x.A \nearrow (\searrow) T \dashv \Gamma'$*, then* $\Gamma' \preceq \Gamma$*.*
2. *If* $\Gamma \vdash_{jDOTS} x.A \nearrow T \dashv \Gamma'$*, then* $\Gamma \vdash_{jDOT} x.A <: T$*.*
3. *If* $\Gamma \vdash_{jDOTS} x.A \searrow T \dashv \Gamma'$*, then* $\Gamma \vdash_{jDOT} T <: x.A$*.*

Upcast and **Downcast** strictly decrease the accounting measure.

Lemma 7.10. *If* $\Gamma \vdash_{jDOTS} x.A \nearrow (\searrow) T \dashv \Gamma'$*, then* $M_A(\Gamma, x.A) > M_A(\Gamma', T)$*.*

Extending *Figure* 5.4 with

$$\frac{}{\Gamma_1 \gg \mu D <: \mu D \ll \Gamma_2}\ \text{SA-M}\textsc{u}$$

Figure 7.7: Definition of stare-at subtyping

7.7 Stare-at Subtyping

Once **Revealing** is settled, stare-at subtyping is just routine.

Definition 7.7. *The stare-at subtyping is defined in Figure 7.7.*

For a stare-at subtyping judgment $\Gamma_1 \gg S <: U \ll \Gamma_2$, all four places are the inputs, and it outputs true if inputs satisfy its definition.

Since μ types do not participate in the subtyping relation, the only obvious rule to apply is reflexivity. SA-Mu is added to reflect that. Otherwise, stare-at subtyping of $jDOT$ is the same as the one of $D_\wedge$. The same as for $D_\wedge$, the execution order of stare-at subtyping is not obvious. The execution order has been discussed in detail in Section 5.4, and the situation in $jDOT$ is identical. In favor of conciseness, I omit the discussion here.

Theorem 7.11. *Stare-at subtyping is reflexive.*

$$\Gamma_1 \gg T <: T \ll \Gamma_2$$

Theorem 7.12. *(soundness of stare-at subtyping) If $\Gamma_1 \gg S <: U \ll \Gamma_2$, $\Gamma \preceq \Gamma_1$ and $\Gamma \preceq \Gamma_2$, then $\Gamma \vdash_{jDOT} S <: U$.*

Theorem 7.13. *If $\Gamma_1 \gg S <: U \ll \Gamma$, then $\Gamma \vdash_{jDOT} S <: U$.*

The termination argument of stare-at subtyping is worth discussing.

Theorem 7.14. *Stare-at subtyping terminates as a non-deterministic algorithm.*

Proof. For a stare-at subtyping judgment $\Gamma_1 \gg S <: U \ll \Gamma_2$, the measure is the lexicographic order of $M_A(\Gamma_1, S) + M_A(\Gamma_2, U)$ and $M_S(S) + M_S(U)$.

Here, the accounting measure can be seen strictly decreasing in almost all cases, except for those related to intersection types, SA-Left1, SA-Left2 and SA-Right. For these rules, the accounting measure might not decrease. However, in all these cases, the structural measure does decrease. Notice that the accounting measure must not increase in those cases. Therefore, it is shown that all cases in stare-at subtyping handle smaller problems than the input ones. □

Similarly, since the choices are finite and the recursive problems are shown to always be smaller, stare-at subtyping terminates.

At this point, the algorithmic subtyping problem of $jDOT$ has been resolved.

7.8 Variable Typing

Starting from this section, I will start the discussion of the bi-directional type checking algorithm in $jDOT$. In Section 6.12, I discussed why variable typing is special and deserves more focus. I also pointed out that variable typing in $\mu DART$ is not satisfactory, and this is fundamentally improved in $jDOT$. Notice that the situation in $jDOT$ is more complicated than in $\mu DART$: the Rec-I and Rec-E rules which have existed in $\mu DART$ make variables special, and the And rule allows further flexibility in typing behavior. In $\mu DART$, there are only six rules in total for variable type assignment. In $jDOT$, with intersection types, more rules are needed.

7.8.1 Variable type check

Since the algorithm is bi-directional, there are two directions: the check direction and the synthesis direction. I choose to first describe the check direction, because its form is very close to stare-at subtyping. Recall that in Chapter 6, to handle the complication of Rec-I and Rec-E, I introduced a concept called *convertibility*. The check direction is to check if two types are convertible provided a variable.

Definition 7.8. *The definitions of the* ***Conversion*** *operation and variable type check $jDOT$ is defined in Figure 7.8.*

For a ***Conversion*** *judgment $\Gamma_1 \rhd S \overset{x}{\Rrightarrow} U \lhd \Gamma_2$ and a variable type check judgment $\Gamma \vdash_{jDOTS} x \overset{\leftarrow}{:} T$, all components are inputs and they output true if there is a derivation satisfies the definition.*

The goal can be shown by this theorem.

Conversion

$$\frac{}{\Gamma_1 \rhd x.A \overset{y}{\Rrightarrow} x.A \lhd \Gamma_2}\ \textsc{Cv-Refl} \qquad \frac{\Gamma_1 \gg S <: U \ll \Gamma_2}{\Gamma_1 \rhd S \overset{y}{\Rrightarrow} U \lhd \Gamma_2}\ \textsc{Cv-Sub}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{jDOTS} x.A \nearrow T \dashv \Gamma_1' \\ \Gamma_1' \rhd T \overset{y}{\Rrightarrow} U \lhd \Gamma_2\end{array}}{\Gamma_1 \rhd x.A \overset{y}{\Rrightarrow} U \lhd \Gamma_2}\ \textsc{Cv-Sel-Left} \qquad \frac{\begin{array}{c}\Gamma_2 \vdash_{jDOTS} x.A \searrow T \dashv \Gamma_2' \\ \Gamma_1 \rhd S \overset{y}{\Rrightarrow} T \lhd \Gamma_2'\end{array}}{\Gamma_1 \rhd A \overset{y}{\Rrightarrow} x.A \lhd \Gamma_2}\ \textsc{Cv-Sel-Right}$$

$$\frac{\Gamma_1 \rhd D_y \overset{y}{\Rrightarrow} T \lhd \Gamma_2}{\Gamma_1 \rhd \mu D \overset{y}{\Rrightarrow} T \lhd \Gamma_2}\ \textsc{Cv-Mu-Left} \qquad \frac{\Gamma_1 \rhd T \overset{y}{\Rrightarrow} D_y \lhd \Gamma_2}{\Gamma_1 \rhd T \overset{y}{\Rrightarrow} \mu D \lhd \Gamma_2}\ \textsc{Cv-Mu-Right}$$

$$\frac{\begin{array}{c}T \text{ is not an intersection} \\ \Gamma_1 \rhd S \overset{y}{\Rrightarrow} T \lhd \Gamma_2\end{array}}{\Gamma_1 \rhd S \wedge U \overset{y}{\Rrightarrow} T \lhd \Gamma_2}\ \textsc{Cv-Left1} \qquad \frac{\begin{array}{c}T \text{ is not an intersection} \\ \Gamma_1 \rhd U \overset{y}{\Rrightarrow} T \lhd \Gamma_2\end{array}}{\Gamma_1 \rhd S \wedge U \overset{y}{\Rrightarrow} T \lhd \Gamma_2}\ \textsc{Cv-Left2}$$

$$\frac{\Gamma_1 \rhd T \overset{y}{\Rrightarrow} S \lhd \Gamma_2 \quad \Gamma_1 \rhd T \overset{y}{\Rrightarrow} U \lhd \Gamma_2}{\Gamma_1 \rhd T \overset{y}{\Rrightarrow} S \wedge U \lhd \Gamma_2}\ \textsc{Cv-Right}$$

Variable Type Check

$$\frac{\Gamma \rhd \Gamma(x) \overset{x}{\Rrightarrow} T \lhd \Gamma}{\Gamma \vdash_{jDOTS} x \overleftarrow{:} T}\ \textsc{Chk-Var}$$

Figure 7.8: Definitions of convertibility and variable checking

Theorem 7.15. *(soundness) If $\Gamma \vdash_{jDOTS} x \overleftarrow{:} T$, then $\Gamma \vdash_{jDOT} x : T$.*

Clearly, the variable type check rule Chk-Var is just a specialization of **Conversion**, and therefore I will put more focus on the **Conversion** operation. **Conversion** says that a type can be converted to another provided there is a variable of this type. A quick glance indicates that **Conversion** greatly resembles stare-at subtyping, and some might wonder why it is necessary to have a separate definition. The rules are designed to have as large a decidable fragment as possible and the complication comes from the Rec-I, Rec-E and And rules. Illustration through examples makes the necessity of the rules clear.

Rec-I and Rec-E motivate the Cv-Mu-Left and Cv-Mu-Right rules. Consider the following context:

$$\Gamma = x : \mu\{w.A : \bot..\top\}$$

The variable checking problem is the following:

$$\Gamma \vdash_{jDOTS} x \overset{\leftarrow}{:} \{A : \bot..\top\}$$

Clearly, this checking problem cannot be successful via subtyping because there is no direct subtyping between μ types and declarations. However, with Rec-I, this checking problem can be admitted. The following derivation shows how **Conversion** admits this checking problem.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \gg \{A : \bot..\top\} <: \{A : \bot..\top\} \ll \Gamma}\text{ REFLEXIVITY}}{\Gamma \rhd \{A : \bot..\top\} \overset{x}{\Rrightarrow} \{A : \bot..\top\} \lhd \Gamma}\text{ Cv-Sub}}{\Gamma \rhd \mu\{w.A : \bot..\top\} \overset{x}{\Rrightarrow} \{A : \bot..\top\} \lhd \Gamma}\text{ Cv-Mu-Left}}{\Gamma \vdash_{jDOTS} x \overset{\leftarrow}{:} \{A : \bot..\top\}}\text{ Chk-Var}$$

And motivates Cv-Right. Consider the following context:

$$\Gamma = x : \mu\{w.A : \bot..\top\} \wedge \{B : \bot..\top\}$$

The variable checking problem is the following:

$$\Gamma \vdash_{jDOTS} x \overset{\leftarrow}{:} \{A : \bot..\top\} \wedge \mu\{w.B : \bot..\top\}$$

This can be seen from the following derivation.

$$\dfrac{\dfrac{\begin{matrix}\Gamma \rhd \mu\{w.A : \bot..\top\} \wedge \{B : \bot..\top\} \overset{x}{\Rrightarrow} \{A : \bot..\top\} \lhd \Gamma \\ \Gamma \rhd \mu\{w.A : \bot..\top\} \wedge \{B : \bot..\top\} \overset{x}{\Rrightarrow} \mu\{w.B : \bot..\top\} \lhd \Gamma\end{matrix}}{\Gamma \rhd \mu\{w.A : \bot..\top\} \wedge \{B : \bot..\top\} \overset{x}{\Rrightarrow} \{A : \bot..\top\} \wedge \mu\{w.B : \bot..\top\} \lhd \Gamma}\text{ Cv-Right}}{\Gamma \vdash_{jDOTS} x \overset{\leftarrow}{:} \{A : \bot..\top\} \wedge \mu\{w.B : \bot..\top\}}$$

On the top of the derivation, Cv-Right requires two **Conversion** witnesses. The first one has been done in the previous example, after a Cv-Left1. The second witness can be seen from a Cv-Left2 followed by a Cv-Mu-Right.

Moreover, imagine the context in this example:

$$\Gamma = y : \{C : \bot..\mu\{w.A : \bot..\top\}\}; x : y.C \wedge \{B : \bot..\top\}$$

To check x's type, **Upcast** needs to be applied to $y.C$ first, which justifies all the path type related rules.

The execution of variable type check also requires backtracking, due to the **∧-Traversal** and stare-at subtyping in it, as well as Cv-Left1 and Cv-Left2 rules. The termination argument is similar to stare-at subtyping and should be direct.

Theorem 7.16. ***Conversion*** *terminates as a non-deterministic algorithm.*

Proof. The only rules unique in **Conversion** are the Cv-Mu-Left and Cv-Mu-Right rules. The accounting measure clearly decreases when go from μD to D. □

7.8.2 Variable type synthesis

Variable type synthesis has a different focus from variable type check. This operation needs to make sure the returned type successfully avoids some set of free variables. The Rec-I, Rec-E and And rules also have impact in the rules.

Definition 7.9. *The definitions of* ***Synthesizer*** *and variable type synthesis are shown in Figure 7.9.*

For a ***Synthesizer*** *judgment $\Gamma \rhd_x S/V \overrightarrow{:} U$, Γ, x, S and a set of variables V are the inputs and U is the output.*

For a variable type synthesis judgment $\Gamma \vdash_{jDOTS} x/V \overrightarrow{:} T$, Γ, x and V are the inputs and T is the output.

Unlike the check direction which is given a type to check, the synthesis direction is required to figure out a type from a variable. Therefore one more soundness condition is needed.

Theorem 7.17. *(soundness) If $\Gamma \vdash_{jDOTS} x/V \overrightarrow{:} T$, then $\Gamma \vdash_{jDOT} x : T$.*

Lemma 7.18. *If $\Gamma \vdash_{jDOTS} x/V \overrightarrow{:} T$, then $fv(T) \subseteq dom(\Gamma) \backslash V$.*

Synthesizer

$$\frac{y \notin V}{\Gamma \rhd_x y.A/V \overrightarrow{:} y.A}\ \textsc{Sth-Stop} \qquad \frac{\Gamma \vdash_{jDOTS} y.A \nearrow S \dashv \Gamma' \quad \Gamma' \rhd_x S/V \overrightarrow{:} U}{\Gamma \rhd_x y.A/V \overrightarrow{:} U}\ \textsc{Sth-Sel}$$

$$\frac{\Gamma \rhd_x D_x/V \backslash x \overrightarrow{:} D'_x}{\Gamma \rhd_x \mu D/V \overrightarrow{:} \mu D'}\ \textsc{Sth-Mu} \qquad \frac{\Gamma \rhd_x S/V \overrightarrow{:} S' \quad \Gamma \rhd_x U/V \overrightarrow{:} U'}{\Gamma \rhd_x S \wedge U/V \overrightarrow{:} S' \wedge U'}\ \textsc{Sth-And}$$

$$\frac{\Gamma \vdash_{jDOTS} S \Uparrow_V U}{\Gamma \rhd_x S/V \overrightarrow{:} U}\ \textsc{Sth-Prom}$$

Variable Type Synthesis

$$\frac{\Gamma_1 \rhd_x S/V \overrightarrow{:} U}{\Gamma_1; x : S; \Gamma_2 \vdash_{jDOTS} x/V \overrightarrow{:} U}\ \textsc{Syn-Var}$$

Figure 7.9: Definitions of type synthesizer and variable type synthesis

Similar to type check, variable type synthesis is a specialization of **Synthesizer**. Notice that the context passed into **Synthesizer** is Γ_1, the part of the context preceding x. This is to ensure the termination argument is easier to make. Sth-Mu uses both the Rec-I and Rec-E rules. Notice that in the recursive case, x is removed from V because even if D' refers to x, it is guaranteed to be gone once Rec-I is applied. Sth-And uses the And rule. The Sth-Prom rule uses **Promotion** which is to be shown next.

Theorem 7.19. *Type variable synthesis terminates as a non-deterministic algorithm.*

Proof. The measure is the lexicographic order of the accounting measure and the structural measure. The termination argument is the easiest so far and should be straightforward. □

7.8.3 Promotion and Demotion

In Chapter 6, variants of **Promotion** and **Demotion**, **Promotion**$^\mu$ and **Demotion**$^\mu$, are defined to handle μ types, which I concluded unnecessary at the beginning of the chapter. Indeed, with the strengthened **Revealing**, **Promotion** and **Demotion** can have more reliance on **Revealing** and the definitions are simplified. The full definitions of

Promotion and **Demotion** are too verbose. To focus on the point, I will just discuss two rules that reflect the modification, and present the definitions in Appendix B.

Definition 7.10. *See full definitions in Appendix B.*

*For a judgment of **Promotion** $\Gamma \vdash_{jDOTS} S \Uparrow_V U$, Γ, S and V are the inputs and U is the output. A **Demotion** judgment $\Gamma \vdash_{jDOTS} S \Downarrow_V U$ works similarly.*

Consider the path type cases where a type declaration or a μ declaration is selected by **$\wedge$-Traversal** in **Promotion**.

$$\frac{\begin{array}{c} T \mapsto T_1 \,\&\, T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Lleftarrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \{A : S..U\} \,\&\, T_4 \quad \Gamma_1' \vdash_{jDOTS} U \Uparrow_V U' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow_V U'} \text{ P-SEL2}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \,\&\, T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Lleftarrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \mu\{w.A : S..U_w\} \,\&\, T_4 \quad \Gamma_1'; x : T_2 \wedge T_4 \vdash_{jDOTS} U_x \Uparrow_V U' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow_V U'} \text{ P-SEL3}$$

In **Promotion**, if a path type is encountered, and $x \in V$, these rules might apply. The first three predicates are similar to **Revealing**. P-SEL2 should be quite straightforward. Since T_3 is guaranteed to be closed in Γ_1', so is $\{A : S..U\}$ and the recursion just recurs down U which is clearly a strictly smaller subproblem.

In P-SEL3, the recursive call recurs on a extended context $\Gamma_1'; x : T_2 \wedge T_4$. Applying the same argument as in **Revealing**, this subproblem is also smaller. These two cases are enough to show that the improvement in **Revealing** has fundamental effects in other operations to replace all other extra operations in $\mu DART$.

A number of properties of **Promotion** and **Demotion** can be shown.

Lemma 7.20. *(soundness)*

1. *If $\Gamma \vdash_{jDOTS} S \Uparrow_V U$, then $\Gamma \vdash_{jDOT} S <: U$.*
2. *If $\Gamma \vdash_{jDOTS} S \Downarrow_V U$, then $\Gamma \vdash_{jDOT} U <: S$.*

Lemma 7.21. *If $\Gamma \vdash_{jDOTS} S \Uparrow_V (\Downarrow_V) U$, the following hold.*

1. *U is closed in Γ.*
2. *The variables in V do not occur free in U.*

7.9 Bi-directional General Term Typing

This section briefly discusses the general term typing. Once variable typing, **Promotion** and **Demotion** are defined, the general term typing is actually quite routine. To make the discussion concise, I put the full definition in Appendix C and here I only discuss two selected rules, and focus more on the execution of the typing rules.

Definition 7.11. *See full definitions in Appendix C.*

For a type synthesis judgment $\Gamma \vdash_{jDOTS} t/V \overrightarrow{:} T$, Γ, t *and* V *are inputs and* T *is output.*

For a type checking judgment $\Gamma \vdash_{jDOTS} t \overleftarrow{:} T$, Γ, t *and* T *are inputs and the algorithm returns true if the derivation satisfies the definition.*

Consider the following two rules in the synthesis and check directions.

$$\frac{\begin{array}{c} T \mapsto T_1 \mathbin{\&} T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \mu\{w.a : U_w\} \mathbin{\&} T_4 \quad \Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} U_x \Uparrow_V U' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.a/V \overrightarrow{:} U'} \text{ SYN-OBJ-E3}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \mathbin{\&} T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \forall(z : S)U_z \mathbin{\&} T_4 \\ \Gamma \vdash_{jDOTS} y \overleftarrow{:} S \quad \Gamma_2; x : T; \Gamma_1 \gg U_y <: U' \ll \Gamma_2; x : T; \Gamma_1 \end{array}}{\Gamma_2; x : T; \Gamma_1 \vdash_{jDOTS} x\ y \overleftarrow{:} U'} \text{ CHK-ALL-E2}$$

Due to intersection types, this pattern of **$\wedge$-Traversal** with **Revealing** is spread in the typing rules as well. In these cases, the second **$\wedge$-Traversal**s are used to find the wanted types so that the type can either be synthesized or checked.

The following properties can be proved.

Theorem 7.22. *(soundness) If* $\Gamma \vdash_{jDOTS} t \overleftarrow{:} T$, *then* $\Gamma \vdash_{jDOT} t : T$.

Theorem 7.23. *(soundness) If* $\Gamma \vdash_{jDOTS} t/V \overrightarrow{:} T$, *then* $\Gamma \vdash_{jDOT} t : T$.

Lemma 7.24. *If* $\Gamma \vdash_{jDOTS} t/V \overrightarrow{:} T$, *then* $fv(T) \subseteq dom(\Gamma) \backslash V$.

Theorem 7.25. *Type synthesis and type checking both terminate as non-deterministic algorithms.*

Proof. The measure is the structure of the term. □

Since $\wedge$**-Traversal** is involved in typing rules, backtracking is needed to fully extend the decidable fragment. However, some backtracking is obviously unnecessary. Specifically, for judgments like stare-at subtyping and typing in the check direction, only yes and no answers are drawn from these judgments, so once these judgments have given yes answers, there is no need to have later backtracking get into them. On the other hand, since type synthesis gives types as results, it might make a wrong guess in the middle so later backtracking needs to trace back into type synthesis, and $\wedge$**-Traversal** and **Revealing** in it, to guess a better type. This indicates that type synthesis with intersection types with this large a feature set might be very expensive to do.

7.10 Encoding into $jDOT$

The decidable fragment of subtyping can be seen to subsume those of $D_\wedge$ and $\mu DART$. The decidable fragment of $D_\wedge$ has been gracefully described by strong kernel $D_\wedge$ while one of $\mu DART$ can only be understood operationally at this moment, discussed in detail in Section 6.10. For this reason, the decidable fragment of algorithmic $jDOT$ is better understood empirically. In this section, I will show that $jDOT$ can encode a covariant list data structure and that the bi-directional type assignment is capable of type checking this encoding.

The covariant list encoding is taken from Amin et al. [2016] modulo adaptation to $jDOT$. To make the presentation concise, I adopt the following abbreviations.

$$\begin{aligned}
\mu\{D_1; ...; D_n\} &\equiv \mu D_1 \wedge ... \wedge \mu D_n \\
\{A\} &\equiv \{A : \bot..\top\} \\
\{A <: T\} &\equiv \{A : \bot..T\} \\
\{A = T\} &\equiv \{A : T..T\} \\
\forall(x : T_1, ..., x : T_n)T &\equiv \forall(x : T_1)...\forall(x : T_n)T \\
\lambda(x : T_1, ..., x : T_n)t &\equiv \lambda(x : T_1)...\lambda(x : T_n)t
\end{aligned}$$

In the encoding, I will just define *head* and *tail* members for lists. The definition of list resides in a package object, which has the self reference p. The type of the package object is as follows.

$$\begin{aligned}
&\mu\{p.List = \mu\{w.head : w.A; w.tail : p.List \wedge \{A <: w.A\}\}\\
&\quad ; p.nil : p.List \wedge \{A <: \bot\}\\
&\quad ; p.cons : \forall(x : \{B\}, y : x.B, z : p.List \wedge \{A <: x.B\})p.List \wedge \{A <: x.B\}\\
&\quad \}
\end{aligned}$$

To capture the pattern in the encoding, I will need these additional abbreviations.

$$\begin{aligned}
ListT &\equiv \mu\{w.head : w.A; w.tail : List[w.A]\}\\
List[T] &\equiv p.List \wedge \{A <: T\}
\end{aligned}$$

Then the package object type can be written as

$$\begin{aligned}
packageT \equiv \mu\{&p.List = ListT\\
&; p.nil : List[\bot]\\
&; p.cons : \forall(x : \{B\}, y : x.B, z : List[x.B])List[x.B]\\
&\}
\end{aligned}$$

Notice that I intentionally missed the A type member declaration in the definition of $p.List$. This is fine, because in $jDOT$ (and in DOT), object types do not have to be "well-formed". In fact, if $w.A$ is used non-trivially, the definition cannot be type checked if it is not defined. This seems to suggest a form of encoding of higher kinded types in $jDOT$.

The package itself can be defined by an object.

$$\begin{aligned}
&\nu(p : packageT)\{\\
&\quad List = \quad ListT\\
&\quad ;nil = \quad \text{let result } = \nu(w : \mu\{w.A = \bot\} \wedge ListT)\\
&\qquad\qquad\qquad \{A = \bot; head = w.head; tail = w.tail\}\\
&\qquad\qquad\quad \text{in result}\\
&\quad ;cons = \quad \lambda(x : \{B\}, y : x.B, z : List[x.B])\\
&\qquad\qquad\qquad \text{let result } = \nu(w : \mu\{w.A = x.B\} \wedge ListT)\\
&\qquad\qquad\qquad\quad \{A = x.B; head = y; tail = z\}\\
&\qquad\qquad\qquad \text{in result}\\
&\quad \}
\end{aligned}$$

This encoding can be seen typeable in the declarative definition, and the whole package object will have type $packageT$ as defined above.

Lemma 7.26. *The encoding is typeable.*

By the same reasoning as in Amin et al. [2016], when typing nil and $cons$, AND and REC-E are needed. What is more exciting is that the type synthesis direction can synthesize the type of this object from the empty context.

Lemma 7.27. *The type of encoding is synthesized to be $packageT$.*

Though the entry point is type synthesis, the object definition forces the direction quickly switches to the type checking direction. When checking nil, variable type checking is necessary to convert $\mu\{w.A = \bot\} \wedge ListT$ to $p.List \wedge \{A <: \bot\}$, where the former syntactically expands to $\mu\{w.A = \bot; w.head : w.A; w.tail : p.List \wedge \{A <: w.A\}\}$. In **Conversion**, CV-RIGHT first applies to check if the object type can be converted to $p.List$ and $\{A <: \bot\}$ separately. CV-SEL-RIGHT is needed to find out the actual definition of $p.List$, and CV-MU-LEFT is needed to eliminate the μ type. The checking of $cons$ works similarly, after a few invocations of CHK-ALL-I2.

Chapter 8

Discussion and Future Work

In the previous chapters, I described ways to establish undecidability proofs of a number of calculi and designed their algorithmic typing and subtyping rules. During the technical development, several problems were left behind. In this chapter, I collect some of those problems and discuss their importance and potential solutions. Though I might propose potential directions or solutions, they are not formally verified and remain speculative. Yet, I still hope the speculations can eventually contribute to the future progress of these problems.

8.1 Decidability Analysis on Language Features

In Chapter 3, I performed an in-depth discussion on how to establish the undecidability proof of $D_{<:}$ if subtyping comparison between parameter types of dependent function types is allowed. In Chapter 4, I made use of the insight obtained from the undecidability analysis and proposed stare-at subtyping, which overcomes two limitations identified in the chapter.

However, reviewing stare-at subtyping and its corresponding declarative form strong kernel $D_{<:}$, one can notice that the algorithm does not attempt to decide bad bounds at all. This can be seen in strong kernel $D_{<:}$ in which the BB rule is simply removed. This raises the following question.

Problem 1. *Do bad bounds introduce undecidability?*

This problem has two aspects.

1. The first one requires an investigation on the bad bound phenomenon. If the attempt is to show that bad bounds do introduce undecidability, then one must show the halting problem (or other undecidable problem) can be reduced to subtyping decision with the presence of bad bounds. To achieve this, it is required to isolate the effect of dependent function types, so the best calculus to start with is probably $F^{-}_{<:>}$ with syntactically identical parameter types. Once this is done, one can see if the proof works for kernel $D_{<:}$ with bad bounds. (Recall that kernel $D_{<:}$ is the declarative form of step subtyping which requires the parameter types of dependent functions to be identical in a subtyping comparison.)

2. The second aspect is about what we can learn from the analysis. Since the behavior of bad bounds is unknown, bad bounds in stare-at subtyping are completely unsupported. However, is it possible to support a fragment of bad bounds with an algorithm that still terminates? This aspect cannot be resolved without performing a rigorous study on this particular phenomenon.

Therefore, though $D_{<:}$ subtyping has been proven undecidable, it does not mean more undecidability analysis is meaningless. Despite being the simplest calculus in the *DOT* family, $D_{<:}$ is still filled with unknowns.

Similarly, the recursive types (or μ types) might also introduce undecidability.

Problem 2. *Do μ types introduce undecidability?*

The argument of why this analysis is important is analogous to the one about bad bounds. What is more, lack of knowledge on μ types gives more impact than bad bounds. In Chapter 4, kernel and strong kernel $D_{<:}$ are described, and bad bounds do not give troubles introducing them (because bad bounds are removed from the kernels). However, in Chapter 6 and Chapter 7, I am not able to give kernel forms precisely due to μ types. Moreover, the algorithmic typing and subtyping in Chapter 6 and Chapter 7 relies on the idea of types as resources, which requires the operations to maintain a certain invariant of type consumption. I expect that analyzing the decidability of μ types can give a more graceful algorithm which can be shown sound and terminating without adding significant complication to the proofs.

I think the main difficulty of decidability analysis on these two features is going to be considering how to make subtyping derivations deterministic. Specifically, decidability analysis of both features will require new reduction from known undecidable problems and therefore what is done in Pierce [1992] is unlikely to be directly applicable. Usually, a well-known undecidable problem tends to have tighter correspondence to some model of machines. The post-correspondence problem [Forster et al., 2018], PCP, is one of the examples. Yes-witnesses of these problems have only one witness, while a yes-witness in $D_{<:}$ subtyping usually has more than one derivations. In fact, Pierce [1992] started with first discovering a deterministic and undecidable fragment. With the large freedom of derivation in $D_{<:}$, I speculate that this is going to be difficult and at this moment it is hard to speculate what can be the best candidate problems for both features.

8.2 Soundness of $jDOT$

In Chapter 7, I introduced an alternative core calculus of Scala, $jDOT$, and explained why it is a better choice. Due to time limitation, I am not able to prove its soundness, while for a calculus to be interesting, the first requirement is to present its soundness proof. I made an informal argument of why I believe this calculus is sound, by considering the close relation between its definition and the usage of runtime information presented in references [Rompf and Amin, 2016, Amin et al., 2016, Rapoport et al., 2017, Amin and Rompf, 2017]. This discussion can be found in details in Section 7.2.

Therefore, being able to prove the soundness of $jDOT$ is an important reason to justify the definition of the calculus.

Problem 3. *Is $jDOT$ sound?*

Again, I speculate it is true, because the adjustment of the calculus is motivated to simply capture only the runtime portion needed in the soundness proof. Following Rapoport et al. [2017] should generate the desired proof.

8.3 Undecidability of $jDOT$

The next interesting problem is ask if $jDOT$ is undecidable. I conjecture the answer is yes. Notice that proving its undecidability might be related to the undecidability of bad bounds and/or μ types, but I speculate that might not be necessary, if a normal form of $jDOT$ can be found.

Problem 4. *Is $jDOT$ (sub)typing undecidable?*

Problem 5. *What is $jDOT$ normal form?*

If $jDOT$ normal form is discovered, then one should be able to show reduction from one of the known undecidable calculi presented in this thesis, hypothetically $F_{<:}^{-}$ or $D_{<<:}$.

Recall that in the previous discussion in Section 3.11.2, I pointed out that the main difficulty is how to unravel the mutual dependency between typing and subtyping rules, due to μ types. One inspiration to approach this problem is to realized a recurring relation in this thesis, convertibility.

Intuitively, a type S is convertible to another type U, if a variable x of type S can be witnessed at type U. In languages without μ types, e.g. $D_{<:}$ and $D_{\wedge}$, this relation coincides with subtyping, but with μ types, they are no longer the same. I speculate that $jDOT$ normal form can no longer be a ternary predicate but a quaternary one, with convertibility taken into account. Formally, I propose the following relation.

Definition 8.1. *A sub-conversion relation $\Gamma \vdash_{jDOT} S <:_{x?} U$ is a quaternary predicate, where $x?$ means an optional variable.*

1. *If the variable is given, as indicated by $\Gamma \vdash_{jDOT} S <:_{x} U$, then this is a convertibility relation. The desired property of the convertibility relation is that if $\Gamma \vdash_{jDOT} x : S$ and $\Gamma \vdash_{jDOT} S <:_{x} U$, then $\Gamma \vdash_{jDOT} x : U$.*
2. *If the variable is not given, indicated by $\Gamma \vdash_{jDOT} S <:_{\not{x}} U$, the concept should be equivalent to the usual subtyping relation $\Gamma \vdash_{jDOT} S <: U$.*

Moreover, a sub-conversion relation must not be mutually defined. I propose the following properties to test if a potential definition of sub-conversion can be used to establish an undecidability proof.

1. If $\Gamma \vdash_{jDOT} x : S$ and $\Gamma \vdash_{jDOT} S <:_{x} U$, then $\Gamma \vdash_{jDOT} x : U$.
2. If $\Gamma \vdash_{jDOT} x : T$, then $\Gamma \vdash_{jDOT} \Gamma(x) <:_{x} T$.
3. $\Gamma \vdash_{jDOT} S <: U$ iff $\Gamma \vdash_{jDOT} S <:_{\not{x}} U$.
4. If $\Gamma \vdash_{jDOT} S <:_{\not{x}} T$ and $\Gamma \vdash_{jDOT} T <:_{\not{x}} U$, then $\Gamma \vdash_{jDOT} S <:_{\not{x}} U$.
5. If $\Gamma \vdash_{jDOT} S <:_{x} T$ and $\Gamma \vdash_{jDOT} T <:_{x} U$, then $\Gamma \vdash_{jDOT} S <:_{x} U$.

The first property tests that sub-conversion indeed indicates convertibility when a variable is given. The second property tests convertibility is strong enough. The third property tests that if a variable is not given, indeed sub-conversion becomes the subtyping relation.

The remaining two properties are transitivity with the optional variable given or not. They should be required when proving the third property.

I propose a tentative definition of sub-conversion in Figure 8.1.

To select some rules to discuss, in the Sc-Fld rule, when the member types are compared, the optional variable in the premise is guaranteed not given. This is because even if a variable x is given, it cannot influence the subtyping judgment just by using the typing and subtyping rules.

On the other hand, in the Sc-Sel1 rule, the premise necessarily requires a variable because $\Gamma(x)$ surely is a type of the variable x. Moreover, this rule is very close to the Sc-Sel3 rule. Their differences are (a) Sc-Sel1 requires no variable given as input while Sc-Sel3 requires a variable, and (b) Sc-Sel1 has one less premise than Sc-Sel3. This is because the given variable y in Sc-Sel3 might contribute to establishing the witness of $\Gamma \vdash_{jDOT} S' <:_y S$, while in Sc-Sel1, due to higher dimensional absorption, this extra predicate is clearly redundant.

The same differences can be seen in the pairs of Sc-Sel2 and Sc-Sel4, and Sc-BB1 and Sc-BB2. Sc-Sel2 and Sc-Sel4 are the dual cases of Sc-Sel1 and Sc-Sel3 and Sc-BB1 and Sc-BB2 are the bad bound generalization.

Sc-Rec-I and Sc-Rec-E are designed to replace Rec-I and Rec-E and therefore they must require a variable.

Notice that rules like Sc-Sel2 are storing a type which is not found in the inputs, so one might argue this definition is not a normal form. But if the focus is on the $\Gamma \vdash_{jDOT} S \wedge U <:_{\not{x}} T$ fragment, then all the rules would satisfy the definition of normal form. After all, the problem is about proving undecidability of subtyping, not about sub-conversion.

Clearly, this definition contains a large number of rules, and the proofs most likely require case analysis on whether the variable is given or not, so there is a substantial amount of technical work to do in order to understand this definition and potentially use it to establish undecidability of subtyping and typing of $jDOT$. I further speculate that, in the reduction from $F_{<:}^{-}$ or $D_{<<:}$, the images of the interpretation functions are restrictive enough, so that all uses of variables in the convertibility relation are not substantial and effectively turn the whole sub-conversion to subtyping relation. Formally,

$$\text{If } \langle\langle\Gamma\rangle\rangle \vdash_{jDOT} [\![S]\!] <:_x [\![U]\!], \text{ then } \langle\langle\Gamma\rangle\rangle \vdash_{jDOT} [\![S]\!] <:_{\not{x}} [\![U]\!].$$

If this statement can be shown, then the undecidability of $jDOT$ is likely achievable, but at the current stage, it is very difficult to judge whether this is the right direction to pursue.

$$\dfrac{}{\Gamma \vdash_{jDOT} T <:_{x?} \top}\ \textsc{Sc-Top} \qquad \dfrac{}{\Gamma \vdash_{jDOT} \bot <:_{x?} T}\ \textsc{Sc-Bot} \qquad \dfrac{}{\Gamma \vdash_{jDOT} T <:_{x?} T}\ \textsc{Sc-Refl}$$

$$\dfrac{\Gamma \vdash_{jDOT} S_2 <:_{\not x} S_1 \quad \Gamma \vdash_{jDOT} U_1 <:_{\not x} U_2}{\Gamma \vdash_{jDOT} \{A : S_1..U_1\} <:_{x?} \{A : S_2..U_2\}}\ \textsc{Sc-Bnd} \qquad \dfrac{\Gamma \vdash_{jDOT} T_1 <:_{\not x} T_2}{\Gamma \vdash_{jDOT} \{a : T_1\} <:_{x?} \{a : T_2\}}\ \textsc{Sc-Fld}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash_{jDOT} S_2 <:_{\not y} S_1 \\ \Gamma ; x : S_2 \vdash_{jDOT} U_1 <:_{\not y} U_2\end{array}}{\Gamma \vdash_{jDOT} \forall(x : S_1)U_1 <:_{y?} \forall(x : S_2)U_2}\ \textsc{Sc-All} \qquad \dfrac{\begin{array}{c}\Gamma \vdash_{jDOT} \Gamma(x) <:_{x} \{A : S..\top\} \\ \Gamma \vdash_{jDOT} \Gamma(x) <:_{x} \{A : \bot..U\}\end{array}}{\Gamma \vdash_{jDOT} S <:_{\not y} U}\ \textsc{Sc-BB1}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash_{jDOT} \Gamma(x) <:_{x} \{A : S..\top\} \quad \Gamma \vdash_{jDOT} \Gamma(x) <:_{x} \{A : \bot..U\} \\ \Gamma \vdash_{jDOT} S' <:_{y} S \quad \Gamma \vdash_{jDOT} U <:_{y} U'\end{array}}{\Gamma \vdash_{jDOT} S' <:_{y} U'}\ \textsc{Sc-BB2}$$

$$\dfrac{\Gamma \vdash_{jDOT} \Gamma(x) <:_{x} \{A : S..\top\}}{\Gamma \vdash_{jDOT} S <:_{\not y} x.A}\ \textsc{Sc-Sel1} \qquad \dfrac{\Gamma \vdash_{jDOT} \Gamma(x) <:_{x} \{A : \bot..U\}}{\Gamma \vdash_{jDOT} x.A <:_{\not y} U}\ \textsc{Sc-Sel2}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash_{jDOT} \Gamma(x) <:_{x} \{A : S..\top\} \\ \Gamma \vdash_{jDOT} S' <:_{y} S\end{array}}{\Gamma \vdash_{jDOT} S' <:_{y} x.A}\ \textsc{Sc-Sel3} \qquad \dfrac{\begin{array}{c}\Gamma \vdash_{jDOT} \Gamma(x) <:_{x} \{A : \bot..U\} \\ \Gamma \vdash_{jDOT} U <:_{y} U'\end{array}}{\Gamma \vdash_{jDOT} x.A <:_{y} U}\ \textsc{Sc-Sel4}$$

$$\dfrac{\Gamma \vdash_{jDOT} S <:_{x?} T}{\Gamma \vdash_{jDOT} S \wedge U <:_{x?} T}\ \textsc{Sc-And-E1} \qquad \dfrac{\Gamma \vdash_{jDOT} U <:_{x?} T}{\Gamma \vdash_{jDOT} S \wedge U <:_{x?} T}\ \textsc{Sc-And-E2}$$

$$\dfrac{\Gamma \vdash_{jDOT} T <:_{x?} S \quad \Gamma \vdash_{jDOT} T <:_{x?} U}{\Gamma \vdash_{jDOT} T <:_{x?} S \wedge U}\ \textsc{Sc-And-I} \qquad \dfrac{\Gamma \vdash_{jDOT} D_x <:_{x} T}{\Gamma \vdash_{jDOT} \mu D <:_{x} T}\ \textsc{Sc-Rec-I}$$

$$\dfrac{\Gamma \vdash_{jDOT} \mu D <:_{x} T}{\Gamma \vdash_{jDOT} D_x <:_{x} T}\ \textsc{Sc-Rec-E}$$

Figure 8.1: Sub-conversion relation of *jDOT*

Chapter 9

Conclusion

In Chapter 1, I posed two questions on *DOT*.

Question 9. *(restate Question 1) Is (sub)typing of DOT decidable?*

Question 10. *(restate Question 2) How should DOT programs be type checked?*

To tackle the first question, in Chapter 3, I performed a decidability analysis on $D_{<:}$, the simplest calculus in the *DOT* family, and proposed the special focus on a special kind of declarative form of subtyping, normal form. Through defining $D_{<:}$ normal form, I succeeded in establishing the undecidability proofs of both typing and subtyping of $D_{<:}$, and extended the same idea to $D_{\wedge}$ in Chapter 5 where I showed the undecidability of $D_{\wedge}$ subtyping.

However, in Chapter 3, I also discussed why it is difficult to show the undecidability of (sub)typing of *DOT*, and I attempted to give a definition of *jDOT* normal form in Section 8.3, but to actually establish a rigorous proof, future work is needed.

To tackle the second question, in Chapter 4, I approached it by considering an existing work, step subtyping, and borrowing ideas from the decidability analysis done previously. I proposed a new algorithmic subtyping procedure, stare-at subtyping, and showed it to be strictly stronger than step subtyping. Algorithmic analysis combined with decidability analysis revealed two declarative decidable fragments of $D_{<:}$, kernel and strong kernel $D_{<:}$. I established the proofs that step subtyping is sound and complete w.r.t. kernel $D_{<:}$ and stare-at subtyping is sound and complete w.r.t. strong kernel $D_{<:}$. This result gracefully resolves the subtyping decision problem in $D_{<:}$.

In Chapter 5, I showed the extension of stare-at sutbyping to intersection types. I outlined in detail the execution of stare-at subtyping, by emphasizing the need for extensive backtracking. I showed that $D_{\wedge}$ also has a strong kernel form and stare-at subtyping is still sound and complete w.r.t. it.

In Chapter 6, I investigated recursive types. To handle recursive types, I proposed to consider types as resources, and stare-at subtyping was adapted in a way so that each path dependent type can only be mentioned by Alice and Bob once respectively.

In Chapter 7, I briefly reviewed the methods preceding this chapter and Wadlerfest DOT, and proposed $jDOT$ as a better alternative than DOT. Stare-at subtyping was refined, so that it is capable of handling all three features of path dependent types, intersection types and recursive types. Bi-directional type assignment of $jDOT$ is also discussed in detail. In this chapter, I have shown a bi-directional algorithm which is proven sound and terminating, and strong enough to type check a covariant list example implemented in $jDOT$.

This thesis has done an extensive investigation on the three important features in DOT calculi and introduced various techniques to handle their type checking problem. The technical work fills in the blank of decidability and algorithmic study of the DOT calculi.

References

Dotty documentation, 2019.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, September 1993. ISSN 0164-0925. doi: 10.1145/155183.155231. URL http://doi.acm.org/10.1145/155183.155231.

Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 666–679, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009866. URL http://doi.acm.org/10.1145/3009837.3009866.

Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, number EPFL-CONF-183030, 2012.

Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 233–249, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660216. URL http://doi.acm.org/10.1145/2660193.2660216.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. *The Essence of Dependent Object Types*, pages 249–272. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30936-1. URL https://doi.org/10.1007/978-3-319-30936-1_14.

David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1):273 – 309, 2001. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(00)00175-4. URL http://www.sciencedirect.com/science/article/pii/S0304397500001754.

Steve Awodey. *Category Theory*. Oxford University Press, Inc., New York, NY, USA, 2nd edition, 2010. ISBN 0199237182, 9780199237180.

H.P. Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984. ISBN 9780444867483. URL https://books.google.ca/books?id=eMtTAAAAYAAJ.

L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system f with subtyping. *Information and Computation*, 109(1):4 – 56, 1994. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1994.1013. URL http://www.sciencedirect.com/science/article/pii/S0890540184710133.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. ISSN 0360-0300. doi: 10.1145/6041.6042. URL http://doi.acm.org/10.1145/6041.6042.

Pierre Casteran and Matthieu Sozeau. A gentle introduction to type classes and relations in coq. 05 2012.

Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, Oct 2012. ISSN 1573-0670. doi: 10.1007/s10817-011-9225-2. URL https://doi.org/10.1007/s10817-011-9225-2.

Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN 0262026651, 9780262026659.

M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19(1):139–156, Dec 1978. ISSN 1432-0665. doi: 10.1007/BF02011875. URL https://doi.org/10.1007/BF02011875.

M. Coppo, M. Dezani-Ciancaglini, and P. Salle'. Functional characterization of some semantic equalities inside λ-calculus. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, pages 133–146, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg. ISBN 978-3-540-35168-9.

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In Rastislav Královič and Paweł Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006*, pages 1–23, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37793-1.

Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. In A. Arnold, editor, *CAAP '90*, pages 132–146, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. ISBN 978-3-540-47042-7.

Rowan Davies. *Practical Refinement-type Checking*. PhD thesis, Pittsburgh, PA, USA, 2005. AAI3168521.

Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and minsky machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 104–117, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6222-1. doi: 10.1145/3293880.3294096. URL http://doi.acm.org/10.1145/3293880.3294096.

Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in coq. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 189–206, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66107-0.

Yannick Forster, Edith Heiter, and Gert Smolka. Verification of pcp-related computational reductions in coq. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Oxford, UK, July 9-12, 2018*, LNCS 10895, pages 253–269. Springer, Jul 2018. Preliminary version appeared as arXiv:1711.07023.

Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting f-bounded polymorphism into shape. *SIGPLAN Not.*, 49(6):89–99, June 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594308. URL http://doi.acm.org/10.1145/2666356.2594308.

Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 73–85, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009871. URL http://doi.acm.org/10.1145/3009837.3009871.

J. Roger Hindley, Jonathan P. Seldin, and Garrel Pottinger. A type assignment for the strongly normalizabile -terms. 2013.

R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. ISSN 00029947. URL http://www.jstor.org/stable/1995158.

Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. 2006.

Daniel Leivant. Discrete polymorphism. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 288–297, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: 10.1145/91556.91675. URL http://doi.acm.org/10.1145/91556.91675.

Zhaohui Luo. Computation and reasoning: A type theory for computer science. 04 2019.

Cyprien Mangin and Matthieu Sozeau. Equations reloaded. working paper or preprint, December 2017. URL https://hal.inria.fr/hal-01671777.

J. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill Education, 2010. ISBN 9780073191461. URL https://books.google.ca/books?id=arluQAAACAAJ.

Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. ISBN 88-7088-105-9. Notes by Giovanni Sambin.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(78)90014-4. URL http://www.sciencedirect.com/science/article/pii/0022000078900144.

Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

Adriaan Moors, Frank Piessens, and Martin Odersky. Safe type-level abstraction in scala adriaan moors. 2007.

Abel Nieto. Towards algorithmic typing for dot (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 2–7, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5529-2. doi: 10.1145/3136000.3136003. URL http://doi.acm.org/10.1145/3136000.3136003.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 41–53, 2001.

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, 2003.

Christine Paulin-Mohring. Introduction to the calculus of inductive constructions, 2015.

Frank Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1):84 – 141, 2000. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1999.2832. URL `http://www.sciencedirect.com/science/article/pii/S0890540199928328`.

Benjamin C Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University, 1991.

Benjamin C. Pierce. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 305–315, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: 10.1145/143165.143228. URL `http://doi.acm.org/10.1145/143165.143228`.

Benjamin C. Pierce. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098.

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288.

Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *Proc. ACM Program. Lang.*, 1(OOPSLA):46:1–46:27, October 2017. ISSN 2475-1421. doi: 10.1145/3133870. URL `http://doi.acm.org/10.1145/3133870`.

Tiark Rompf and Nada Amin. Type soundness for dependent object types (dot). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 624–641, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984008. URL `http://doi.acm.org/10.1145/2983990.2984008`.

Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-71067-7.

Agda Team. Agda 2.5.4.2, 2019.

The Coq Development Team. The coq proof assistant, version 8.8.0, April 2018. URL https://doi.org/10.5281/zenodo.1219885.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

Stefan Wehr and Peter Thiemann. On the decidability of subtyping with bounded existential types. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 111–127, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10671-2. doi: 10.1007/978-3-642-10672-9_10. URL http://dx.doi.org/10.1007/978-3-642-10672-9_10.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *INFORMATION AND COMPUTATION*, 115:38–94, 1992.

APPENDICES

Appendix A

Various Operations in $\mu DART$

This appendix defines various operations that are not defined in the thesis. These operations are verbose to define while not too difficult to understand once the specification is listed. So I consider them a distraction from the actual content. Also note that the work in $\mu DART$ is completely revised and is entirely superseded by the work in $jDOT$. The definitions are here just to ensure the completeness of the document.

A.1 Exposure

Definition A.1. *The definition of **Exposure** can be found in Figure A.1.*

*An **Exposure** judgment $\Gamma \vdash_{\mu DARTS} S \Uparrow U$ has Γ and S as inputs and U as output.*

Lemma A.1. *If $\Gamma \vdash_{\mu DARTS} S \Uparrow U$, Following holds.*

1. $\Gamma \vdash_{\mu DART} S <: U$.
2. *U is not a path.*

A.2 Imposure

Imposure is the dual operation of **Exposure**.

$$\frac{T \text{ is not a path}}{\Gamma \vdash_{\mu DARTS} T \Uparrow T}\ \textsc{Ep-Stop} \qquad \frac{}{\Gamma \vdash_{\mu DARTS} T \Uparrow \top}\ \textsc{Ep-Top*}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \bot}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow \bot}\ \textsc{Ep-Bot}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \{DS\} \quad \{A : S..U\} \in \{DS\} \quad \Gamma_1 \vdash_{\mu DARTS} U \Uparrow U'}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow U'}\ \textsc{Ep-Rcd}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \mu(z : DS_z) \\ DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu} U \dashv DS'_x \quad \Gamma_1 \vdash_{\mu DARTS} U \Uparrow U'\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow U'}\ \textsc{Ep-Mu}$$

Figure A.1: The definition of **Exposure** in $\mu DART$

Definition A.2. *The definition of **Imposure** can be found in Figure A.2.*

*An **Imposure** judgment $\Gamma \vdash_{\mu DARTS} S \Downarrow U$ has Γ and S as inputs and U as output.*

Notice that **Imposure** uses **Exposure** to resolve x's type, T. This is correct, because if two types have subtype relation, then the types in their corresponding contra-variant position have supertype relation.

In the Ip-Mu rule, it has the premise $DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu} S \dashv DS'_x$, which is called **Imposure**$^{\mu}$ and is the dual operation of **Exposure**$^{\mu}$ to be defined in the next section.

Lemma A.2. *If $\Gamma \vdash_{\mu DARTS} S \Downarrow U$, Following holds.*

1. $\Gamma \vdash_{\mu DART} U <: S$.
2. *U is not a path.*

A.3 Imposure$^{\mu}$

Definition A.3. *The definition of **Imposure**$^{\mu}$ is defined in Figure A.3.*

*For a **Imposure**$^{\mu}$ judgment $DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu} T \dashv DS'_x$, declarations DS_x, variable x and type member label A are inputs and type T and declaations DS'_x are outputs.*

$$\frac{T \text{ is not a path}}{\Gamma \vdash_{\mu DARTS} T \Downarrow T}\ \textsc{Ip-Stop} \qquad \frac{}{\Gamma \vdash_{\mu DARTS} T \Downarrow \bot}\ \textsc{Ip-Bot*}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \bot}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Downarrow \top}\ \textsc{Ip-Top}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \{DS\} \quad \{A : S..U\} \in \{DS\} \quad \Gamma_1 \vdash_{\mu DARTS} S \Downarrow S'}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Downarrow S'}\ \textsc{Ip-Rcd}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \mu(z : DS_z) \quad DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu} S \dashv DS'_x \quad \Gamma_1 \vdash_{\mu DARTS} S \Downarrow S'}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Downarrow S'}\ \textsc{Ip-Mu}$$

Figure A.2: The definition of **Imposure** in $\mu DART$

$$\frac{A \notin dom(DS_x)}{DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu} \bot \dashv DS_x}\ \textsc{Im-Bot} \qquad \frac{\{A : y.B..U\} \in DS_x \quad x \neq y}{DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu} y.B \dashv DS_x \backslash A}\ \textsc{Im-Sel}$$

$$\frac{\{A : S..T\} \in DS_x \quad S \text{ is not a path}}{DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu} S \dashv DS_x \backslash A}\ \textsc{Im-Stop}$$

$$\frac{\{A : x.B..U\} \in DS_x \quad DS_x \backslash A \vdash_{\mu DARTS} x.B \Downarrow^{\mu} T \dashv DS'_x}{DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu} T \dashv DS'_x}\ \textsc{Im-Recur}$$

Figure A.3: Definition of **Imposure**$^{\mu}$

Lemma A.3. *(soundness) If $\Gamma \vdash_{\mu DART} x : \{DS_x\}$ and $DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu} T \dashv DS'_x$, then $\Gamma \vdash_{\mu DART} T <: x.A$.*

Lemma A.4. ***Imposure***$^{\mu}$ *returns either a non-path type or a path type that does not refer to the self reference of the μ type.*

Lemma A.5. *Returned declarations $\{DS'_x\}$ are a subset of the input declarations $\{DS_x\}$. To use set theoretic notation $\{DS'_x\} \subseteq \{DS_x\}$.*

A.4 Promotion and Demotion

Promotion and **Demotion** are responsible for finding supertype / subtype of a given type so that a given set of free variables do not occur free in the returned type. These operations are used in the type synthesis rules to ensure the returned type remains closed in the context.

Definition A.4. *The* ***Promotion*** *and* ***Demotion*** *operations are defined in Figure A.4.*

For a judgment of ***Promotion*** $\Gamma \vdash_{\mu DARTS} S \Uparrow_V U$*,* Γ*,* S *and* V *are the inputs and* U *is the output. A* ***Demotion*** *judgment* $\Gamma \vdash_{\mu DARTS} S \Downarrow_V U$ *works similarly.*

In the P-Sel3 rule, a judgment $DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu}_{x} U$ is used. This operation is called **Promotion**$^{\mu}$. Its relation to **Promotion** is similar to **Exposure**$^{\mu}$ to **Exposure**. Similarly, in the D-Sel3 rule, a **Demotion**$^{\mu}$ judgment $DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu}_{x} U$ is used. These two operations are to be defined in the next section. In the PD-Rcd rule, another judgment $\Gamma \vdash_{\mu DARTSD} \{DS\} \Downarrow_V (\Uparrow_V)\{DS'\}$ is used. This is the variant of **Promotion** and **Demotion** to loop over the record types, which is defined in Figure A.5.

Lemma A.6. *If* $\Gamma \vdash_{\mu DARTS} S \Uparrow_V (\Downarrow_V) U$*, the following hold.*

1. *(soundness)* $\Gamma \vdash_{\mu DART} S <: U$*.*
2. U *is closed in* Γ*.*
3. U *doesn't have any variables in* V *occur free.*

This justifies this set of operations.

A.5 Promotion$^{\mu}$ and Demotion$^{\mu}$

Similar to **Exposure**$^{\mu}$ and **Imposure**$^{\mu}$, **Promotion** and **Demotion** also need their variants to traverse the declarations as a cyclic context, which motivates **Promotion**$^{\mu}$ and **Demotion**$^{\mu}$.

Definition A.5. *The* ***Promotion***$^{\mu}$ *and* ***Demotion***$^{\mu}$ *operations are defined in Figure A.6.*

For a judgment of ***Promotion***$^{\mu}$ $DS_x \vdash_{\mu DARTS} S \Uparrow^{\mu}_{x} U$*,* DS_x*,* S *and* x *are the inputs and* U *is the output. A* ***Demotion*** *judgment* $DS_x \vdash_{\mu DARTS} S \Downarrow^{\mu}_{x} U$ *works similarly.*

$$\frac{}{\Gamma \vdash_{\mu DARTS} \top \Uparrow_V (\Downarrow_V) \top} \text{ PD-Top} \qquad \frac{}{\Gamma \vdash_{\mu DARTS} \bot \Uparrow_V (\Downarrow_V) \bot} \text{ PD-Bot}$$

$$\frac{}{\Gamma \vdash_{\mu DARTS} T \Uparrow_V \top} \text{ P-Top*} \qquad \frac{}{\Gamma \vdash_{\mu DARTS} T \Downarrow_V \bot} \text{ D-Bot*}$$

$$\frac{x \notin V}{\Gamma \vdash_{\mu DARTS} y.A \Uparrow_V (\Downarrow_V) y.A} \text{ PD-Var} \qquad \frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \bot}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow_V \bot} \text{ P-Sel1}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \bot}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Downarrow_V \top} \text{ D-Sel1}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \{DS\} \quad \{A : S..U\} \in \{DS\} \quad \Gamma_1 \vdash_{\mu DARTS} U \Uparrow_V U'}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow_V U'} \text{ P-Sel2}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \{DS\} \quad \{A : S..U\} \in \{DS\} \quad \Gamma_1 \vdash_{\mu DARTS} S \Downarrow_V S'}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Downarrow_V S'} \text{ D-Sel2}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \mu(z : DS_z) \quad DS_x \vdash_{\mu DARTS} x.A \Uparrow^{\mu}_{x} U \quad \Gamma_1 \vdash_{\mu DARTS} U \Uparrow_V U'}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Uparrow_V U'} \text{ P-Sel3}$$

$$\frac{\Gamma_1 \vdash_{\mu DARTS} T \Uparrow \mu(z : DS_z) \quad DS_x \vdash_{\mu DARTS} x.A \Downarrow^{\mu}_{x} U \quad \Gamma_1 \vdash_{\mu DARTS} S \Downarrow_V S'}{\Gamma_1; x : T; \Gamma_2 \vdash_{\mu DARTS} x.A \Downarrow_V S'} \text{ D-Sel3}$$

$$\frac{\Gamma \vdash_{\mu DARTS} S \Downarrow_V (\Uparrow_V) S' \quad \Gamma \vdash_{\mu DARTS} U \Uparrow_V (\Downarrow_V) U'}{\Gamma \vdash_{\mu DARTS} \forall(z : S)U \Uparrow_V (\Downarrow_V) \forall(z : S')U'} \text{ PD-All}$$

$$\frac{\Gamma \vdash_{\mu DARTSD} \{DS\} \Uparrow_V (\Downarrow_V) \{DS'\}}{\Gamma \vdash_{\mu DARTS} \{A : S..U\} \Uparrow_V (\Downarrow_V) \{A : S'..U'\}} \text{ PD-Rcd}$$

$$\frac{fv(DS) \cap V = \emptyset}{\Gamma \vdash_{\mu DARTS} \mu(z : DS_z) \Uparrow_V (\Downarrow_V) \mu(z : DS_z)} \text{ PD-Mu}$$

Figure A.4: Definitions of **Promotion** and **Demotion**

$$\frac{}{\Gamma \vdash_{\mu DARTSD} \{\} \Uparrow_V (\Downarrow_V)\{\}} \text{ PDR-NIL}$$

$$\frac{\Gamma \vdash_{\mu DARTS} T \Uparrow_V (\Downarrow_V)T' \quad \Gamma \vdash_{\mu DARTSD} \{DS\} \Uparrow_V (\Downarrow_V)\{DS'\}}{\Gamma \vdash_{\mu DARTSD} \{a : T; DS\} \Uparrow_V (\Downarrow_V)\{a : T'; DS'\}} \text{ PDR-FLD}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mu DARTS} S \Downarrow_V (\Uparrow_V)S' \\ \Gamma \vdash_{\mu DARTS} U \Uparrow_V (\Downarrow_V)U' \quad \Gamma \vdash_{\mu DARTSD} \{DS\} \Uparrow_V (\Downarrow_V)\{DS'\}\end{array}}{\Gamma \vdash_{\mu DARTSD} \{A : S..U; DS\} \Uparrow_V (\Downarrow_V)\{A : S'..U'; DS'\}} \text{ PDR-BND}$$

Figure A.5: Promotion and **Demotion** for record types

In the PDM-RCD rule, judgment $DS_x \vdash_{\mu DARTSD} \{DS_1\} \Uparrow^\mu_x (\Downarrow^\mu_x)\{DS_2\}$ is used. This is the variant of **Promotion**$^\mu$ and **Demotion**$^\mu$ that loops over the record types, which is defined in Figure A.7.

The following are properties of **Promotion**$^\mu$ and **Demotion**$^\mu$.

Lemma A.7. *If $DS_x \vdash_{\mu DARTS} S \Uparrow^\mu_x (\Downarrow^\mu_x)U$, then $x \notin fv(U)$.*

Lemma A.8. *1. If $\Gamma \vdash_{\mu DART} x : \{DS_x\}$ and $DS_x \vdash_{\mu DARTS} T \Uparrow^\mu_x U$, then $\Gamma \vdash_{\mu DART} T <: U$.*

2. If $\Gamma \vdash_{\mu DART} x : \{DS_x\}$ and $DS_x \vdash_{\mu DARTS} T \Downarrow^\mu_x U$, then $\Gamma \vdash_{\mu DART} U <: T$.

A.6 Bi-directional type assignment

With all those operations defined, bi-directional type assignment for general terms can be defined. Clearly, variable typing is necessary and has been discussed in Section 6.12.

Definition A.6. *The definition of type synthesis is defined in Figure A.8 and type checking is defined in Figure A.9.*

For a type synthesis judgment $\Gamma \vdash_{\mu DARTS} t/V \overrightarrow{:} T$, Γ, t and V are inputs and T is output.

For a type checking judgment $\Gamma \vdash_{\mu DARTS} t \overleftarrow{:} T$, Γ, t and T are inputs and the algorithm returns true if the derivation satisfies the definition.

$$\frac{}{DS_x \vdash_{\mu DARTS} \top \Uparrow^\mu_x (\Downarrow^\mu_x) \top}\ \text{PDM-Top} \qquad \frac{}{DS_x \vdash_{\mu DARTS} \bot \Uparrow^\mu_x (\Downarrow^\mu_x) \bot}\ \text{PDM-Bot}$$

$$\frac{}{DS_x \vdash_{\mu DARTS} T \Uparrow^\mu_x \top}\ \text{PM-Top*} \qquad \frac{}{DS_x \vdash_{\mu DARTS} T \Downarrow^\mu_x \bot}\ \text{DM-Bot*}$$

$$\frac{x \neq y}{DS_x \vdash_{\mu DARTS} y.A \Uparrow^\mu_x (\Downarrow^\mu_x) y.A}\ \text{PDM-Var}$$

$$\frac{A \in dom(DS_x) \quad DS_x \vdash_{\mu DARTS} x.A \Uparrow^\mu T \dashv DS'_x \quad DS_x \vdash_{\mu DARTS} T \Uparrow^\mu_x T'}{DS_x \vdash_{\mu DARTS} x.A \Uparrow^\mu_x T'}\ \text{PM-Sel}$$

$$\frac{A \in dom(DS_x) \quad DS_x \vdash_{\mu DARTS} x.A \Downarrow^\mu T \dashv DS'_x \quad DS_x \vdash_{\mu DARTS} T \Downarrow^\mu_x T'}{DS_x \vdash_{\mu DARTS} x.A \Downarrow^\mu_x T'}\ \text{DM-Sel}$$

$$\frac{DS_x \vdash_{\mu DARTS} S \Downarrow^\mu_x (\Uparrow^\mu_x) S' \quad DS_x \vdash_{\mu DARTS} U \Uparrow^\mu_x (\Downarrow^\mu_x) U'}{DS_x \vdash_{\mu DARTS} \forall(z : S)U \Uparrow^\mu_x (\Downarrow^\mu_x) \forall(z : S')U'}\ \text{PDM-All}$$

$$\frac{DS_x \vdash_{\mu DARTSD} \{DS_1\} \Uparrow^\mu_x (\Downarrow^\mu_x) \{DS_2\}}{DS_x \vdash_{\mu DARTS} \{DS_1\} \Uparrow^\mu_x (\Downarrow^\mu_x) \{DS_2\}}\ \text{PDM-Rcd}$$

$$\frac{x \notin fv(DS'_y)}{DS_x \vdash_{\mu DARTS} \mu(y : DS'_y) \Uparrow^\mu_x (\Downarrow^\mu_x) \mu(y : DS'_y)}\ \text{PDM-Mu}$$

Figure A.6: Definitions of **Promotion**$^\mu$ and **Demotion**$^\mu$

Lemma A.9. *(soundness) If* $\Gamma \vdash_{\mu DARTS} t \overleftarrow{:} T$*, then* $\Gamma \vdash_{\mu DART} t : T$.

Lemma A.10. *(soundness) If* $\Gamma \vdash_{\mu DARTS} t/V \overrightarrow{:} T$*, then* $\Gamma \vdash_{\mu DART} t : T$.

Lemma A.11. *If* $\Gamma \vdash_{\mu DARTS} t/V \overrightarrow{:} T$*, then* $fv(T) \cap V = \emptyset$.

$$\frac{}{DS_x \vdash_{\mu DARTSD} \{\} \Uparrow^{\mu}_{x} (\Downarrow^{\mu}_{x}) \{\}} \text{ PDMR-}\textsc{Nil}$$

$$\frac{DS_x \vdash_{\mu DARTS} T \Uparrow^{\mu}_{x} (\Downarrow^{\mu}_{x}) T' \quad DS_x \vdash_{\mu DARTSD} \{DS\} \Uparrow^{\mu}_{x} (\Downarrow^{\mu}_{x}) \{DS'\}}{DS_x \vdash_{\mu DARTSD} \{a : T; DS\} \Uparrow^{\mu}_{x} (\Downarrow^{\mu}_{x}) \{a : T'; DS'\}} \text{ PDNR-}\textsc{Fld}$$

$$\frac{\begin{array}{c} DS_x \vdash_{\mu DARTS} S \Downarrow^{\mu}_{x} (\Uparrow^{\mu}_{x}) S' \\ DS_x \vdash_{\mu DARTS} U \Uparrow^{\mu}_{x} (\Downarrow^{\mu}_{x}) U' \quad DS_x \vdash_{\mu DARTSD} \{DS\} \Uparrow^{\mu}_{x} (\Downarrow^{\mu}_{x}) \{DS'\} \end{array}}{DS_x \vdash_{\mu DARTSD} \{A : S..U; DS\} \Uparrow^{\mu}_{x} (\Downarrow^{\mu}_{x}) \{A : S'..U'; DS'\}} \text{ PDMR-}\textsc{Bnd}$$

Figure A.7: Promotion and **Demotion** for record types

$$\frac{\begin{array}{c}\Gamma; x : \mu(z : DS_z) \vdash_{\mu DARTS} \{ds_x\} \overleftarrow{:} \{DS_x\} \\ dom(ds_x) \text{ unique} \quad \Gamma \vdash_{\mu DARTS} \mu(z : DS_z) \Uparrow_V T\end{array}}{\Gamma \vdash_{\mu DARTS} \nu(z : DS_z)\{ds_z\}/V \overrightarrow{:} T} \textsc{ Syn-Obj}$$

$$\frac{\begin{array}{c}\Gamma; x : T \vdash_{\mu DARTS} t/V \overrightarrow{:} U \\ \Gamma \vdash_{\mu DARTS} T \Downarrow_V T'\end{array}}{\Gamma \vdash_{\mu DARTS} \lambda(x : T).t/V \overrightarrow{:} \forall(x : T')U} \textsc{ Syn-All-I} \qquad \frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \bot}{\Gamma \vdash_{\mu DARTS} x.a/V \overrightarrow{:} \bot} \textsc{ Syn-Obj-E1}$$

$$\frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \{DS\} \quad \{a : T\} \in \{DS\} \quad \Gamma \vdash_{\mu DARTS} U \Uparrow_V U'}{\Gamma \vdash_{\mu DARTS} x.a/V \overrightarrow{:} U'} \textsc{ Syn-Obj-E2}$$

$$\frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \mu(z : DS_z) \quad \{a : T\} \in \{DS_x\} \quad \Gamma \vdash_{\mu DARTS} U \Uparrow_V U'}{\Gamma \vdash_{\mu DARTS} x.a/V \overrightarrow{:} U'} \textsc{ Syn-Obj-E3}$$

$$\frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \bot \quad y \in dom(\Gamma)}{\Gamma \vdash_{\mu DARTS} x\ y/V \overrightarrow{:} \bot} \textsc{ Syn-All-E1}$$

$$\frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \forall(z : S)U_z \quad \Gamma \vdash_{\mu DARTS} y \overleftarrow{:} S \quad \Gamma \vdash_{\mu DARTS} U_y/V \overrightarrow{:} U'}{\Gamma \vdash_{\mu DARTS} x\ y/V \overrightarrow{:} U'} \textsc{ Syn-All-E2}$$

$$\frac{\Gamma \vdash_{\mu DARTS} t/\emptyset \overrightarrow{:} T \quad \Gamma; x : T \vdash_{\mu DARTS} u_x/V \cup x \overrightarrow{:} U}{\Gamma \vdash_{\mu DARTS} \text{let } x = t \text{ in } u/V \overrightarrow{:} U} \textsc{ Syn-Let1}$$

$$\frac{\Gamma \vdash_{\mu DARTS} t \overleftarrow{:} T \quad \Gamma; x : T \vdash_{\mu DARTS} u_x/V \cup x \overrightarrow{:} U}{\Gamma \vdash_{\mu DARTS} \text{let } x : T = t \text{ in } u/V \overrightarrow{:} U} \textsc{ Syn-Let2}$$

Figure A.8: The definition of type synthesis of terms

$$\frac{\begin{array}{c}\Gamma; x : \mu(z : DS_z) \vdash_{\mu DARTS} \{ds_x\} \overset{\leftarrow}{:} \{DS_x\} \\ dom(ds_x) \text{ unique} \quad \Gamma \gg \mu(z : DS_z) <: T \ll \Gamma\end{array}}{\Gamma \vdash_{\mu DARTS} \nu(z : DS_z)\{ds_z\} \overset{\leftarrow}{:} T} \ \textsc{Chk-Obj}$$

$$\frac{\Gamma \vdash_{\mu DARTS} S \Downarrow_{\forall(x:T')U_x} \quad \Gamma \gg T' <: T \ll \Gamma \quad \Gamma; x : T \vdash_{\mu DARTS} t_x \overset{\leftarrow}{:} U_x}{\Gamma \vdash_{\mu DARTS} \lambda(x : T).t_x \overset{\leftarrow}{:} S} \ \textsc{Chk-All-I1}$$

$$\frac{\begin{array}{c}\Gamma; x : T \vdash_{\mu DARTS} t_x/\emptyset \overset{\rightarrow}{:} U_x \\ \Gamma \gg \forall(x : T)U <: S \ll \Gamma\end{array}}{\Gamma \vdash_{\mu DARTS} \lambda(x : T).t_x \overset{\leftarrow}{:} S} \ \textsc{Chk-All-I2} \qquad \frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \bot}{\Gamma \vdash_{\mu DARTS} x.a \overset{\leftarrow}{:} U} \ \textsc{Chk-Obj-E1}$$

$$\frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \{DS\} \quad \{a : T\} \in \{DS\} \quad \Gamma \gg U <: U' \ll \Gamma}{\Gamma \vdash_{\mu DARTS} x.a \overset{\leftarrow}{:} U'} \ \textsc{Chk-Obj-E2}$$

$$\frac{\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \mu(z : DS_z) \quad \{a : T\} \in \{DS_x\} \quad \Gamma \gg U <: U' \ll \Gamma}{\Gamma \vdash_{\mu DARTS} x.a \overset{\leftarrow}{:} U'} \ \textsc{Chk-Obj-E3}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \bot \\ y \in dom(\Gamma)\end{array}}{\Gamma \vdash_{\mu DARTS} x\ y \overset{\leftarrow}{:} U} \ \textsc{Chk-All-E1} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mu DARTS} \Gamma(x) \Uparrow \forall(z : S)U_z \\ \Gamma \vdash_{\mu DARTS} y \overset{\leftarrow}{:} S \quad \Gamma \gg U_y <: U' \ll \Gamma\end{array}}{\Gamma \vdash_{\mu DARTS} x\ y \overset{\leftarrow}{:} U'} \ \textsc{Chk-All-E2}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mu DARTS} t/\emptyset \overset{\rightarrow}{:} T \\ \Gamma; x : T \vdash_{\mu DARTS} u_x \overset{\leftarrow}{:} U\end{array}}{\Gamma \vdash_{\mu DARTS} \text{let } x = t \text{ in } u \overset{\leftarrow}{:} U} \ \textsc{Chk-Let1} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mu DARTS} t \overset{\leftarrow}{:} T \\ \Gamma; x : T \vdash_{\mu DARTS} u_x \overset{\leftarrow}{:} U\end{array}}{\Gamma \vdash_{\mu DARTS} \text{let } x : T = t \text{ in } u \overset{\leftarrow}{:} U} \ \textsc{Chk-Let2}$$

Figure A.9: The definition of type checking of terms

Appendix B

Promotion and Demotion in $jDOT$

Definition B.1. *The definitions of* ***Promotion*** *and* ***Demotion*** *are shown in Figure B.1 and Figure B.2.*

For a judgment of ***Promotion*** $\Gamma \vdash_{jDOTS} S \Uparrow_V U$, Γ, S *and* V *are the inputs and* U *is the output. A* ***Demotion*** *judgment* $\Gamma \vdash_{jDOTS} S \Downarrow_V U$ *works similarly.*

$$\frac{}{\Gamma \vdash_{jDOTS} \top \Uparrow_V (\Downarrow_V) \top}\ \textsc{PD-Top} \qquad \frac{}{\Gamma \vdash_{jDOTS} T \Uparrow_V \top}\ \textsc{P-Top*}$$

$$\frac{}{\Gamma \vdash_{jDOTS} \bot \Uparrow_V (\Downarrow_V) \bot}\ \textsc{PD-Bot} \qquad \frac{}{\Gamma \vdash_{jDOTS} T \Downarrow_V \bot}\ \textsc{D-Bot*}$$

$$\frac{x \notin V}{\Gamma \vdash_{jDOTS} y.A \Uparrow_V (\Downarrow_V) y.A}\ \textsc{PD-Var} \qquad \frac{fv(D) \cap V = \emptyset}{\Gamma \vdash_{jDOTS} \mu D \Uparrow_V (\Downarrow_V) \mu D}\ \textsc{PD-Mu}$$

$$\frac{T \mapsto T_1 \ \&\ T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \bot \ \&\ T_4}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow_V \bot}\ \textsc{PD-Sel1}$$

$$\frac{T \mapsto T_1 \ \&\ T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \bot \ \&\ T_4}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow_V \bot}\ \textsc{P-Sel1}$$

$$\frac{T \mapsto T_1 \ \&\ T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \bot \ \&\ T_4}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Downarrow_V \top}\ \textsc{D-Sel1}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \ \&\ T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \{A : S..U\} \ \&\ T_4 \quad \Gamma_1' \vdash_{jDOTS} U \Uparrow_V U' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow_V U'}\ \textsc{P-Sel2}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \ \&\ T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \{A : S..U\} \ \&\ T_4 \quad \Gamma_1' \vdash_{jDOTS} S \Downarrow_V S' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Downarrow_V S'}\ \textsc{D-Sel2}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \ \&\ T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \mu\{w.A : S..U_w\} \ \&\ T_4 \quad \Gamma_1'; x : T_2 \wedge T_4 \vdash_{jDOTS} U_x \Uparrow_V U' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Uparrow_V U'}\ \textsc{P-Sel3}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \ \&\ T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \mu\{w.A : S_w..U\} \ \&\ T_4 \quad \Gamma_1'; x : T_2 \wedge T_4 \vdash_{jDOTS} S_x \Uparrow_V S' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.A \Downarrow_V S'}\ \textsc{D-Sel3}$$

Figure B.1: Definitions of **Promotion** and **Demotion** (Part 1)

$$\frac{\Gamma \vdash_{jDOTS} S \Downarrow_V (\Uparrow_V) S' \quad \Gamma \vdash_{jDOTS} U \Uparrow_V (\Downarrow_V) U'}{\Gamma \vdash_{jDOTS} \forall(z : S)U \Uparrow_V (\Downarrow_V) \forall(z : S')U'} \text{ PD-All}$$

$$\frac{\Gamma \vdash_{jDOTS} S \Downarrow_V (\Uparrow_V) S' \quad \Gamma \vdash_{jDOTS} U \Uparrow_V (\Downarrow_V) U'}{\Gamma \vdash_{jDOTS} \{A : S..U\} \Uparrow_V (\Downarrow_V) \{A : S'..U'\}} \text{ PD-Bnd}$$

$$\frac{\Gamma \vdash_{jDOTS} T \Uparrow_V (\Downarrow_V) T'}{\Gamma \vdash_{jDOTS} \{a : T\} \Uparrow_V (\Downarrow_V) \{a : T'\}} \text{ PD-Fld}$$

$$\frac{\Gamma \vdash_{jDOTS} S \Uparrow_V (\Downarrow_V) S' \quad \Gamma \vdash_{jDOTS} U \Uparrow_V (\Downarrow_V) U'}{\Gamma \vdash_{jDOTS} S \wedge U \Uparrow_V (\Downarrow_V) S' \wedge U'} \text{ PD-And}$$

Figure B.2: Definitions of **Promotion** and **Demotion** (Part 2)

Appendix C

General Term Typing Rules of $jDOT$

Definition C.1. *The definition of type synthesis is defined in Figure C.1, type checking is defined in Figure C.2 and type checking of definitions is defiend in Figure C.3.*

For a type synthesis judgment $\Gamma \vdash_{jDOTS} t/V \overrightarrow{:} T$, Γ, t and V are inputs and T is output.

For a type checking judgment $\Gamma \vdash_{jDOTS} t \overleftarrow{:} T$, Γ, t and T are inputs and the algorithm returns true if the derivation satisfies the definition.

For a type checking judgment $\Gamma \vdash_{jDOTS} t \overleftarrow{:}_x T$, Γ, t, x and T are inputs and the algorithm returns true if the derivation satisfies the definition.

$$\frac{\Gamma; x : T \vdash_{jDOTS} \{d_x\} \overset{\leftarrow}{:}_x T \quad \mathit{dom}(d_x) \text{ unique} \quad \Gamma \vdash_{jDOTS} T \Uparrow_V T'}{\Gamma \vdash_{jDOTS} \nu(z : T)\{d_z\}/V \overset{\rightarrow}{:} T'}\ \textsc{Syn-Obj}$$

$$\frac{\Gamma; x : T \vdash_{jDOTS} t/V \overset{\rightarrow}{:} U \quad \Gamma \vdash_{jDOTS} T \Downarrow_V T'}{\Gamma \vdash_{jDOTS} \lambda(x : T).t/V \overset{\rightarrow}{:} \forall(x : T')U}\ \textsc{Syn-All-I}$$

$$\frac{T \mapsto T_1 \,\&\, T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \bot \,\&\, T_4}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.a/V \overset{\rightarrow}{:} \bot}\ \textsc{Syn-Obj-E1}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \,\&\, T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \{a : U\} \,\&\, T_4 \quad \Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} U \Uparrow_V U' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.a/V \overset{\rightarrow}{:} U'}\ \textsc{Syn-Obj-E2}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \,\&\, T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \mu\{w.a : U_w\} \,\&\, T_4 \quad \Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} U_x \Uparrow_V U' \end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.a/V \overset{\rightarrow}{:} U'}\ \textsc{Syn-Obj-E3}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \,\&\, T_2 \\ \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \bot \,\&\, T_4 \quad y \in \mathit{dom}(\Gamma_2; x : T; \Gamma_1) \end{array}}{\Gamma_2; x : T; \Gamma_1 \vdash_{jDOTS} x\ y/V \overset{\rightarrow}{:} \bot}\ \textsc{Syn-All-E1}$$

$$\frac{\begin{array}{c} T \mapsto T_1 \,\&\, T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \forall(z : S)U_z \,\&\, T_4 \\ \Gamma_2; x : T; \Gamma_1 \vdash_{jDOTS} y \overset{\leftarrow}{:} S \quad \Gamma_2; x : T; \Gamma_1 \vdash_{jDOTS} U \Uparrow_V U' \end{array}}{\Gamma_2; x : T; \Gamma_1 \vdash_{jDOTS} x\ y/V \overset{\rightarrow}{:} U'}\ \textsc{Syn-All-E2}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{jDOTS} t/\emptyset \overset{\rightarrow}{:} T \\ \Gamma; x : T \vdash_{jDOTS} u_x/V \cup x \overset{\rightarrow}{:} U \end{array}}{\Gamma \vdash_{jDOTS} \text{let } x = t \text{ in } u/V \overset{\rightarrow}{:} U}\ \textsc{Syn-Let1} \qquad \frac{\begin{array}{c} \Gamma \vdash_{jDOTS} t \overset{\leftarrow}{:} T \\ \Gamma; x : T \vdash_{jDOTS} u_x/V \cup x \overset{\rightarrow}{:} U \end{array}}{\Gamma \vdash_{jDOTS} \text{let } x : T = t \text{ in } u/V \overset{\rightarrow}{:} U}\ \textsc{Syn-Let2}$$

Figure C.1: The definition of type synthesis of terms

$$\frac{\Gamma; x : T \vdash_{jDOTS} \{d_x\} \overset{\leftarrow}{:}_x T \quad dom(ds_x) \text{ unique} \quad \Gamma \gg T <: T' \ll \Gamma}{\Gamma \vdash_{jDOTS} \nu(z : T)\{d_z\}/V \overset{\rightarrow}{:} T'} \textsc{ Chk-Obj}$$

$$\frac{\begin{array}{c}\Gamma; x : T \vdash_{jDOTS} t_x/\emptyset \overset{\rightarrow}{:} U_x \\ \Gamma \gg \forall(x : T)U <: S \ll \Gamma\end{array}}{\Gamma \vdash_{jDOTS} \lambda(x : T).t_x \overset{\leftarrow}{:} S} \textsc{ Chk-All-I1} \qquad \frac{\begin{array}{c}\Gamma \gg S <: T \ll \Gamma \\ \Gamma; x : T \vdash_{jDOTS} t_x \overset{\leftarrow}{:} U_x\end{array}}{\Gamma \vdash_{jDOTS} \lambda(x : T).t_x \overset{\leftarrow}{:} \forall(x : S)U} \textsc{ Chk-All-I2}$$

$$\frac{T \mapsto T_1 \;\&\; T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \bot \;\&\; T_4}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.a \overset{\leftarrow}{:} VU} \textsc{ Chk-Obj-E1}$$

$$\frac{\begin{array}{c}T \mapsto T_1 \;\&\; T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \{a : U\} \;\&\; T_4 \quad \Gamma_1; x : T; \Gamma_2 \gg U <: U' \ll \Gamma_1; x : T; \Gamma_2\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.a \overset{\leftarrow}{:} VU'} \textsc{ Chk-Obj-E2}$$

$$\frac{\begin{array}{c}T \mapsto T_1 \;\&\; T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \\ T_3 \mapsto \mu\{w.a : U_w\} \;\&\; T_4 \quad \Gamma_1; x : T; \Gamma_2 \gg U_x <: U' \ll \Gamma_1; x : T; \Gamma_2\end{array}}{\Gamma_1; x : T; \Gamma_2 \vdash_{jDOTS} x.a \overset{\leftarrow}{:} VU'} \textsc{ Chk-Obj-E3}$$

$$\frac{T \mapsto T_1 \;\&\; T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \bot \;\&\; T_4 \quad y \in dom(\Gamma)}{\Gamma_2; x : T; \Gamma_1 \vdash_{jDOTS} x\ y \overset{\leftarrow}{:} U} \textsc{ Chk-All-E1}$$

$$\frac{\begin{array}{c}T \mapsto T_1 \;\&\; T_2 \quad \Gamma_1 \vdash_{jDOTS} T_1 \Uuparrow T_3 \dashv \Gamma_1' \quad T_3 \mapsto \forall(z : S)U_z \;\&\; T_4 \\ \Gamma \vdash_{jDOTS} y \overset{\leftarrow}{:} S \quad \Gamma_2; x : T; \Gamma_1 \gg U_y <: U' \ll \Gamma_2; x : T; \Gamma_1\end{array}}{\Gamma_2; x : T; \Gamma_1 \vdash_{jDOTS} x\ y \overset{\leftarrow}{:} U'} \textsc{ Chk-All-E2}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{jDOTS} t/\emptyset \overset{\rightarrow}{:} T \\ \Gamma; x : T \vdash_{jDOTS} u_x \overset{\leftarrow}{:} U\end{array}}{\Gamma \vdash_{jDOTS} \text{let } x = t \text{ in } u \overset{\leftarrow}{:} U} \textsc{ Chk-Let1} \qquad \frac{\begin{array}{c}\Gamma \vdash_{jDOTS} t \overset{\leftarrow}{:} T \\ \Gamma; x : T \vdash_{jDOTS} u_x \overset{\leftarrow}{:} U\end{array}}{\Gamma \vdash_{jDOTS} \text{let } x : T = t \text{ in } u \overset{\leftarrow}{:} U} \textsc{ Chk-Let2}$$

Figure C.2: The definition of type checking of terms

$$\frac{\Gamma \vdash_{jDOTS} t_x \overleftarrow{:} T_x}{\Gamma \vdash_{jDOTS} \{a = t_w\} \overleftarrow{:}_x \mu\{w.A : T_w\}} \textsc{ Def-Trm}$$

$$\frac{}{\Gamma \vdash_{jDOTS} \{A = T_w\} \overleftarrow{:}_x \mu\{w.A : T_w..T_w\}} \textsc{ Def-Typ}$$

$$\frac{\Gamma \vdash_{jDOTS} d_w \overleftarrow{:}_x S \quad \Gamma \vdash_{jDOTS} d'_w \overleftarrow{:}_x U}{\Gamma \vdash_{jDOTS} d_w \wedge d'_w \overleftarrow{:}_x S \wedge U} \textsc{ Def-And}$$

Figure C.3: The definition of definitions checking