Theory and Implementation of Coercive Subtyping

Tao Xue

Department of Computer Science
Royal Holloway, University of London

A Thesis Presented for the Degree of
Doctor of Philosophy

January 2013

DECLARATION OF AUTHORSHIP

I , Tao Xue, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:

Date:

ABSTRACT

Coercive subtyping is a useful and powerful framework of subtyping for type theories. In this thesis, we point out the problem in the old formulation of coercive subtyping in [Luo99], give a new and adequate formulation $T[\mathcal{C}]$, the system that extends a type theory T with coercive subtyping based on a set $\mathcal{C}$ of basic subtyping judgements, and show that coercive subtyping is a conservative extension and, in a more general sense, a definitional extension.

We introduce an intermediate system, the star-calculus $T[\mathcal{C}]^*$, in which the positions that require coercion insertions are marked, and show that $T[\mathcal{C}]^*$ is a conservative extension of T and that $T[\mathcal{C}]^*$ is equivalent to $T[\mathcal{C}]$. Further more, in order to capture all the properties of the coercive subtyping framework, we introduce another intermediate system $T[\mathcal{C}]_{0K}$ which does not contain the coercion application rules. We show that $T[\mathcal{C}]^*$ is actually a definitional extension of $T[\mathcal{C}]_{0K}$, which is a conservative extension of T. This makes clear what we mean by coercive subtyping being a conservative extension and amends a technical problem that has led to a gap in the earlier conservativity proof.

Another part of the work in this thesis concerns the implementation of coercive subtyping in the proof assistant Plastic. Coercive subtyping was implemented in Plastic by Paul Callaghan [CL01]. We have done some improvement based on that work, fixed some problems of Plastic, and implemented a new kind of data type called *dot-types*, which are special data types useful in formal semantics to describe interesting linguistic phenomena such as copredication, in Plastic.

ACKNOWLEDGEMENTS

First of all, I must thank my supervisor Professor Zhaohui Luo. He helped me to put my hands on the type theory and spent a lot of time on guiding me. He brought me a lot of ideas, taught me how to start my research and made me love this subject. He also gave me a lot of support for my British days.

Also, I would like to thank Dr. Robin Adams who always kindly offered helps on my study and gave me lots of advices. And thanks to Professor Sergei Soloviev in University of Toulouse, through his paper with Zhaohui and also our discussion, I've got the idea for the main work of this thesis. Thanks Herman Geuvers and David Aspinall for the useful discussion we had.

Thanks to all the members of type theory group at Royal Holloway. They helped me to make progress on my research. Thanks to all the staffs and research students in Department of Computer Science of Royal Holloway. They gave me a nice environment for my research.

I was financially supported by the Reid Scholarship of Royal Holloway, University of London.

Finally, I want to appreciate my parents for their support all these years and thank all my friends for their help through the days. All of you have shown me how wonderful the life is.

COPYRIGHT

The copyright in text of this thesis rests with the author. Any copy of this work, either fully or partly, should not be published without the permission of the author.

CONTENTS

1. INTRODUCTION

This chapter will introduce the background of type theory and the area of my interest. It will explain the significance of the work and major contributions as well. It also includes the structure of the thesis at the end of the chapter.

1.1 Basic Concepts of Type Theory

Type theory was originally introduced by logicians as a logical language for researching on the foundation of mathematics. Since the development of computer science, it has also become a powerful computational and logical language which could help us understand and design computer languages.

Compared with *elements* and *sets* in set theory, the basic concepts of type theory are *objects* and *types*. Informally, a type could be considered as a collection of data, and an object is a piece of data that belongs to the type. For example, the natural numbers constitute a type Nat, and 0 is an object of the type Nat. Take another example, we could consider $Human$ as a type, and each person is just an object of the type. Generally, presenting a relationship that object m is of type M, we write an assertion as following:

$$m : M$$

Mathematicians, logicians and computer scientists have introduce various kinds of type theories, such as the Simply Typed λ-Calculus ($\lambda_{\rightarrow}$) [Chu40], the Polymorphic λ-calculus F and the High-order Polymorphic λ-calculus F_ω [Gir72, Rey74] , the Calculus of Constructions (λC) [CH88], the Calculus of Inductive Constructions(CIC) [PM93], the Martin-Löf's Intuitionistic Type Theory [ML75, ML84], the Extended Calculus of Constructions(ECC) [Luo90] and the Unify Theory for Dependent Types(UTT) [Luo94].

Unlike other logical or mathematical languages, type theory is a language where computation is taken as the basic notion. A type-theoretic language with a rich type structure can provide nice abstraction mechanisms for modular development of programs, specifications and proofs. Also, type theory has a simple operational semantics, it is a more manageable language since its good proof-theoretic properties like decidability provide a good basis for the computer implementation. It could offer us certain helps in simplicity of computer languages as well. There are some other interesting use of type theory as well, for example, as will be shown in this thesis, we are also trying to use type theory for linguistic semantics.

1.2 Objects and Types

In type theory, the relationship between objects and types is a very interesting issue. There were many studies on this ([Sco70, ML75, ML84, dB80]) and showed some fantastic results which connected type theory with mathematics or computer programming, such as: types could be considered as problems, while objects as solutions; partial specification of a program could be thought as a type, and an implementation as an object of the type. As described by Martin-Löf in [ML84], some relations could be captured in figure 1.1:

A type	a: A
A is a type	a is an element of the type A
A is a proposition	a is a proof (construction) of the proposition A
A is an intention	a is a method of fulfilling the intention A
A is an expectation	a is a method of realizing the expectation A
A is a problem	a is a method of solving the problem A
A is a task	a is a method of doing the task A

Fig. 1.1: object and type

The most important result among these, is a principle called *propositions-*

as-types, also known as Curry-Howard correspondence (or isomorphism). It is a brilliant idea which connects type theory with logical systems. Basically, it says that a type could be considered as a logic proposition and an object of this type could be considered as a proof of the proposition. This idea was discovered by Curry[CF58] and Howard [How80].

Based on this principle, many type systems are connected with logical systems. For example:

- Simply Typed λ-calculus corresponds to the intuitionistic propositional logic.
- Girard's system F corresponds to the second order propositional logic, and system F_ω corresponds to the higher-order propositional logic.
- Calculus of Constructions, λC corresponds to the higher-order predicate logic.
- Unifying Theory of Dependent Types (UTT) contains an internal logic SOL which is a second order logic. It becomes higher-order logic, if we introduce Π-types.

For objects and types we can have several different studies on them. Given an object m, and type M: we can check whether m is of type M (type checking, e.g. [Car88]); we can find out the type of m; we can try to find out if M is well typed and give an object of M (decidability). Typing checking helps us in program verification and the problem of decidability in type theory could help us in program designing and in logic reasoning.

1.3 Inductive Data Types

We can define types inductively with certain rules, these types are called *inductive data types* . Inductive data types have been studied by Gentzen [Gen35] and Prawitz [Pra73, Pra74] for traditional logical systems, later by Martin-Löf [ML84], Backhouse [Bac88], Dybjer [Dyb91], Coquand and Mohring [CPM90], Luo [Luo94] in type theory.

For example, we can define a type $Week$ as follows. It is a trivial inductive data type consisting of finite many objects.

$$\begin{aligned} Week &= Monday \mid Tuesday \mid Wednesday \mid Thursday \\ &\quad \mid Friday \mid Saturday \mid Sunday \end{aligned}$$

Taking another example, we can define Nat to be a data type for all the numbers inductively in the following way,

$$Nat = 0|succ(Nat)$$

Intuitively, 0 is an object of Nat, and if n is an object of Nat, then $succ(n)$ is an object of Nat. More precisely, the inference rules should be

$$\frac{}{Nat : Type}$$

$$\frac{}{0 : Nat} \qquad \frac{n : Nat}{succ(n) : Nat}$$

We can define various of data types inductively in the similar way, like, Σ-type, Π-Type, types $Bool$, $List$, $Vector$ etc.

All these inductive data types consist of *constructors*. $Monday$, $Tuesday$, $Wednesday$, $Thursday$, $Friday$, $Saturday$ and $Sunday$ are called constructors of $Week$. $0 : Nat$ and $succ : Nat \to Nat$ are called constructors of Nat.

With the inductive data types, we have a special kind of objects, called *canonical objects*. Informally, as the relation we describe in Figure 1.2, the objects could be calculated into values with some reduction rules, like β-reduction, and the canonical objects are the values of objects of the type under computation. For example, for the type $Week$, $Monday$ to $Sunday$

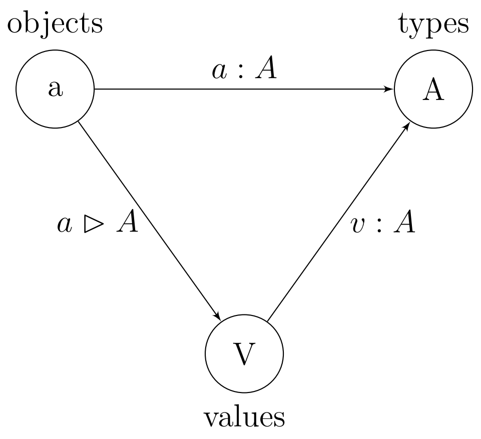

Fig. 1.2: type, object and value

are all its canonical objects; for the type Nat, a canonical natural number is either 0 or successor of a canonical natural number. Consider a more intuitive example: if we define an operation of addition '+' on natural number, using notation $1 \equiv succ(0)$ and $2 \equiv succ(succ(0))$, the natural number 1+1 is not canonical, it could compute to 2 as its value, which is a canonical object. In other words, a canonical object is an object that cannot be further computed and has itself as value.

This view of canonical objects is the basis to consider inductive data types in dependent type theories, each of which is equipped with some induction principles. In order to prove a property for all objects of the inductive type, one only has to prove it for all of its canonical objects. For example, in order to prove a property for all objects of type $Week$, we only have to prove it from $Monday$ to $Sunday$; if we prove a property of 0 and all canonical natural numbers inductively (if it holds for a canonical natural number m, then it holds for $succ(m)$), we can prove that it holds all natural numbers of type Nat.

Such type theories with canonical objects have the following property called canonicity [AMS07]:

- Canonicity: Any closed object of an inductive type is definitionally equal to a canonical object of that type.

Although Curry-Howard's isomorphism powerfully shows that propositions could be viewed as types, there's a feeling that it is not natural to view all the types to be propositions. For some of the types, e.g. Nat, though one could say it to be just true or a provable proposition, this seems too brute force and not reasonable enough. It would be more sensible to think data types such as the type of natural numbers are not logical propositions.

1.4 Different Views of Type Theory

In the literature, there are two different views of type theory: type theory with *type assignment* and type theory with *canonical objects*.

In the type theories with *type assignment*[Mil78], types are assigned to already defined terms. The types are polymorphic, informally saying, objects and types are not dependent on each other. Types could be used to assign to various objects and it is possible for an object to take different types. For example, $\lambda x.x : A \to A$ would hold for any type A.

In the type theories with *canonical objects*, types are considered consisting of its all canonical objects. Like in Martin-Löf's type theory or UTT, objects and their types depend on each other and cannot be thought of to exist independently. Like the type of natural number Nat, it consists of canonical objects 0 and $succ(n)$; numbers and Nat depend on each other, the numbers exist only because they are objects of Nat.

1.5 Logical Framework and UTT

As described above, there are different kinds of type theory. So a question comes: is there anyway that we can use a single meta-level framework to represent all these different types theories? The answer is yes, and one solution is called *logical framework*. There're several different versions of logical framework. The Edinburgh Logical Framework [HHP87] has been studied for formalization of logical systems based on the judgement-as-types principle. Martin-Löf's logical framework [NPS90] has been proposed for Martin-Löf's intensional type theory. The logical framework introduced by Zhaohui Luo in [Luo94] is a typed version of Martin-Löf's logical framework.

One should notice that a logical framework is a meta-language used to represent different type theories according to the users' requirements. A user needs to specify the type theory he needs in the form of the logical framework, in order to represent his type theory.

The Unified Theory of Types (UTT) [Luo94] is one of the type theories we represent in logical framework. UTT is an extension of Luo's extended calculus of constructions (ECC). It could be considered as a higher-order λ-calculus extended with inductive definition and rewriting, and with a hierarchy of type universes.

One advantage for us to use a meta-language to present the type theory, rather than presenting it directly, is that it allows us to give a clearer distinction between the language of the type theory and the meta-level mechanisms that are used to define the type theory, with the understanding that the former is the language that is to be used and the latter is a language that provides schematical mechanisms for language designers to specify languages and meta-level definitional mechanisms.

In particular, this presentation allows one to understand the structure of the conceptual universe of types in our type theory more clearly, and enables us to elaborate our views on some of the related technical and philosophical issues such as the notion of pure logical truth, hierarchical understanding of the language of type theory, intensionality of computational equality, and relationship between the logical universe and predicative universe.

1.6 Subtyping

Comparing type theory with set theory which has the notion subset, we would like to consider subtypes in type theory as well. Usually, we use $\leqslant$ or $<$ for the subtyping relation. We write $A \leqslant B$ (or $A < B$) for type A being a subtype of type B. There're a lot of works on subtyping in different ways based on different systems. For example, the system $F_{\leqslant}$ [CG92], the system $\lambda P_{\leqslant}$ [AC01], coercive subtyping [Luo97] ect. When we consider the subtyping problem, we are facing two questions:

1. How to introduce a subtyping relation between supertypes and subtypes?
2. How to use the subtyping relation?

1.6.1 How to Introduce Subtypes

To introduce subtype relations, one usually starts with an *initial set* of axioms of subtyping and considers general rules for subtyping and the type constructors. The subtyping relation is usually reflexive and transitive

$$\frac{A : \mathbf{Type}}{A \leqslant A}$$

$$\frac{A \leqslant B \qquad B \leqslant C}{A \leqslant C}$$

For the function types, subtyping is contravariant:

$$\frac{A_1 \leqslant A_2 \qquad B_2 \leqslant B_1}{A_2 \to B_2 \leqslant A_1 \to B_1}$$

Besides initial axioms, there are other different ways of getting subtype relation, some of which are discussed below.

Subset Types

One attempt [NPS90] is to generate subtype like what we do for subset in set theory: if A is a type, $B(x)$ is a type under the assumption that $x : A$, then $\{x : A|B(x)\}$ will also be a type and it's a subtype of type A. The formation and introduction rules are:

$$(S1)\frac{A : \mathbf{Type} \qquad \begin{matrix}[x : A] \\ B(x) : \mathbf{Type}\end{matrix}}{\{x : A|B(x)\} : \mathbf{Type}}$$

$$(S2)\frac{a : A \qquad b : B(a)}{a : \{x : A|B(x)\}}$$

This seems to be an natural idea and would be perfect if it works. But it is unfortunately problematic. In the introduction rule (S2), we have $b : B(a)$ in the premises, but in the conclusion $a : \{x : A|B(x)\}$, we lose the information of b. If we want to use this rule for constructive reasoning which goes from bottom to up, we cannot decide what the b is from only $a : \{x : A|B(x)\}$.

Constructor Subtyping

Another way to introduce subtyping is called *constructor subtyping* for inductive types[BF99, BvR00]. It specifies a subtype by declaring its constructor to be a subset of the constructors of an existing supertype. More precisely, the idea is that an inductive type A is viewed as a subtype of another type B if B has more constructors than A. Let's take a simple example with the type $Week$. We could define another data type $Weekend$, with only $Saturday$ and $Sunday$:

$$Weekend = Saturday \mid Sunday$$

The type *Week* has constructors *Monday*, *Tuesday*, *Wednesday*, *Thursday*, *Friday*, *Saturday*, and *Sunday*, and the type *Weekend* has constructors *Satur-*

day and *Sunday*. It's very clear by the definition that *Weekend* is a subtype of *Week*.

We could consider another example with the type Nat. Based on the definition of Nat, we could define another data type $NonZero$:

$$NonZero = succ(Nat)$$

Since Nat has constructors 0 and $succ(Nat)$, and $NonZero$ has only constructor $succ(Nat)$, $NonZero$ is a subtype of Nat, which is also a very reasonable relation.

Constructor subtyping could describe the subtypes relations in some data types. But this would exclude some interesting applications of subtyping. One might want to think of the type *Even* which is trivially known as a subtype of Nat. We can hardly use the subset of the constructors of Nat above to represent the type *Even*.

Projective Subtyping

Projective subtyping is based on 'projections', from a type of pairs (or a record type) to a component type. For a product type $A \times B$, we can think of subtyping relation:

$$A \times B \leqslant A \;\; \textit{or} \;\; A \times B \leqslant B.$$

For a Σ-type $\Sigma(A, B)$, we can think of subtyping relation [LL05]:

$$\Sigma(A, B) \leqslant A$$

For a record type $< R, l : A >$, we can think of subtyping relation [BT98, Luo09a]:

$$< R, l : A > \leqslant R \;\; \textit{and} \;\; < R, l : A > \leqslant < l : A > .$$

We can consider a more concrete example, let $P : (Nat)Prop$ be a predicate over Nat. The Σ-type, $\Sigma(Nat, P)$ is sometimes used to represent the subtype of Nat of those natural numbers n such that $P(n)$ holds. Now, it would be natural to consider the following subtyping relation:

$$\Sigma(Nat, P) \leqslant Nat$$

Structural Subtyping for Inductive Types

Structural subtyping is a natural subtyping relation for an inductive type and has been studied for arbitrary inductive types in [LL05, LA08]. The basic idea is that the subtype relation should pass through some data structures. Consider the example of lists. The structural subtyping relationship for lists is that, if A is a subtype of B, then $List(A)$ is a subtype of $List(B)$. This would be expressed by means of the following rule:

$$\frac{A \leqslant B}{List(A) \leqslant List(B)}$$

1.6.2 How to Use Subtypes

Subtyping with Subsumption Rule

One traditional and basic mechanism for subtyping in type theories is *subsumption.* Intuitively, the basic idea of this mechanism says that, a type A is the subtype of type B if all the objects of A are also objects of B. More precisely, if A is subtype of B, and an object a is of type A, then a is of type B as well. Formally, it should be written as the following rule, which is called *subsumption rule*:

$$\frac{a : A \qquad A \leqslant B}{a : B}$$

It is a quite natural rule in type theories with *type assignment.* It is fairly sensible for an object of the subtype to inherit the type of the supertype. However, this rule would be problematic when we consider the type theories with canonic objects: it would destroy the canonicity of property, as we will discuss in detail in chapter 3.

Coercive Subtyping

Coercive subtyping [Luo97, Luo99] is an adequate approach to introduce subtyping into the type theories with canonical objects such as Martin-löf's type theory [ML84, NPS90] and UTT [Luo94]. The basic idea is that subtyping is an abbreviation mechanism: given two different types A and B, we can make A into a subtype of B by declaring a function c from A to B to be a coercion, meaning that any object a of type A is identified with the object $c(a)$ of type B in any context that requires an object of type B. We use $A <_c B$ to notate that there's a coercion c from A to B. We can also think it in this way: when we want to apply a function to an object, if we find there's a gap between them, then, we use the coercion to fill the gap to complete the application. Writing it as a rule, we should have:

$$\frac{f : B \to C \quad a : A \qquad A <_c B}{f(a) = f(c(a)) : C}$$

Unlike subsumption subtyping, in coercive subtyping, object a does not obtain any more types. Although $f(a)$ is of type C, a itself is *not* of type B: it is only used as an object of type B when the context is required. And $f(a)$ is just an abbreviation of and definitionally equal to $f(c(a))$ which is already of type C.

Coercive subtyping is a quite powerful mechanism in the study of type theory. Whether the type system extended with coercive subtyping is consistent and conservative is an interesting question. Since we think coercive subtyping is just an abbreviation mechanism, we believe it should not bring any extra power to the original type system. We will investigate this problem in this thesis.

1.7 Type Theory and Linguistic Semantics

Type theory is not only useful in mathematics and computer language study, it is also used in natural language study for linguistic semantics.

Montague grammar [Mon74] is an approach to natural language semantics by Richard Montague. It is based on Church's simple type theory [Chu40]. In Montague grammar, there is a universal type e of entities: a common noun or a verb is interpreted as a function of type $e \to t$ and an adjective as a function of type $(e \to t) \to (e \to t)$, where t is the type of truth values.

Type-theoretical semantics was studied by Ranta [Ran94] for developing formal semantics based on Martin-Löf's type theory. In type-theoretical semantics, one of the most basic differences with Montague grammar is that, common nouns are interpreted as types. For example, the interpretations of man, human and book are types:

$$[[human]], [[book]] : Type$$

In a type-theoretical semantics, we use a modern type theory, which is many-sorted in the sense that there are many types like $[[human]]$ and $[[book]]$ consisting of objects standing for different sorts of entities, while in Montague grammar, the simple type theory may be thought of as single sorted in the sense that there is the type e of all entities.

Verbs and adjectives are interpreted as predicates in a type-theoretical semantics. For example, we have:

$$\begin{array}{rcl} [[heavy]] & : & [[book]] \to Prop \\ [[read]] & : & [[human]] \to [[book]] \to Prop \end{array}$$

However, type-theoretical semantics brings a limitation of expressive power as well. Compared with Montague grammar on functional subsets, the operations on types are fewer. Type theoretical semantics with coercive subtyping [Luo10] provides a powerful way to extend type theories with more expressive powers for formal semantics.

1.8 Implementation: Proof Assistants

A proof assistant [Geu09] is a software to assist with the formal proofs in a man-machine interactive way. One can define mathematical problems in the provided formal language, choose the right strategy or algorithms in the library to achieve the proof. Although it is not fully automatic, a proof assistant can save a lot of job, and more importantly, it checks every small detail of the proof hence guarantees that the proof is correct.

$$\begin{tikzcd} input \arrow[r] & \overset{human\ guide}{proof\ assistant} \arrow[r] & result \end{tikzcd}$$

There are various implementations of proof assistants, like Lego [LP92], Coq [Coq10], Agda [Agd08], Matita [Mat08], Plastic [CL01]. They are based on different type systems, Coq and Matita are based on the Calculus of (Co)Inductive Constructions [PM93], Agda is on Martin-Löf's intuitionistic type theory [ML84]. Lego and Plastic are on Luo's UTT [Luo94]. Plastic, like many of the proof assistants, uses the Proof General [Pro12] which is a general front-end for proof assistants based on the customizable text editor Emacs.

Coercive subtyping has been implemented in some of the proof assistants. Saïbi has introduced this into Coq [Saï97], Bailey has done this for Lego [Bai99], Callaghan has implemented it in Plastic [CL01] , and Matita [Mat08] has also implemented implicit coercions.

Saïbi introduced an inheritance mechanism and implemented coercions in Coq. In his mechanism, he has the standard coercive subtyping: allow to apply $f : forall\ x : A, B$ to $a' : A'$ when A' is a subtype of A, written as $f(a)$. He also introduced two abstract classes: Funclass and Sortclass. Funclass is a class of functions, whose objects are all terms with a function type; it allows us to write $f(x)$ when f is not a function but can be seen in a certain sense as a function such as bijection, functor, any structure morphism etc. Sortclass is the class of sorts, whose objects are the terms whose type is a sort(Prop,Set,Type), which allows to write $x : A$ where A is not a type, but can be seen in a certain sense as a type such as set, group, category etc. There're also special cases of coercions call Identity Coercion which are used to go around the uniform inheritance condition. Coercions in Coq are represented by a coercion graph with coercion path, which could guarantee the satisfaction of transitivity and coherence.

Bailey has introduced coercions into Lego system. He has three different kinds of coercion. Apart from the standard coercions, he introduced kind coercion and Π-coercions. Kind coercion el coerces a non-type object into a type; Π-coercions use coercion c to coerce an object a of type A into a function $c(a)$.

In Callaghan's Plastic, the three different kinds of coercions above are implemented, furthermore, it could support parameterized coercions, dependent coercions and coercions rules, like

$$\frac{A <_c B}{List(A) <_c List(B)}$$

The implementation work in this thesis will be based on the proof assistant Plastic.

1.9 Motivation and Contributions

The main work and contributions of the thesis are:

- We study an adequate formulation of coercive subtyping. In particular, we give a counter-example to show that the earlier formulation, in which basic coercions may be generated by arbitrary basic subtyping rules [Luo99], is inadequate in that there are 'bad rules that can violate the conservativity property and may even lead to logical inconsistency of the extension. The new formulation solves the problem. It only considers basic coercions by means of sets of coercion judgements, which not only can be shown to be a conservative extension but captures the general formulation by means of coercion rules as well.

- As we said, coercive subtyping would naturally be considered a conservative extension. However, because of the special syntax in coercive subtyping, to describe the notion of conservativity is not straightforward and, because of this, it has not been made clear in [SL02] and, in fact, there was a gap of the conservativity proof presented in [SL02]. In order to describe clearly the relationship between original type system and its extension by coercive subtyping, we introduce an intermediate system called *star-calculus*, which marks the positions that require coercion insertions with the $*$-symbol. After introducing these new systems, we find that the method in [SL02] could go through with some modification. We inherit the main proof idea in [SL02], prove that the star-calculus is equivalent to the coercive subtyping extension and conservative over the original type system, and furthermore, it is a definitional extension in a certain sense.
- We have conducted implementation work that improve the coercive subtyping in proof assistant Plastic, modify the implementation code and fix bugs in transitivity and coherence checking. Based on the implementation of coercive subtyping, we have also implemented the dot-types in Plastic, as studied in formal semantics of natural languages that are used to describe interesting linguistic phenomena such as copredication.

Part of the work as summarized above has been described in the papers [LSX13] and [XL12].

1.10 Overview of the Thesis

Chapter 2 presents the formal details of Luo's UTT specified by a typed version of Martin-Löf's logical framework. UTT includes an internal logic, a collection of inductive data types generated by inductive schemata, and an infinite number of predicative universes. We also show some data types specified in LF which will be used in the rest of the thesis.

In Chapter 3, we discuss the basic idea of coercive subtyping and compare it with another basic subtyping mechanism – subtyping with subsumption rule. We give the original formulation of coercive subtyping introduced by Luo in [Luo99], study the importance of coherence and conservativity, and give a counter example to show the original formulation is problematic. We analyze the reason for the problem and propose the solutions to solve the problems.

We give a new formulation of coercive subtyping in Chapter 4, and introduce the intermediate system with star-calculus. We prove the intermediate system is equivalent to the coercive subtyping extension and conservative over the original type theory. We also prove some other meta-properties.

In Chapter 5, we explain the implementation of coercive subtyping in proof assistant Plastic with some examples, the existing problems and the improvements we have done for coercions in Plastic.

Chapter 6 studies a special kind of types called dot-types, gives their type-theoretical definition with inference rules and explain their uses in linguistic semantics. We also show their implementation in Plastic together with examples.

Finally, conclusions and future work are discussed in Chapter 7.

2. LOGICAL FRAMEWORK AND UTT

In this chapter, as a background of our main work, we give a formal description of Zhaohui Luo's Logical Framework – LF, and Unifying Theory of Types – UTT [Luo94]. UTT is an intensional type theory specified by LF, a typed version of Martin-Löf's logical framework. It includes an internal logic, a class of inductive data types generated by inductive schemata, and an infinite number of predicative universes. We will also show how to specify some often used inductive data types in examples.

2.1 Logical Framework

Logical frameworks are introduced because we want a single framework as meta-language to represent variant kinds of type theory. There're several different versions of logical framework. The Edinburgh Logical Framework [HHP87] is based on the judgement-as-types principle and comprises a formal system for a formal presentation of logical systems. Martin-Löf's logical framework [NPS90] was introduced for Martin-Löf's intensional type theory. The logical framework we use in this thesis, is a typed version of Martin-Löf's logical framework, which was introduced by Luo in [Luo94], in which the functional abstractions of the form $(x)k$ are replaced by typed $[x : K]k$. We will simply call it LF in the rest of this thesis.

2.1.1 The Logical Framework

LF is a type system with terms of the following forms:

$$\mathbf{Type},\ El(A),\ (x:K)K',\ [x:K]k',\ f(k)$$

The kind **Type** denotes the conceptual universe of types; $El(A)$ denotes the kind of objects of type A; $(x\colon K)K'$ denotes a dependent product; $[x\colon K]k'$ denotes an abstraction; and $f(k)$ denotes an application. The free occurrences of the variable x in K' and k' are bound by the binding operators $(x:K)$ and $[x:K]$.

Since LF is used as a meta-language to specify type theories, the types in LF are called *kinds*, and **Type** is a special kind in LF, in order to distinguish them from the types in the specified type theories. There are five forms of judgements in LF:

- $\Gamma \vdash \mathbf{valid}$, which asserts that Γ is a valid context.
- $\Gamma \vdash K\ \ \mathbf{kind}$, which asserts that K is a valid kind.
- $\Gamma \vdash k : K$, which asserts that k is an object of kind K.
- $\Gamma \vdash k = k' : K$, which asserts that k and k' are equal objects of kind K.
- $\Gamma \vdash K = K'$, which asserts that K and K' are two equal kinds.

Figure 2.1 shows the rules in LF, which contains the rules for context validity and assumptions, the general equalities rules, the type equalities rules, the substitution rules, the rules for kind Type and the rules for dependent product kinds.

Definition 2.1. *(types, kinds, and small kinds) A is called a* Γ*-type if* $\Gamma \vdash A : \mathbf{Type}$ *and* K *is called a* Γ*-kind if* $\Gamma \vdash K\ \ \mathbf{kind}$*. A* Γ*-kind is called small if it is either of the form El(A) or of the form* $(x : K_1)K_2$ *for some small* Γ*-kind* K_1 *and small* $(\Gamma, x : K_1)$*-kind* K_2

Contexts and assumptions

$$(1.1)\frac{}{<>\vdash \textbf{valid}} \qquad (1.2)\frac{\Gamma\vdash K\ \textbf{kind}\quad x\notin FV(\Gamma)}{\Gamma, x:K\ \vdash \textbf{valid}} \qquad (1.3)\frac{\Gamma, x:K,\Gamma'\vdash \textbf{valid}}{\Gamma, x:K,\Gamma'\vdash x:K}$$

General equality rules

$$(2.1)\frac{\Gamma\vdash K\ \textbf{kind}}{\Gamma\vdash K=K} \qquad (2.2)\frac{\Gamma\vdash K=K'}{\Gamma\vdash K'=K} \qquad (2.3)\frac{\Gamma\vdash K=K'\quad \Gamma\vdash K'=K''}{\Gamma\vdash K=K''}$$

$$(2.4)\frac{\Gamma\vdash k:K}{\Gamma\vdash k=k:K} \qquad (2.5)\frac{\Gamma\vdash k=k':K}{\Gamma\vdash k'=k:K} \qquad (2.6)\frac{\Gamma\vdash k=k':K\quad \Gamma\vdash k'=k'':K}{\Gamma\vdash k=k'':K}$$

Equality typing rules

$$(3.1)\frac{\Gamma\vdash k:K\quad \Gamma\vdash K=K'}{\Gamma\vdash k:K'} \qquad (3.2)\frac{\Gamma\vdash k=k':K\quad \Gamma\vdash K=K'}{\Gamma\vdash k=k':K'}$$

Substitution rules

$$(4.1)\frac{\Gamma, x:K,\Gamma'\ \vdash \textbf{valid}\quad \Gamma\vdash k:K}{\Gamma,[k/x]\Gamma'\ \vdash \textbf{valid}}$$

$$(4.2)\frac{\Gamma, x:K,\Gamma'\vdash K'\ \textbf{kind}\quad \Gamma\vdash k:K}{\Gamma,[k/x]\Gamma'\vdash [k/x]K'\ \textbf{kind}} \qquad (4.3)\frac{\Gamma, x:K,\Gamma'\vdash k':K'\quad \Gamma\vdash k:K}{\Gamma,[k/x]\Gamma'\vdash [k/x]k':[k/x]K'}$$

$$(4.4)\frac{\Gamma, x:K,\Gamma'\vdash K'=K''\quad \Gamma\vdash k:K}{\Gamma,[k/x]\Gamma'\vdash [k/x]K'=[k/x]K''} \qquad (4.5)\frac{\Gamma, x:K,\Gamma'\vdash k'=k'':K'\quad \Gamma\vdash k:K}{\Gamma,[k/x]\Gamma'\vdash [k/x]k'=[k/x]k'':[k/x]K'}$$

$$(4.6)\frac{\Gamma, x:K,\Gamma'\vdash K'\ \textbf{kind}\quad \Gamma\vdash k=k':K}{\Gamma,[k/x]\Gamma'\vdash [k/x]K'=[k'/x]K'} \qquad (4.7)\frac{\Gamma, x:K,\Gamma'\vdash k':K'\quad \Gamma\vdash k_1=k_2:K}{\Gamma,[k_1/x]\Gamma'\vdash [k_1/x]k'=[k_2/x]k':[k_1/x]K'}$$

The kind Type

$$(5.1)\frac{\Gamma\ \vdash \textbf{valid}}{\Gamma\vdash \textbf{Type kind}} \qquad (5.2)\frac{\Gamma\vdash A:\textbf{Type}}{\Gamma\vdash El(A)\ \textbf{kind}} \qquad (5.3)\frac{\Gamma\vdash A=B:\textbf{Type}}{\Gamma\vdash El(A)=El(B)}$$

Dependent product kinds

$$(6.1)\frac{\Gamma\vdash K\ \textbf{kind}\quad \Gamma, x:K\vdash K'\ \textbf{kind}}{\Gamma\vdash (x:K)K'\ \textbf{kind}} \qquad (6.2)\frac{\Gamma\vdash K_1=K_2\quad \Gamma, x:K_1\vdash K_1'=K_2'}{\Gamma\vdash (x:K_1)K_1'=(x:K_2)K_2'}$$

$$(6.3)\frac{\Gamma, x:K\vdash k:K'}{\Gamma\vdash [x:K]k:(x:K)K'} \qquad (6.4)(\xi)\frac{\Gamma\vdash K_1=K_2\quad \Gamma, x:K_1\vdash k_1=k_2:K}{\Gamma\vdash [x:K_1]k_1=[x:K_2]k_2:(x:K_1)K}$$

$$(6.5)\frac{\Gamma\vdash f:(x:K)K'\quad \Gamma\vdash k:K}{\Gamma\vdash f(k):[k/x]K'} \qquad (6.6)\frac{\Gamma\vdash f=f':(x:K)K'\quad \Gamma\vdash k_1=k_2:K}{\Gamma\vdash f(k_1)=f'(k_2):[k_1/x]K'}$$

$$(6.7)(\beta)\frac{\Gamma, x:K\vdash k':K'\quad \Gamma\vdash k:K}{\Gamma\vdash ([x:K]k')(k)=[k/x]k':[k/x]K'} \qquad (6.8)(\eta)\frac{\Gamma\vdash f:(x:K)K'\quad x\notin FV(f)}{\Gamma\vdash [x:K]f(x)=f:(x:K)K'}$$

Fig. 2.1: The inference rules of LF

Notation 2.2. *We shall use the following notations:*

- *We often write $(K)K'$ instead of $(x : K)K'$ if x does not occur free in K.*
- *We write A for $El(A)$ and $(A)B$ for $(El(A))El(B)$ when no confusion may occur.*
- Substitution

  We write $[N/x]M$ for the expression obtained from M by substituting N for the free occurrences of variable x in M, defined as usual with possible changes of bound variables to avoid variable capture. Informally, we sometimes use $M[x]$ to denote an expression M in which the variable x may occur free in M and subsequently write $M[N]$ for $[N/x]M$, when no confusion may occur.
- Function composition

  If $F : (x : K_1)K_2$ and $G : (y : K_2)K_3$, we define

  $$G \circ F \equiv_{df} [x : K_1]G(F(x)) : (x : K_1)[F(x)/y]K_3$$
- *We write $\Gamma \overset{d}{\vdash} J$, to denote that d is a derivation with final judgement $\Gamma \vdash J$.*

2.1.2 Specifying Type Theory in LF

LF can be used to specify type theories, such as Martin-Löf's type theory [NPS90] and UTT [Luo94]. In general, a specification of a type theory in LF consists of a collection of new *constants* and new *computation rules.* Formally, we declare a new constant k of kind K by writing

$$k : K$$

This has the effect of introducing the following inference rule:

$$\frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash k : K}$$

We declare a new computation rule by writing

$$k = k' : K \ \ where \ \ k_i : K_i \ (i = 1, \ldots, n)$$

This has the effect of introducing the following inference rule:

$$\frac{\Gamma \vdash k_i : K_i \ (i = 1, \ldots, n) \quad \Gamma \vdash k : K \quad \Gamma \vdash k' : K}{\Gamma \vdash k = k' : K}$$

Example 2.3. *To introduce the type N of natural numbers, we declare the following constants:*

$$\begin{aligned} N &: \textbf{Type} \\ 0 &: N \\ succ &: (N)N \\ Rec_N &: (C : (N)\textbf{Type})(c : C(0))(f : (x : N)(C(x))C(succ(x)))(n : N)C(n) \end{aligned}$$

and the computation rules:

$$Rec_N(C, c, f, 0) = c : C(0)$$

$$Rec_N(C, c, f, succ(n)) = f(n, Rec_N(C, c, f, n)) : C(succ(n))$$

where $C : (N)\textbf{Type}, c : C(0), f : (x : N)(C(x))C(succ(x)), n : N$.

So, the corresponding inference rules are:

$$(N1) \quad \frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash N : \textbf{Type}}$$

$$(N2) \quad \frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash 0 : N}$$

$$(N3)\quad \frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash succ : (N)N}$$

$$(N4)\quad \frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash Rec_N : (C : (N)\textbf{Type})(c : C(0))(f : (x : N)(C(x))C(succ(x)))(n : N)C(n)}$$

$$(N5)\quad \frac{\begin{array}{c}\Gamma \vdash C : (N)\textbf{Type} \quad \Gamma \vdash f : (x : N)(C(x))C(succ(x)) \\ \Gamma \vdash c : C(0) \quad \Gamma \vdash Rec_N(C, c, f, 0) : C(0)\end{array}}{\Gamma \vdash Rec_N(C, c, f, 0) = c : C(0)}$$

$$(N6)\quad \frac{\begin{array}{c}\Gamma \vdash C : (N)\textbf{Type} \quad \Gamma \vdash c : C(0) \quad \Gamma \vdash f : (x : N)(C(x))C(succ(x)) \\ \Gamma \vdash n : N \quad \Gamma \vdash succ : (N)N \\ \Gamma \vdash Rec_N(C, c, f, succ(n)) : C(succ(n)) \quad \Gamma \vdash f(n, Rec_N(C, c, f, n)) : C(succ(n))\end{array}}{\Gamma \vdash Rec_N(C, c, f, succ(n)) = f(n, Rec_N(C, c, f, n)) : C(succ(n))}$$

2.2 The Formulation of UTT

UTT is a type theory consisting of an impredicative universe of logical propositions, a large class of inductive data types, and an infinite number of predicative universes.

2.2.1 The Internal Logical Mechanism

The internal logic of UTT contains a universe *Prop* of logical propositions and their proof types. They are introduced by declaring the following constants.

$$\begin{array}{rcl}
Prop & : & \textbf{Type} \\
\textbf{Prf} & : & (Prop)\textbf{Type} \\
\forall & : & (A : \textbf{Type})((A)Prop)Prop \\
\Lambda & : & (A : \textbf{Type})(P : (A)Prop)((x : A)\textbf{Prf}(P(x)))\textbf{Prf}(\forall(A, P)) \\
\mathbf{E}_\forall & : & (A : \textbf{Type})(P : (A)Prop)(R : (\textbf{Prf}(\forall(A, P)))Prop) \\
 & & ((g : (x : A)\textbf{Prf}(P(x)))\textbf{Prf}(R(\Lambda(A, P, g)))) \\
 & & (z : \textbf{Prf}(\forall(A, P)))\textbf{Prf}(R(z))
\end{array}$$

and the computation rule:

$$\mathbf{E}_\forall(A, P, R, f, \Lambda(A, P, g)) = f(g) : Prf(R(\Lambda(A, P, g)))$$

The logical universe *Prop* is impredicative since universal quantification $\forall(A, P)$ can be formed for any type A and (meta-level) predicate P over A. In particular, A can be *Prop* itself or more complex. It is worth remarking that one cannot use $\forall$ to quantify over meta-level kinds such as **Type**, (A)**Type**, or $(A)Prop$.

Notation 2.4. *We will notate* $\forall(A, P)$ *with* $\forall x : A.P(x)$ *and* $\Lambda(A, P, f)$ *with* $\Lambda x : A.f(x)$ *when no confusion may occur.*

A usual elimination (application) operator **App** of kind $(A : Type)(P : (A)Prop)(\mathbf{Prf}(\forall(A, P)))(a : A)\mathbf{Prf}(P(a))$ can be defined as:

$$\mathbf{App}(A, P, F, a) =_{df} \mathbf{E}_\forall(A, P, [G : \mathbf{Prf}(\forall(A, P))]P(a), [g : (x : A)\mathbf{Prf}(P(x))]g(a), F)$$

which satisfies the equality (β-rule for Λ and **App**)

$$\mathbf{App}(A, P, \Lambda(A, P, g), a) = g(a) : Prf(P(a))$$

The usual logical operators could be defined as follows, where P_1 and P_2 are propositions, A is a type, $P : (A)Prop$.

$$\begin{aligned}
P_1 \supset P_2 &=_{df} \forall x : P_1.P_2 \\
true &=_{df} \forall X : Prop.(X \supset X) \\
false &=_{df} \forall X : Prop.X \\
P_1 \wedge P_2 &=_{df} \forall X : Prop.(P_1 \supset P_2 \supset X) \supset X \\
P_1 \vee P_2 &=_{df} \forall X : Prop.(P_1 \supset X) \supset (P_2 \supset X) \supset X \\
\neg P_1 &=_{df} P_1 \supset false \\
\exists x : A.P(x) &=_{df} \forall X : Prop.(\forall x : A.(P(x) \supset X) \supset X
\end{aligned}$$

2.2.2 Inductive Types

Definition 2.5. *(inductive schemata) Let Γ be a valid context and X be a variable such that $X \notin FV(\Gamma)$*

- *Φ is a strictly positive operator in Γ w.r.t. X, notation $Pos_{\Gamma;X}(\Phi)$, if Φ is of the form $(x_1 : K_1)...(x_n : K_n)X$, where $n \geqslant 0$ and K_i is a small ($\Gamma, x_1 : K_1, ...x_{i-1} : K_{i-1}$)-kind for i = 1,...,n*
- *Θ is an inductive schema in Γ w.r.t. X, if it is of one of the following forms*
  1. *$\Theta \equiv X$,*
  2. *$\Theta \equiv (x : K)\Theta_0$, where K is a small Γ-kind and Θ_0 is an inductive schema in $\Gamma, x : K$ w.r.t. X,*
  3. *$\Theta \equiv (\Phi)\Theta_0$ where $Pos_{\Gamma;X}(\Phi)$ and Θ_0 is an inductive schema in Γ w.r.t. X.*

An inductive schema w.r.t. X is of the form $(x_1 : M_1)...(x_m : M_m)X$ where M_i is either a small kind such that $X \notin FV(M_i)$ or a strictly positive operator w.r.t. X. A strictly positive operator is an inductive schema where $M_i's$ are all small kinds such that $X \notin FV(M_i)$. Using inductive schemata to introduce types into type theory, the smallness condition of the kinds occurring in inductive schemata is important. For example, neither $(\mathbf{Type})X$ nor $((A)\mathbf{Type})X$ is an inductive schema since **Type** is not a small kind.

Notation 2.6. *We shall write $\Phi[A]$ and $\Theta[A]$ for $[A/X]\Phi$ and $[A/X]\Theta$, and $\bar{\Theta}$ for a sequence of inductive schemata $\Theta_1, ...\Theta_n$ $(n \geqslant 0)$.*

Definition 2.7.

- *Let $\Theta \equiv (x_1 : M_1)...(x_m : M_m)X$ be an inductive schema and $\langle \Phi_{i_1}, ..., \Phi_{i_k} \rangle$ the subsequence of $\langle M_1, ...M_m \rangle$, which consists of the strictly positive* *operators. Then for A : **Type**, $C : (A)$**Type** and $z : \Theta(A)$, we could define kind $\Theta^\circ[A, C, z]$ as follows:*

$$\begin{aligned}\Theta^\circ[A,C,z] \;\;=_{df}\;\; & (x_1 : M_1[A])...(x_m : M_m[A]) \\ & (\Phi^\circ_{i_1}[A,C,x_{i_1}])...(\Phi^\circ_{i_k}[A,C,x_{i_k}])C(z(x_1,...,x_m))\end{aligned}$$

In the special case when Θ is a strictly positive operator Φ (i.e.$X \notin FV(M_1,...,M_m)$), $\Theta^\circ[A,C,z] \equiv (x_1 : M_1)...(x_m : M_m)C(z(x_1,...,x_m))$.

- *Let $\Phi \equiv (x_1 : M_1)...(x_m : M_m)X$ be a strictly positive operator wi.r.t. X. Define $\Phi^\natural[A,C,f,z]$ of kind $\Phi^\circ[A,C,z]$ for A : **Type**, $C : (A)$**Type**, $f : (x : A)C(x)$ and $z : \Phi[A]$, as follows:*

  $$\Phi^\natural[A,C,f,z] =_{df} [x_1 : K_1]...[x_m][K_m]f(z(x_1,...,x_m))$$

  . When $m = 0$, we simply have $\Phi^\natural[A,C,f,z] \equiv f(z)$.

With the above notations, we can now introduce the inductive data types in UTT as follows. Let Γ be a valid context, and $\bar{\Theta} = \langle \Theta_1, ..., \Theta_n \rangle$ ($n \in \omega$) a sequence of inductive schemata in Γ. Then, $\bar{\Theta}$ generates a Γ-type which is introduced by declaring the following constant expressions w.r.t. $\bar{\Theta}$:

$$\begin{aligned}\mathcal{M}[\bar{\Theta}] \;&:\; \mathbf{Type} \\ \iota[\bar{\Theta}] \;&:\; \Theta_i[\mathcal{M}[\bar{\Theta}]] \\ \mathbf{E}[\bar{\Theta}] \;&:\; (C : (\mathcal{M}[\bar{\Theta}])\mathbf{Type}) \\ & \quad\; f_1 : \Theta_1^\circ[\mathcal{M}[\bar{\Theta}], C, \iota_1[\bar{\Theta}]])...(f_n : \Theta_n^\circ[\mathcal{M}[\bar{\Theta}], C, \iota_n[\bar{\Theta}]]) \\ & \quad\; (z : \mathcal{M}[\bar{\Theta}])C(z)\end{aligned}$$

and asserting the following n computation rules for $i = 1, ..., n$:

$$\begin{aligned} & \mathbf{E}[\bar{\Theta}](C, \bar{f}, \iota[\bar{\Theta}](\bar{x})) \\ = \;\; & f_i(\bar{x}, \Phi^\natural_{i_1}[\mathcal{M}[\bar{\Theta}], C, \mathbf{E}[\bar{\Theta}](C,\bar{f}), x_{i_1}], ..., \Phi^\natural_{i_k}[\mathcal{M}[\bar{\Theta}], C, \mathbf{E}[\bar{\Theta}](C,\bar{f}), x_{i_k}]) \\ : \;\; & C(\iota_i[\bar{\Theta}](\bar{x}) \end{aligned}$$

where it is assumed that Φ_i be of form $(x_1 : M_1)...(x_{m_i} : M_{m_i})X$, $\langle \Phi_{i_1}, ..., \Phi_{i_k} \rangle$ be the subsequence of $\langle M_1, ..., M_{m_i} \rangle$ that consists of the strictly positive operators, and $\bar{f}$ stand for $f_1, ..., f_n$ and $\bar{x}$ for $x_1, ..., x_{m_i}$

Example 2.8. *Inductive types can also be introduced directly in LF and this could be equivalent as introduced by schemata as the following examples:*

1. *We can define natural numbers:*

$$N =_{df} \mathcal{M}[X, (X)X]$$

The direct declarations of constants in LF can be formed in Example 2.3.

2. *We can define the types of lists*

$$List =_{df} [A : \mathbf{Type}]\mathcal{M}[X, (A)(X)X]$$

and specify it in LF as:

$$\begin{array}{rcl} List & : & (\mathbf{Type})\mathbf{Type} \\ nil & : & (A : \mathbf{Type})List(A) \\ cons & : & (A : \mathbf{Type})(a : A)(l : List(A))List(A) \\ \mathbf{E}_{List} & : & (A : \mathbf{Type})(C : (List(A))\mathbf{Type})(c : C(nil(A))) \\ & & (f : (a : A)(l : List(A))(C(l))C(cons(A, a, l))) \\ & & (z : List(A))C(z) \end{array}$$

$$\mathbf{E}_{List}(A, C, c, f, nil(A)) = c : C(nil(A))$$
$$\mathbf{E}_{List}(A, C, c, f, cons(A, a, l)) = f(a, l, \mathbf{E}_{List}(A, C, c, f, l)) : C(cons(A, a, l))$$

2.2.3 Predicative Universes

In UTT, besides the impredicative universe *Prop*, we also introduce the predicative universes $Type_i$ $(i \in \omega)$. For any object a in $Type_i$, $T_i(a)$ is the type named by a:

$$Type_i : \mathbf{Type}, \qquad T_i : (Type_i)\mathbf{Type}$$

Each predicative universe $Type_i$ has a name $types_i$ in $Type_{i+1}$:

$$type_i : Type_{i+1}, \qquad T_{i+1}(type_i) = Type_i : \mathbf{Type}$$

The impredicative universe of propositions has a name *prop* in $Type_0$:

$$prop : Type_0, \qquad T_0(prop) = Prop : \mathbf{Type}$$

Every type with a name $Type_i$ has a name in $Type_{i+1}$:

$$t_{i+1} : (Type_i)Type_{i+1}, \qquad T_{i+1}(t_{i+1}(a)) = T_i(a) : \mathbf{Type}$$

The proof type of any proposition in *Prop* has a name in $Type_0$:

$$t_0 : (Prop)Type_0, \qquad T_0(t_0(P)) = \mathbf{Prf}(P) : \mathbf{Type}$$

Finally, the inductive types generated by the inductive schemata have names in the appropriate predicative universes, whose introduction conforms with the predicativity of $Types_i$(see §9.2.3 of [Luo94] for formal details).

Remark 2.9. *Martin-Löf's type theory [NPS90] is formally a subsystem of UTT: the former can be obtained by removing the impredicative universe Prop from UTT. So, although in this thesis we prove conservativity of coercive subtyping for UTT, it also holds for Martin-Löf's type theory.*

3. COERCIVE SUBTYPING – THE IDEA, ORIGINAL DESCRIPTION AND PROBLEMS

Subtyping is an interesting issue when we consider type theory. There are different ways of introducing subtypes into type theory. Coercive subtyping, which considers subtyping as an abbreviation mechanism, is a simple but powerful approach. In this chapter, we will first talk about its basic idea, then give a formal description of coercive subtyping and discuss two important properties of the system with coercive subtyping: coherence and conservativity. Further more, we will present a counter example to show that the original formulation of coercive subtyping is problematic and propose the solution for the problem. The detail of the solution will be given in the next chapter.

3.1 The Idea of Coercive Subtyping

Coercive subtyping [Luo97, Luo99] is an approach to introducing subtyping into type theory and it considers subtyping by means of *abbreviations*.

The basic idea is that, when we consider A as a subtype of B, we choose a unique function c from A to B, and declare c to be a *coercion*, written as $A <_c B$. Intuitively, the idea means that, anywhere we need to write an object of type B, we can write an object a of type A instead, and in this context, the object a is to be seen as an *abbreviation* for the object $c(a) : B$. More precisely, if f is a function from B to C, then f can be applied to any object a of type A to form $f(a)$ of type C, which is definitionally equal to $f(c(a))$. We can consider $f(a)$ to be an abbreviation for $f(c(a))$, with coercion c being inserted to fill the *gap* between f and a. The idea above could be captured by means of the following formal rules:

$$\frac{f : (x : B)C \quad a : A \quad A <_c B}{f(a) : [c(a)/x]C}$$

$$\frac{f : (x : B)C \quad a : A \quad A <_c B}{f(a) = f(c(a)) : [c(a)/x]C}$$

It is a simple but really powerful mechanism with many uses. One of the interesting uses is in type-theoretical semantics in linguistic interpretation [Luo10]. We use the following example to show how it works here.

Example 3.1. *Suppose we want to interpret 'John runs', and we have interpretations of both 'John' and 'run':*

$$[[run]] : Human \to Prop$$

$$[[John]] : Man$$

We cannot apply $[[run]]$ *to* $[[John]]$ *yet. If we define a coercion* c *from* Man *to* $Human$*,*

$$Man <_c Human$$

which means Man *is a subtype of* $Human$*, we could naturally consider* $[[John]]$ *as an object of* $Human$ *via* c*, hence 'John runs' could be interpreted as:*

$$[[runs]]([[John]]) = [[runs]](c([[John]])) : Prop$$

3.2 $T[\mathcal{R}]$ – the original description of coercive subtyping

In this section, we will give a formal definition of coercive subtyping introduced in [Luo99].

Consider a type theory T (UTT or Martin-löf's type theory) which is a type theory specified in LF. Let $\mathcal{R}$ to be a set of basic subtyping rules, the extension of T with coercive subtyping is defined by the following steps.

System $T[\mathcal{R}]_0$ $T[\mathcal{R}]_0$ is an extension of T with the subtyping judgement of form $\Gamma \vdash A <_c B : \mathbf{Type}$ by using a set $\mathcal{R}$ of basic subtyping rules whose conclusions are subtyping judgements and the following rules in Figure 3.1:

Congruence

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash A = A' : \mathbf{Type} \quad \Gamma \vdash B = B' : \mathbf{Type} \quad \Gamma \vdash c = c' : (A)B}{\Gamma \vdash A' <_{c'} B' : \mathbf{Type}}$$

Transitivity

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type} \qquad \Gamma \vdash B <_{c'} C : \mathbf{Type}}{\Gamma \vdash A <_{c' \circ c} C : \mathbf{Type}}$$

Substitution

$$\frac{\Gamma, x : K, \Gamma' \vdash A <_c B : \mathbf{Type} \qquad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]A <_{[k/x]c} [k/x]B : \mathbf{Type}}$$

Fig. 3.1: The general subtyping rules

Definition 3.2. *(coherence condition)We say that the basic subtyping rules are coherent if $T[\mathcal{R}]_0$ has the following coherence properties:*

1. *If* $\Gamma \vdash A <_c B : \mathbf{Type}$, *then* $\Gamma \vdash A : \mathbf{Type}$, $\Gamma \vdash B : \mathbf{Type}$, *and* $\Gamma \vdash c : (A)B$.
2. $\Gamma \not\vdash A <_c A : \mathbf{Type}$ *for any* Γ, A *and* c.
3. *If* $\Gamma \vdash A <_c B : \mathbf{Type}$ *and* $\Gamma \vdash A <_{c'} B : \mathbf{Type}$, *then* $\Gamma \vdash c = c' : (A)B$.

System $T[\mathcal{R}]$ Let $\mathcal{R}$ be a set of basic subtyping rules. System T[$\mathcal{R}$] is the system obtained from T[$\mathcal{R}$]$_0$ by adding the new subkinding judgement form $\Gamma \vdash K <_c K'$ and the rules in Figure 3.2.

New rules for application

$$(CA1)\frac{\Gamma \vdash f : (x:K)K' \quad \Gamma \vdash k_0 : K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) : [c(k_0)/x]K'}$$

$$(CA2)\frac{\Gamma \vdash f = f' : (x:K)K' \quad \Gamma \vdash k_0 = k_0' : K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) = f'(k_0') : [c(k_0)/x]K'}$$

Coercive definition rule

$$(CD)\frac{\Gamma \vdash f : (x:K)K' \quad \Gamma \vdash k_0 : K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$$

Basic subkinding rule

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type}}{\Gamma \vdash El(A) <_c El(B)}$$

Subkinding for dependent product kinds

$$\frac{\Gamma \vdash K_1' <_{c_1} K_1 \;\; \Gamma, x' : K_1' \vdash [c_1(x')/x]K_2 = K_2' \;\; \Gamma, x : K_1 \vdash K_2 : \mathbf{kind}}{\Gamma \vdash (x:K_1)K_2 <_c (x':K_1')K_2'}$$

where $c \equiv [f:(x:K_1)K_2][x':K_1']f(c_1(x'))$

$$\frac{\Gamma \vdash K_1 = K_1 \;\; \Gamma, x' : K_1' \vdash K_2 <_{c_2} K_2' \;\; \Gamma, x : K_1 \vdash K_2 : \mathbf{kind}}{\Gamma \vdash (x:K_1)K_2 <_c (x':K_1')K_2'}$$

where $c \equiv [f:(x:K_1)K_2][x':K_1']c_2 f(x')$

$$\frac{\Gamma \vdash K_1' <_{c_1} K_1 \;\; \Gamma, x' : K_1' \vdash [c_1(x')/x]K_2 <_{c_2} K_2' \;\; \Gamma, x : K_1 \vdash K_2 : \mathbf{kind}}{\Gamma \vdash (x:K_1)K_2 <_c (x':K_1')K_2'}$$

where $c \equiv [f:(x:K_1)K_2][x':K_1']c_2 f(c_1(x'))$

Congruence for subkinding

$$\frac{\Gamma \vdash K_1 <_c K_2 \;\; \Gamma \vdash K_1 = K_1' \;\; \Gamma \vdash K_2 = K_2' \;\; \Gamma \vdash c = c' : (K_1)K_2}{\Gamma \vdash K_1' <_c K_2'}$$

Transitivity for subkinding

$$\frac{\Gamma \vdash K <_c K' \;\; \Gamma \vdash K' <_c' K'}{\Gamma \vdash K <_{c' \circ c} K''}$$

Substitution for subkinding

$$\frac{\Gamma, x:K, \Gamma' \vdash K_1 <_c K_2 \;\; \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K_1 <_{[k/x]c} [k/x]K_2}$$

Fig. 3.2: The subkinding rules of $T[\mathcal{R}]$.

Remark 3.3. *There're several other early versions of coercive subtyping. The formulation in [Luo97] used a set of pairs of types for the basic coercions. In [LL05], a set of coercion judgement called WDC (well-defined coercion, satisfies congruence, transitivity, substitution and coherence) is used as the basic coercion set. These two versions are not bothered by the problem we will show in the subsection 3.4.2, because they are not based on the open set of coercion rules. However the formulation in [Luo97] is a very early version not formulated well enough, and the conditions of WDC in [LL05] were too strict, apart from coherence, it also requires to be a closure of congruence, transitivity and substitution.*

3.3 Judgements

As we have introduced in chapter 2, a type theory specified in LF has five forms of judgement.

$$\Gamma \vdash \mathbf{valid} \quad \Gamma \vdash K \ \mathbf{kind} \quad \Gamma \vdash k : K \quad \Gamma \vdash K_1 = K_2 \quad \Gamma \vdash k_1 = k_2 : K$$

In the type theory extended with coercive subtyping, we have two more forms of judgement for subtyping and subkinding.

$$\Gamma \vdash A <_c B : \mathbf{Type} \qquad \Gamma \vdash K_1 <_c K_2$$

3.3.1 Presupposed Judgements

In a type system, the well-formedness of a judgement is not given syntactically but governed by judgemental derivability. We can write a judgement which is syntactically fine but not well-formed, hence not derivable in the given type system.

When we say $\Gamma \equiv x_1 : K_1, ..., x_n : K_n$ is a valid context, it actually requires that $K_1, \ldots K_n$ are meaningful kind. When we talk about a judgement, like $\Gamma \vdash k : K$, we should assume that Γ is a valid context, and K is a kind. These pre-assumptions, we call them *presuppositions.* Informally, in LF, every judgement has several presuppositions, which we call *presupposed judgements* . For example, if we have $\Gamma \vdash K_1 = K_2$, we should have that Γ is a valid context and K_1, K_2 are both kinds. If we say $\Gamma \vdash K_1 <_c K_2$, we should have that K_1, K_2 are kinds, c is of type $(K_1)K_2$ and so on. Formally, since a judgement has different forms, we have different presuppositions accordingly.

Definition 3.4. *The notion of presupposed judgements is inductively defined as follows:*

1. $\Gamma_1 \vdash$ **valid** *is presupposed judgement of* $\Gamma_1, \Gamma_2 \vdash J$.
2. $\Gamma_1 \vdash K$ **kind** *is presupposed judgement of* $\Gamma_1, x\colon K, \Gamma_2 \vdash J$.
3. $\Gamma, x\colon K_1 \vdash K_2$ **kind** *is presupposed judgement of* $\Gamma \vdash (x\colon K_1)K_2$ **kind**.
4. $\Gamma \vdash K_1$ **kind** *and* $\Gamma \vdash K_2$ **kind** *are presupposed judgements of* $\Gamma \vdash K_1 = K_2$.
5. $\Gamma \vdash K$ **kind** *is presupposed judgement of* $\Gamma \vdash E\colon K$ *(E denotes a term or term equality here).*
6. $\Gamma \vdash k_1\colon K$ *and* $\Gamma \vdash k_2\colon K$ *are presupposed judgements of* $\Gamma \vdash k_1 = k_2\colon K$.
7. $\Gamma \vdash A\colon$ **Type**, $\Gamma \vdash B\colon$ **Type** *and* $\Gamma \vdash c\colon (El(A))El(B)$ *are presupposed judgements of* $\Gamma \vdash A <_c B$.
8. $\Gamma \vdash K$ **kind**, $\Gamma \vdash K'$ **kind** *and* $\Gamma \vdash c\colon (K)K'$ *are presupposed judgements of* $\Gamma \vdash K <_c K'$.

3.3.2 Equality between Judgements

Now we will introduce some definitions and notations for the equivalence of the contexts and judgements.

Notation 3.5. *In a type system S, let Γ_1 and Γ_2 be*

$$\Gamma_1 \equiv x_1 : K_1, x_2 : K_2, \cdots, x_n : K_n$$

$$\Gamma_2 \equiv x_1 : M_1, x_2 : M_2, \cdots, x_n : M_n$$

The equality $\Gamma \vdash \Gamma_1 = \Gamma_2$ is an abbreviation for the following list of n judgements:

$$\begin{array}{rcl} \Gamma & \vdash & K_1 = M_1; \\ \Gamma,\ x_1 : K_1 & \vdash & K_2 = M_2; \\ & \cdots & \\ \Gamma, x_1 : K_1, \cdots, x_{n-1} : K_{n-1} & \vdash & K_n = M_n. \end{array}$$

Definition 3.6. *(equality between judgements). Let S be a type theory. The notion of equality between judgements of the same form in S, notation $J_1 = J_2$ (with S omitted), is inductively defined as follows:*

1. $(\Gamma_1 \vdash \mathbf{valid}) = (\Gamma_2 \vdash \mathbf{valid})$ *iff* $\vdash \Gamma_1 = \Gamma_2$ *is derivable in S.*
2. $(\Gamma_1 \vdash K_1\ \mathbf{kind}) = (\Gamma_2 \vdash K_2\ \mathbf{kind})$ *iff* $\vdash \Gamma_1 = \Gamma_2$ *and* $\Gamma_1 \vdash K_1 = K_2$ *are derivable in S.*
3. $(\Gamma_1 \vdash k_1 : K_1) = (\Gamma_2 \vdash k_2 : K_2)$ *iff* $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash K_1 = K_2$ *and* $\Gamma_1 \vdash k_1 = k_2 : K_1$ *are derivable in S.*
4. $(\Gamma_1 \vdash K_1 = K_1') = (\Gamma_2 \vdash K_2 = K_2')$ *iff* $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash K_1 = K_2$ *and* $\Gamma_1 \vdash K_1' = K_2'$ *are derivable in S.*
5. $(\Gamma_1 \vdash k_1 = k_1' : K_1) = (\Gamma_2 \vdash k_2 = k_2' : K_2)$ *iff* $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash K_1 = K_2$, $\Gamma_1 \vdash k_1 = k_2 : K_1$ *and* $\Gamma_1 \vdash k_1' = k_2' : K_1$ *are derivable in S.*
6. $(\Gamma_1 \vdash A_1 <_{c_1} B_1 : \mathbf{Type}) = (\Gamma_2 \vdash A_2 <_{c_2} B_2 : \mathbf{Type})$ *iff* $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash A_1 = A_2 : \mathbf{Type}$, $\Gamma_1 \vdash B_1 = B_2 : \mathbf{Type}$, $\Gamma_1 \vdash c_1 = c_2 : (A_1)B_1$ *are derivable in S.*

7. $(\Gamma_1 \vdash K_1 <_{c_1} K_1') = (\Gamma_2 \vdash K_2 <_{c_2} K_2')$ *iff* $\vdash \Gamma_1 = \Gamma_2$, $\Gamma_1 \vdash K_1 = K_2$, $\Gamma_1 \vdash K_1' = K_2'$ *and* $\Gamma_1 \vdash c_1 = c_2 : (K_1)K_1'$ *are derivable in* S*.*

Definition 3.7. *Suppose d is a derivation in type system S, let $conc(d)$ denote the conclusion of derivation d. Given two derivations d_1 and d_2, we write $d_1 \sim d_2$ iff $conc(d_1) = conc(d_2)$ in S.*

Proposition 3.8. *Relation $\sim$ is an equivalence relation.*

Proof. Proved trivially by the definition. □

Notation 3.9. *Sometimes, we will notion $=_S$ and $\sim_S$, for the relation $=$ and $\sim$ in type system S to avoid confusion.*

3.4 Conservative Extension

When we extend a system into a new one, the relations between the two systems interest us. As an extension of a type theory, coercive subtyping is based on the idea that subtyping is abbreviation. On one hand, it should not increase any expressive power of the original system. On the other hand, coercions should always be correctly inserted to obtain the abbreviated expressions as long as the basic coercions are coherent.

Generally, when we say system T_2 is an *extension* of system T_1, it means that for any sequent t of system T_1, if t is derivable in T_1, then t is derivable in T_2. When we say T_2 is a *conservative extension* of system T_1 , we need further require that, for any sequent t of the system T_1, if t is not derivable in T_1 then t is not derivable in its extension T_2. Put in another way, if t is derivable in T_2, t is derivable in T_1 as well. If a sequent does not belong to T_1 (belongs only to T_2), its derivability does not matter.

More precisely, if we use $\vdash_T$ for the judgement derivable in system T, for any judgement $\Gamma \vdash J$ in T_1(it may not be derivable), T_2 is an extension of T_1 requires that:

$$\Gamma \vdash_{T_1} J \quad \Rightarrow \quad \Gamma \vdash_{T_2} J$$

For such an extension to be conservative, we require:

$$\Gamma \vdash_{T_2} J \quad \Rightarrow \quad \Gamma \vdash_{T_1} J$$

We give the following formal definition of conservative extension.

Definition 3.10. ***(conservative extension)*** *We call system T_2 a conservative extension of T_1, if for any judgement J in T_1, there's a derivation of J in T_1 iff there's a derivation of J in T_2.*

In the previous treatments of coercive subtyping, the notion of conservative extension as considered in [SL02] was not explicitly linked to that in the traditional definition and, as a consequence, it was not as well understood as it should have been (see subsection 3.4.2 for details). In this section we will give some discussion and propose some adjustment of the system, to show in what sense coercive subtyping is a conservative extension.

3.4.1 Coherence

When we consider the conservativity in coercive subtyping, *coherence* is an important property that is required (recall the definition 3.2, for the coherence condition in $T[\mathcal{R}]_0$). We should point out that this uniqueness is necessary.

Actually, in system $T[\mathcal{R}]$ we can prove that any two different coercions between the same two kinds are equal. More precisely, given $\Gamma \vdash K <_c K'$ and $\Gamma \vdash K <_{c'} K'$, we can use the coercive definition rule and $\beta\eta\xi$ rules (in Figure 2.1) to derive $\Gamma \vdash c = c' : (K)K'$

Proposition 3.11. *If $\Gamma \vdash K <_c K'$ and $\Gamma \vdash K <_{c'} K'$ are derivable judgments in $T[\mathcal{R}]$, we have $\Gamma \vdash c = c' : (K)K'$*

Proof.

$$\begin{aligned}
\Gamma \vdash c &= [x:K](c(x)) \qquad (\eta \text{ rule}) \\
&= [x:K]([y:K']y)c(x) \qquad (\beta \text{ rule}) \\
&= [x:K]([y:K']y)(x) \qquad (\xi \text{ and coercive definition}) \\
&= [x:K]([y:K']y)(c'(x)) \qquad (\xi \text{ and coercive definition}) \\
&= [x:K](c'(x)) \qquad (\beta \text{ rule}) \\
&= c' : (K)K' \qquad (\eta \text{ rule}) \qquad \square
\end{aligned}$$

Suppose the coherence is not held in $T[\mathcal{R}]$, we have two different coercions c_1 and c_2 from type A to type B,

$$\Gamma \vdash A <_{c_1} B : Type, \qquad \Gamma \vdash A <_{c_2} B : Type$$

where $\Gamma \not\vdash c_1 = c_2 : (El(A))El(B)$ in T.

So we have two coercions between kind $El(A)$ and $El(B)$

$$\Gamma \vdash El(A) <_{c_1} El(B), \qquad \Gamma \vdash El(A) <_{c_2} El(B)$$

Using the conclusion of the proof above, we can get $\Gamma \vdash c_1 = c_2 : (El(A))El(B)$ in $T[\mathcal{R}]$, but this is not derivable in T. This means that without the coherence condition, $T[\mathcal{R}]$ is not a conservative extension of T. So it gives us the answer that coherence is necessary for conservativity. And this coherence condition should be given in system $T[\mathcal{R}]_0$ before we introduce the coercive application rules.

3.4.2 A Problem of Original Formulation

We wish $T[\mathcal{R}]$ to be a conservative extension over T. Unfortunately, this formulation of coercive subtyping is too general for the result to hold, the conservativity cannot be satisfied.

We can think of the problem intuitively like this [1]: there might be some *bad rules* in $\mathcal{R}$, they will never be used in $T[\mathcal{R}]_0$ but can be applied in $T[\mathcal{R}]$. Such kind of naughty rules will bring us troubles.

More precisely, the key point of the problem is that we can use the judgements generated by coercive application as premises of the rules in $\mathcal{R}$. Some rules may have premises which contain 'gaps' and are not well-typed in T. These rules cannot be applied in $T[\mathcal{R}]_0$, since there's no coercive application rule in $T[\mathcal{R}]_0$. But in $T[\mathcal{R}]$ the coercive application helps them to insert the 'gaps', making it possible for them to be applied in $T[\mathcal{R}]$. These rules might generate some new coercive subtyping judgements which violate conservativity.

For instance, we can consider the following example.

Example 3.12. *Suppose Nat : **Type**, $Bool$: **Type**, $c_1 : (Nat)Bool$, $c_2 : (Nat)Bool$, and $\Gamma \not\vdash c_1 = c_2 : (Nat)Bool$ in T. Let the set $\mathcal{R}$ consist of the following two rules:*

$$\frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash Nat <_{c_1} Bool : \textbf{Type}} \quad (*)$$

$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash g : (Bool)Bool \quad \Gamma \vdash g(n) : Bool}{\Gamma \vdash Nat <_{c_2} Bool : \textbf{Type}} \quad (**)$$

Then, $\mathcal{R}$ is coherent. The reason is that, since we have no coercive application in $T[\mathcal{R}]_0$, if $g : (Bool)Bool$ and $n : Nat$, then $g(n)$ will not be well-typed. Thus, it is impossible to use any instance of (**) in $T[\mathcal{R}]_0$, so c_1 is the only coercion from Nat to $Bool$ in $T[\mathcal{R}]$.

But in system $T[\mathcal{R}]$, after introducing the coercive application rule, we have:

$$\frac{\Gamma \vdash g : (Bool)Bool \quad \Gamma \vdash n : Nat \quad \dfrac{\Gamma \vdash Nat <_{c_1} Bool : \textbf{Type}}{\Gamma \vdash Nat <_{c_1} Bool}}{\Gamma \vdash g(n) : Bool}$$

[1] The problem was realized by Zhaohui Luo and Sergei Solviev when discussing a question raised by Robin Adams in 2007.

Now, rule (**) can be applied (with $g \equiv id_b$ *and* $n \equiv zero$*), and we get two coercions from* Nat *to* $Bool$*:*

$$\Gamma \vdash Nat <_{c_1} Bool : \mathbf{Type}, \qquad \Gamma \vdash Nat <_{c_2} Bool : \mathbf{Type} \ .$$

By the argument in the previous section , we have $\Gamma \vdash c_1 = c_2 : (Nat)Bool$ *in* $T[\mathcal{R}]$*. It means that this system* $T[\mathcal{R}]$ *with a coherence set of rules* $\mathcal{R}$ *is not a conservative extension.*

More seriously, if we take $c_1 \equiv [x : Nat]true$ *and* $c_2 \equiv [x : Nat]false$ *as the coercion functions, then we can see that it is even possible that for a consistent* T *and a coherent* $\mathcal{R}$*,* $T[\mathcal{R}]$ *is inconsistent, since in* $T[\mathcal{R}]$ *we could have:*

$$true \ = \ c_1(zero) \ = \ c_2(zero) = \ false.$$

This example shows that the formulation of coercive subtyping in [Luo99] is not correct, and also tells us the importance of considering conservativity: non-conservativity might lead to inconsistency in some cases.

To solve the problem, it suffices to realize that, if we consider only subtyping *judgements*, rather than *rules*, the problem as demonstrated in Example 3.12 does not occur anymore. In other words, we still follow the formulation above to extend the coercive subtyping, however, instead of using a set of rules $\mathcal{R}$, we use a set $\mathcal{C}$ of subtyping judgements to form the coercive subtyping extension $T[\mathcal{C}]$. In such a formulation, the above problem does not occur. The formal description will be given in the next chapter.

Theoretically, this is a very general formulation. In fact, we can recast the formulation $T[\mathcal{R}]$ by means of a set $\mathcal{R}$ of basic subtyping rules as the system $T[\mathcal{C}_{\mathcal{R}}]$, where $\mathcal{C}_{\mathcal{R}}$ is the set of subtyping judgements generated by the $\mathcal{R}$-rules without the coercive application/definition rules $(CA1)$, $(CA2)$ and (CD). In this way, we can also encompass the generality and flexibility offered by subtyping rules.

3.4.3 An Intermediate System with '$*$'

When we move into system $T[\mathcal{C}]$, we still find some difficulties. As we mentioned above, in type theories, the validity or well-formedness of expressions is not given syntactically, but governed by judgemental derivability. So, if we consider a type theory T and its extension T', we can possibly have an expression of T that is not well-typed in T but is well-typed in T'. In this case, we cannot say T' is a conservative extension of T, at least in the traditional sense. Unluckily, when we consider the extension with coercive subtyping, this problem exists.

As we have claimed, coercive subtyping is an abbreviation mechanism, the judgements in a system with coercive subtyping in general contain "gaps" where coercions may be inserted. These "gaps" are not marked in the syntax. There could be a judgement not well-formed in the original system but well-formed in the extended system with coercive subtyping, not derivable in the original system but derivable in the extended system, and, the judgement is still a sequent of the original system. More precisely, for instance, in $T[\mathcal{C}]$, consider the coercive application:

$$\frac{\Gamma \vdash f : (x:K)K' \quad \Gamma \vdash k_0 : K_0 \quad \Gamma \vdash K <_c K_0}{\Gamma \vdash f(k_0) : [c(k_0)/x]K'}$$

As a sequent of T, $\Gamma \vdash f(k_0) : [c(k_0)/x]K'$ is not derivable in T, but is derivable in $T[\mathcal{C}]$. This does not satisfy the traditional notion of conservative extension.

To solve this problem, we propose a simple idea: introducing a special symbol '$*$' for coercive application, where '$*$' is used to mark the place of coercion insertion. We call it *star-calculus.*

$$\frac{\Gamma \vdash f : (x:K)K' \quad \Gamma \vdash k_0 : K_0 \quad \Gamma \vdash K <_c K_0}{\Gamma \vdash f * k_0 : [c(k_0)/x]K'}$$

We notate the new system $T[\mathcal{C}]^*$. Now, $\Gamma \vdash f * k_0 : [c(k_0)/x]K'$ is not a sequent of T any more. So although it is not derivable in T but derivable in $T[\mathcal{C}]^*$, it does not violate the requirement for the conservative extension.

However, this does not mean that we need to abandon the coercive subtyping system $T[\mathcal{C}]$ and change into the new one $T[\mathcal{C}]^*$. Although the notation '$*$' makes the star-calculus a more natural conservative extension, there are serious reasons to keep the old description without $*$ in favor of this approach. $T[\mathcal{C}]$ itself is directly connected to important themes in the study of subtyping:

- Implicit coercions;
- Subtyping as abbreviation.

These two are related and supported by the possible "omission" of coercions in $T[\mathcal{C}]$: when $A <_c B, a\colon A$ can be used as an object of type B. So, the same term $f(a)$ becomes well-typed although it is not well-typed before.

In the next Chapter, we will give a formal definition of the system $T[\mathcal{C}]^*$, prove that $T[\mathcal{C}]^*$ is a conservative extension of T in the traditional sense, and show that $T[\mathcal{C}]^*$ is equivalent to $T[\mathcal{C}]$. Hence $T[\mathcal{C}]$ is a conservative extension of T in a certain sense.

3.5 Definitional Extension

Conservativity is not enough to capture the relation between a type theory and its coercive subtyping extension. For our extension with coercive subtyping, apart from the conservative property, we also want to show that every derivation in the system with coercive subtyping corresponds to a derivation in the type system without coercion where all the "gaps" caused by the coercions are inserted. We find that this property is very similar to that of *definitional extension.*

Traditionally, the notion of *definitional extension* was formulated for first-order logical theories: a first-order theory is a definitional extension of another if the former is a conservative extension of the latter and any formula in the former is logically equivalent to its translation in the latter (see, for example [Kle52]). More precisely, a definitional extension S' of a first-order theory S is obtained by successive introductions of relation symbols (or function symbols) in such a way that, for example, for an n-ary relation symbol R, the following *defining axiom* of R is added:

$$\forall x_1 ... \forall x_n.\ R(x_1, ..., x_n) \iff \phi(x_1, ..., x_n),$$

where $\phi(x_1, ..., x_n)$ is a formula in S. For such a definitional extension S', we have

1. For any formula ψ in S', $\psi \iff \psi^*$ is provable in S', where ψ^* is the formula in S obtained from ψ by replacing any occurrence of $R(t_1, ..., t_n)$ by $\phi(t_1, ..., t_n)$ (with necessary changes of bound variables).
2. S' is a conservative extension of S.

In particular, the defining axiom of R cannot be used to prove new theorems expressible without R.

In our type theory extended with coercive subtyping, as we mentioned above, the validity and well-formedness of a formula (or judgement) is linked to its derivation. So in order to show the definitional extension property in a more general sense, we will transform derivations in $T[\mathcal{C}]$ or $T[\mathcal{C}]^*$ with coercions into derivations in T without coercions, and show that the derivation and its transformation are "equivalent".

But there's still difficulty in the notion. If we want $T[\mathcal{C}]^*$ to be a definitional extension of T, we should consider the derivation of judgement $\Gamma \vdash K <_c K'$ in $T[\mathcal{C}]^*$ and find an equivalent derivation for it in T. However we can hardly do this. To solve this problem, we introduce another intermediate system $T[\mathcal{C}]_{0K}$. $T[\mathcal{C}]_{0K}$ is a system obtained by extending T with judgements of subtyping and subkinding form and the inference rules for subtyping and subkinding except the coercive application and definition rules.

It is obvious that $T[\mathcal{C}]_{0K}$ is a conservative extension of T , since the subkinding judgements do not contribute to the derivation of a judgement of any other form.

In next chapter, we will give a formal proof to show that if $\mathcal{C}$ is a coherent set of judgements, $T[\mathcal{C}]^*$ is actually a definitional extension of $T[\mathcal{C}]_{0K}$ in the following sense.

- $T[\mathcal{C}]^*$ is a conservative extension over $T[\mathcal{C}]_{0K}$;
- every $T[\mathcal{C}]^*$-derivation can be transformed into an "equivalent" $T[\mathcal{C}]_{0K}$-derivation

3.6 Coercive Subtyping and Subsumptive Subtyping

Before going to the next chapter for a formulation of coercive subtyping and the proof of its adequacy, we would like to discuss something more about coercive subtyping, to show that this mechanism is useful, powerful and worth studying.

There are several possible ways of introducing a notion of *subtyping* into a type theory. The traditional approaches use a rule called *subsumption*: when a type A is a subtype of type B, the objects of A could be considered as objects of B as well. Such a subtyping mechanism is called *subsumptive subtyping*. Formally, the subsumption rule is:

$$\frac{a : A \quad A < B}{a : B}$$

Traditionally, there are two different views of type theory. One is *type assignment*, which is often found in the study of programming languages [Mil78], when types are assigned to already defined terms. Another is the view that takes so-called *canonical objects* seriously, like in Martin-Löf's type theory or UTT. Under this view, types are considered consisting of its all canonical objects and, furthermore, objects and their types depend on each other and cannot be thought of to exist independently. For example, the type Nat for natural numbers could be defined as $Nat = 0|succ(Nat)$, Nat consists of canonical objects 0 and $succ(n)$, and numbers exist only because they are objects of Nat.

Corresponding to these two different views of types, we have two related but different ways of considering subtyping. For a type assignment system, subsumption is a natural rule to consider for subtyping. However, in type theories with canonical objects, coercive subtyping is a better choice.

Comparing subsumptive subtyping and coercive subtyping, we will find that, in subsumptive subtyping, an object can have more than one type through the subsumption rule; this is natural with the type assignment way of thinking. But in coercive subtyping, objects do not obtain more types (unlike subtyping with subsumption). If a is an object of type A and $A <_c B$, then a is *not* an object of type B. The coercion will be applied only when a function $f : (B)C$ is applied to a, as the example below explains. This is more natural for type theories with canonical objects, since objects and types depend on each other.

If we introduce the subsumption rule into type theories with canonical objects, it causes problems. We often wish to consider a type A to be subtype of a type B(or simply thinking type A to be the type B) , even it is not true that every canonical object of type A is (or reduces to) a canonical object of type B. In this situation, we might have questions: suppose a is a canonical object of A, it should be an object of B by the subsumption rule, but is a a canonical object of B or can it be reduced to a canonical object of B as well? Type theory with canonical objects are based on the induction rules, how would induction principles be formulated to take care of the objects introduced by subtyping?

Such considerations lead to difficulties: subsumptive subtyping is incompatible with the idea of canonical object in the sense that it cannot be employed for a type theory with canonical objects without destroying canonicity.

Since coercive subtyping is just an abbreviation mechanism, it avoids such problems. Here, we consider an example of subtyping to explain the above point that coercive subtyping is more adequate for type theories with canonical objects.

Example 3.13. *Consider $List(M)$, the type of lists of objects of type M, which has constructors $nil(M)$ and $cons_M$. The following is a natural rule for subtyping:*

$$\frac{A < B}{List(A) < List(B)}$$

Now $List(A)$ consists of canonical objects $nil(A)$ and $cons_A(a,l)$ and $List(B)$ consists of canonical objects $nil(B)$ and $cons_B(b,l)$. If we introduce subsumption rule, since $nil(A) : List(A)$, we should have

$$\frac{nil(A) : List(A) \quad List(A) < List(B)}{nil(A) : List(B)}$$

But $nil(A)$ is not equal to any canonical object of $List(B)$, $nil(B)$ or $cons_A(a,l)$.

With coercive subtyping, we could instead introduce the following rule

$$\frac{A <_c B}{List(A) <_{map(A,B,c)} List(B)}$$

where $map(A,B,c)$ is the function from $List(A)$ to $List(B)$ such that

$$map(A,B,c)(nil(A)) = nil(B)$$

$$map(A,B,c)(cons_A(a,l)) = cons_B(c(a), map(A,B,c)(l))$$

Now, if f requires an argument of type $List(B)$, we can write $f(nil(A))$ as an abbreviation for $f(map(A,B,c)(nil(A)))$, which reduces to $f(nil(B))$.

Remark 3.14. *We should mention another approach to subtyping with subsumption rule: the so-called "constructor subtyping" [BF99, BvR00], which introduces subsumption to a type theory with canonical objects. The idea is that an inductive type A is a subtype of another inductive type B, if B has more constructors than A. This requires overloading objects for different* *types and it is unclear how this would reconcile with the induction principles (or elimination rules) of inductive types. Constructive subtyping only works for some limited cases of inductive types and seems difficult to be generalized or applied to other cases.*

4. COERCIVE SUBTYPING AND PROOF

As we've explained above, there are two problems with the previous treatments of coercive subtyping. One is that the notion of 'basic subtyping rule' is too liberal that has failed to exclude the problematic rules and the other is that the notion of conservativity is not linked to the traditional notion of conservative extension. These problems lead to an incomplete proof and inaccurate description of 'conservativity' in [SL02] (recall the discussion in subsection 3.4.2).

To solve the first problem, we consider only subtyping *judgements*, rather than *rules*. In other words, we extend the original type theory with a set $\mathcal{C}$ of subtyping judgements to form the coercive subtyping extension $T[\mathcal{C}]$. To deal with the second problem, we shall introduce below, besides the coercive subtyping calculus $T[\mathcal{C}]$, the star-calculus $T[\mathcal{C}]^*$ that will help to make the notion of conservativity clear.

In such a formulation, the problems, as illustrated in Example 3.12 will not occur and the proof method of [SL02] can be used to prove that, based on a coherent set of coercion judgements, the coercive subtyping extension is indeed a conservative extension. Moreover, we will introduce an intermediate system $T[\mathcal{C}]_{0K}$ without coercive application and definition rules, and show that $T[\mathcal{C}]^*$ is actually a definitional extension of $T[\mathcal{C}]_{0K}$.

In this chapter, we will first present the formal formulation of the systems $T[\mathcal{C}]$, $T[\mathcal{C}]^*$, and the intermediate systems $T[\mathcal{C}]_0$ and $T[\mathcal{C}]_{0K}$. Then we will prove that:

- $T[\mathcal{C}]^*$ is a conservative extension of T.

- $T[\mathcal{C}]^*$ is equivalent to $T[\mathcal{C}]$.
- $T[\mathcal{C}]^*$ is a definitional extension of $T[\mathcal{C}]_{0K}$ which is a conservative extension over T.

4.1 Description of the Systems

In order to make our property proved, we will first make some changes to the logical framework LF. We add the weakening rule (1.4) and the context retyping rule (3.3) where J is of form $\Gamma \vdash$ **valid**, $\Gamma \vdash K$ **kind**, $\Gamma \vdash K_1 = K_2$, $\Gamma \vdash k : K$ or $\Gamma \vdash k_1 = k_2 : K$. However, these rules will be proved admissible later (see Lemma 4.12 and Lemma 4.19), which means the new LF is equivalent to the original one in Figure 2.1, so we will still call this extensional one LF. We will also introduce an intermediate system $T[\mathcal{C}]_{0K}$ which takes all the subkinding rules but without coercion application and definition rules.

The new inference rules for LF are followed in Figure 4.1.

Remark 4.1. *We note that the following inference rule is derivable*

$$\frac{\Gamma, \Gamma_1 \vdash J \quad \Gamma \vdash \Gamma_1 = \Gamma_2}{\Gamma, \Gamma_2 \vdash J}$$

Actually, it is by means of (call for Notation 3.5)

$$\dfrac{\dfrac{\dfrac{\Gamma, x_1 : K_1, x_2 : K_2, \cdots, x_n : K_n \vdash J \quad \Gamma \vdash K_1 = M_1}{\Gamma, x_1 : M_1, x_2 : K_2, \cdots, x_n : K_n \vdash J}}{\dfrac{\cdots\cdots\cdots}{\Gamma, x_1 : M_1, \cdots, x_{n-1} : M_{n-1}, x_n : K_n \vdash J}} \quad \Gamma, x_1 : M_1, \cdots, x_{n-1} : M_{n-1} \vdash K_n = M_n}{\Gamma, x_1 : M_1, x_2 : M_2, \cdots, x_n : M_n \vdash J}$$

4.1.1 System $T[\mathcal{C}]_0$

The system $T[\mathcal{C}]_0$ is the extension of system T with the new judgement form $\Gamma \vdash A <_c B : Type$ and the rules in Figure 4.2.

Contexts and assumptions

$$(1.1)\frac{}{<>\vdash \textbf{valid}} \qquad (1.2)\frac{\Gamma\vdash K\ \textbf{kind}\quad x\notin FV(\Gamma)}{\Gamma, x:K\ \vdash \textbf{valid}} \qquad (1.3)\frac{\Gamma, x:K,\Gamma'\vdash \textbf{valid}}{\Gamma, x:K,\Gamma'\vdash x:K}$$

$$(1.4)\frac{\Gamma,\Gamma'\vdash J\quad \Gamma\vdash K\ \textbf{kind}\quad x\notin FV(\Gamma)\cup FV(\Gamma')}{\Gamma, x:K,\Gamma'\vdash J}$$

General equality rules

$$(2.1)\frac{\Gamma\vdash K\ \textbf{kind}}{\Gamma\vdash K=K} \qquad (2.2)\frac{\Gamma\vdash K=K'}{\Gamma\vdash K'=K} \qquad (2.3)\frac{\Gamma\vdash K=K'\quad \Gamma\vdash K'=K''}{\Gamma\vdash K=K''}$$

$$(2.4)\frac{\Gamma\vdash k:K}{\Gamma\vdash k=k:K} \qquad (2.5)\frac{\Gamma\vdash k=k':K}{\Gamma\vdash k'=k:K} \qquad (2.6)\frac{\Gamma\vdash k=k':K\quad \Gamma\vdash k'=k'':K}{\Gamma\vdash k=k'':K}$$

Equality typing rules

$$(3.1)\frac{\Gamma\vdash k:K\quad \Gamma\vdash K=K'}{\Gamma\vdash k:K'} \qquad (3.2)\frac{\Gamma\vdash k=k':K\quad \Gamma\vdash K=K'}{\Gamma\vdash k=k':K'}$$

$$(3.3)\frac{\Gamma, x:K,\Gamma'\vdash J\quad \Gamma\vdash K=K'}{\Gamma, x:K',\Gamma'\vdash J}$$

Substitution rules

$$(4.1)\frac{\Gamma, x:K,\Gamma'\ \vdash \textbf{valid}\quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\ \vdash \textbf{valid}}$$

$$(4.2)\frac{\Gamma, x:K,\Gamma'\vdash K'\ \textbf{kind}\quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\vdash [k/x]K'\ \textbf{kind}} \qquad (4.3)\frac{\Gamma, x:K,\Gamma'\vdash k':K'\quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\vdash [k/x]k':[k/x]K'}$$

$$(4.4)\frac{\Gamma, x:K,\Gamma'\vdash K'=K''\quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\vdash [k/x]K'=[k/x]K''} \qquad (4.5)\frac{\Gamma, x:K,\Gamma'\vdash k'=k'':K'\quad \Gamma\vdash k:K}{\Gamma, [k/x]\Gamma'\vdash [k/x]k'=[k/x]k'':[k/x]K'}$$

$$(4.6)\frac{\Gamma, x:K,\Gamma'\vdash K'\ \textbf{kind}\quad \Gamma\vdash k=k':K}{\Gamma, [k/x]\Gamma'\vdash [k/x]K'=[k'/x]K'} \qquad (4.7)\frac{\Gamma, x:K,\Gamma'\vdash k':K'\quad \Gamma\vdash k_1=k_2:K}{\Gamma, [k_1/x]\Gamma'\vdash [k_1/x]k'=[k_2/x]k':[k_1/x]K'}$$

The kind Type

$$(5.1)\frac{\Gamma\ \vdash \textbf{valid}}{\Gamma\vdash \textbf{Type kind}} \qquad (5.2)\frac{\Gamma\vdash A:\textbf{Type}}{\Gamma\vdash El(A)\ \textbf{kind}} \qquad (5.3)\frac{\Gamma\vdash A=B:\textbf{Type}}{\Gamma\vdash El(A)=El(B)}$$

Dependent product kinds

$$(6.1)\frac{\Gamma\vdash K\ \textbf{kind}\quad \Gamma, x:K\vdash K'\ \textbf{kind}}{\Gamma\vdash (x:K)K'\ \textbf{kind}} \qquad (6.2)\frac{\Gamma\vdash K_1=K_2\quad \Gamma, x:K_1\vdash K_1'=K_2'}{\Gamma\vdash (x:K_1)K_1'=(x:K_2)K_2'}$$

$$(6.3)\frac{\Gamma, x:K\vdash k:K'}{\Gamma\vdash [x:K]k:(x:K)K'} \qquad (6.4)\frac{\Gamma\vdash K_1=K_2\quad \Gamma, x:K_1\vdash k_1=k_2:K}{\Gamma\vdash [x:K_1]k_1=[x:K_2]k_2:(x:K_1)K}$$

$$(6.5)\frac{\Gamma\vdash f:(x:K)K'\quad \Gamma\vdash k:K}{\Gamma\vdash f(k):[k/x]K'} \qquad (6.6)\frac{\Gamma\vdash f=f':(x:K)K'\quad \Gamma\vdash k_1=k_2:K}{\Gamma\vdash f(k_1)=f'(k_2):[k_1/x]K'}$$

$$(6.7)\frac{\Gamma, x:K\vdash k':K'\quad \Gamma\vdash k:K}{\Gamma\vdash ([x:K]k')(k)=[k/x]k':[k/x]K'} \qquad (6.8)\frac{\Gamma\vdash f:(x:K)K'\quad x\notin FV(f)}{\Gamma\vdash [x:K]f(x)=f:(x:K)K'}$$

Fig. 4.1: The new inference rules of LF

Base Coercion

$$(ST1)\frac{\Gamma \vdash A <_c B : \mathbf{Type} \in \mathcal{C}}{\Gamma \vdash A <_c B : \mathbf{Type}}$$

Congruence and Transitivity

$$(ST2)\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash c = c' : (A)B}{\Gamma \vdash A <_{c'} B : \mathbf{Type}}$$

$$(ST3)\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash A = A' : \mathbf{Type}}{\Gamma \vdash A' <_c B : \mathbf{Type}}$$

$$(ST4)\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash B = B' : \mathbf{Type}}{\Gamma \vdash A <_c B' : \mathbf{Type}}$$

$$(ST5)\frac{\Gamma \vdash A <_{c_1} B : \mathbf{Type} \quad \Gamma \vdash B <_{c_2} C : \mathbf{Type}}{\Gamma \vdash A <_{c_2 \circ c_1} C : \mathbf{Type}}$$

Substitution

$$(ST6)\frac{\Gamma, x : K, \Gamma' \vdash A <_c B : \mathbf{Type} \qquad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]A <_{[k/x]c} [k/x]B : \mathbf{Type}}$$

Weakening

$$(ST7)\frac{\Gamma, \Gamma' \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash K \ \mathbf{kind} \quad x \notin FV(\Gamma) \cup FV(\Gamma')}{\Gamma, x : K, \Gamma' \vdash A <_c B : \mathbf{Type}}$$

Context Retyping

$$(ST8)\frac{\Gamma, x : K, \Gamma' \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash K = K'}{\Gamma, x : K', \Gamma' \vdash A <_c B : \mathbf{Type}}$$

Fig. 4.2: The structural subtyping rules of $T[\mathcal{C}]_0$.

Definition 4.2. *(coherence) $\mathcal{C}$ is called a coherent set of coercive subtyping judgements, if in $T[\mathcal{C}]_0$ we have:*

1. *$\Gamma \vdash A <_c B : \mathbf{Type}$ implies $\Gamma \vdash A : \mathbf{Type}$, $\Gamma \vdash B : \mathbf{Type}$, $\Gamma \vdash c : (A)B$ are derivable in T.*

2. *We cannot derive $\Gamma \vdash A <_c A : \mathbf{Type}$, for any Γ, A, c.*

3. *$\Gamma \vdash A <_{c_1} B : \mathbf{Type}$ and $\Gamma \vdash A <_{c_2} B : \mathbf{Type}$ imply that $\Gamma \vdash c_1 = c_2 : (A)B$ is derivable in T.*

In rest part of this thesis, we will only consider the systems with a coherent set $\mathcal{C}$.

There are several differences with the formulation in [Luo99](or the rules in Figure 3.1) :

1. $\mathcal{C}$ is a set of judgements rather than rules.

2. two more inference rules are introduced: weakening (ST7) and context-retyping (ST8).

3. congruence rule in Figure 3.1 is split into three rules (ST2) (ST3) (ST4)

Actually, congruence rule in Figure 3.1

$$(Cong)\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash A = A' : Type \quad \Gamma \vdash B = B' : \mathbf{Type} \quad \Gamma \vdash c = c' : (A)B}{\Gamma \vdash A' <_{c'} B' : \mathbf{Type}}$$

has the same power with the three rules (ST2) (ST3) (ST4). This can be simply derived as follows.

On one hand, we can use (ST2), (ST3) and (ST4) to derive the (Cong), in the following way:

$$\frac{\dfrac{\dfrac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash c = c' : (A)B}{\Gamma \vdash A <_{c'} B : \mathbf{Type}} \quad \Gamma \vdash A = A' : \mathbf{Type}}{\Gamma \vdash A' <_{c'} B : \mathbf{Type}} \quad \Gamma \vdash B = B' : \mathbf{Type}}{\Gamma \vdash A' <_{c'} B' : \mathbf{Type}}$$

On the other hand, we can restrict the premises of (Cong) to derive (ST2) (ST3) and (ST4).

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash A = A : Type \quad \Gamma \vdash B = B : \mathbf{Type} \quad \Gamma \vdash c = c' : (A)B}{\Gamma \vdash A <_{c'} B : \mathbf{Type}}$$

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash A = A' : Type \quad \Gamma \vdash B = B : \mathbf{Type} \quad \Gamma \vdash c = c : (A)B}{\Gamma \vdash A' <_c B : \mathbf{Type}}$$

$$\frac{\Gamma \vdash A <_c B : \mathbf{Type} \quad \Gamma \vdash A = A : Type \quad \Gamma \vdash B = B' : \mathbf{Type} \quad \Gamma \vdash c = c : (A)B}{\Gamma \vdash A <_c B' : \mathbf{Type}}$$

Lemma 4.3. ***(Conservativity of** $T[\mathcal{C}]_0$ **over** T) $T[\mathcal{C}]_0$ is a conservative extension of T; that is, if J is not of form $A <_c B$: **Type**, then $\Gamma \vdash J$ is derivable in T if and only if $\Gamma \vdash J$ is derivable in $T[\mathcal{C}]_0$.*

Proof. Straightforward because, in $T[\mathcal{C}]_0$, a subtyping judgement cannot occur in the derivation of a judgement of any other form. □

Remark 4.4. *It seems that $T[\mathcal{C}]$ has less power than the original formulation $T[\mathcal{R}]$ does, because we don't include coercion rules. But as we said in section 3.4, we can use rules set $\mathcal{R}$ with the general subtyping rule (figure 3.1) to get a set of coercive judgements. For example, if we have rules:*

$$\frac{\vdash A <_c B}{\vdash List(A) <_{map(A,B,c)} List(B)} \qquad \frac{}{\vdash Nat <_c Bool}$$

we can get following judgements as basic coercion set

$\vdash Nat <_c Bool,$

$\vdash List(Nat) <_{map(Nat,Bool,c)} List(Bool),$

$\vdash List(List(Nat)) <_{map(List(Nat),List(Bool),map(Nat,Bool,c))} List(List(Bool)),$

$\ldots$

We should point out that this basic coercion judgement set might be infinite.

4.1.2 System $T[\mathcal{C}]_{0K}$

The system $T[\mathcal{C}]_{0K}$ is an intermediate system which extends $T[\mathcal{C}]_0$ with subkinding but no coercion application and definition rules. It is obtained from $T[\mathcal{C}]_0$ by adding the new subkinding judgement form $\Gamma \vdash K <_c K'$ and the following subkinding inference rules in Figure 4.3.

There are two differences with the formulation in [Luo99] (or the rules in Figure 3.2) :

1. Two more inference rules are introduced: weakening (SK10) and context-retyping (SK11).
2. The congruence rule in Figure 3.2 is split into three rules (SK5) (SK6) and (SK7)

Like (ST2) (ST3) (ST4) in $T[\mathcal{C}]_0$, the three rules (SK5) (SK6) and (SK7) have the same power with the congruence rule in Figure 3.2.

Lemma 4.5. *(**Conservativity of** $T[\mathcal{C}]_{0K}$ **over** $T[\mathcal{C}]_0$) $T[\mathcal{C}]_{0K}$ is a conservative extension of $T[\mathcal{C}]_0$; that is, if J is not of form $K <_c K'$, then $\Gamma \vdash J$ is derivable in $T[\mathcal{C}]_0$ if and only if $\Gamma \vdash J$ is derivable in $T[\mathcal{C}]_{0K}$.*

Proof. Straightforward because, in $T[\mathcal{C}]_{0K}$, a subkinding judgement cannot occur in the derivation of a judgement of any other form. □

With Lemma 4.3 and 4.5, we have:

Colloary 4.6. *(**Conservativity of** $T[\mathcal{C}]_{0K}$ **over** T) $T[\mathcal{C}]_{0K}$ is a conservative extension of T; that is, if J is not of form $A <_c B :$ **Type** or $K <_c K'$, then $\Gamma \vdash J$ is derivable in T if and only if $\Gamma \vdash J$ is derivable in $T[\mathcal{C}]_{0K}$.*

4.1.3 The Systems $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$

$T[\mathcal{C}]$ is the system obtained from $T[\mathcal{C}]_{0K}$ by adding the *coercive application* and *coercive definition* rules in Figure 4.4.

Basic subkinding rule

$$(SK1)\frac{\Gamma \vdash A <_c B : \mathbf{Type}}{\Gamma \vdash El(A) <_c El(B)}$$

Subkinding for dependent product kinds

$$(SK2)\frac{\Gamma \vdash K_1' <_{c_1} K_1 \qquad \Gamma, x' : K_1' \vdash [c_1(x')/x]K_2 = K_2' \qquad \Gamma, x : K_1 \vdash K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$$

where $c \equiv [f : (x : K_1)K_2][x' : K_1']f(c_1(x'))$;

$$(SK3)\frac{\Gamma \vdash K_1' = K_1 \qquad \Gamma, x' : K_1' \vdash K_2 <_{c_2} K_2' \qquad \Gamma, x : K_1 \vdash K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$$

where $c \equiv [f : (x : K_1)K_2][x' : K_1']c_2 f(x')$;

$$(SK4)\frac{\Gamma \vdash K_1' <_{c_1} K_1 \qquad \Gamma, x' : K_1' \vdash [c_1(x')/x]K_2 <_{c_2} K_2' \qquad \Gamma, x : K_1 \vdash K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$$

where $c \equiv [f : (x : K_1)K_2][x' : K_1']c_2(f(c_1(x')))$.

Congruence and Transitivity for subkinding

$$(SK5)\frac{\Gamma \vdash K_1 <_c K_2 \qquad \Gamma \vdash c = c' : (K_1)K_2}{\Gamma \vdash K_1 <_{c'} K_2}$$

$$(SK6)\frac{\Gamma \vdash K_1 <_c K_2 \qquad \Gamma \vdash K_1 = K_1'}{\Gamma \vdash K_1' <_c K_2}$$

$$(SK7)\frac{\Gamma \vdash K_1 <_c K_2 \qquad \Gamma \vdash K_2 = K_2'}{\Gamma \vdash K_1 <_c K_2'}$$

$$(SK8)\frac{\Gamma \vdash K_1 <_{c_1} K_2 \qquad \Gamma \vdash K_2 <_{c_2} K_3}{\Gamma \vdash K_1 <_{c_2 \circ c_1} K_3}$$

Substitution for subkinding

$$(SK9)\frac{\Gamma, x : K, \Gamma' \vdash K_1 <_c K_2 \qquad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K_1 <_{[k/x]c} [k/x]K_2}$$

Weakening for subkinding

$$(SK10)\frac{\Gamma, \Gamma' \vdash K_1 <_c K_2 \quad \Gamma \vdash K \ \mathbf{kind} \quad x \notin FV(\Gamma) \cup FV(\Gamma')}{\Gamma, x : K, \Gamma' \vdash K_1 <_c K_2}$$

Context Retyping for subkinding

$$(SK11)\frac{\Gamma, x : K, \Gamma' \vdash K_1 <_c K_2 \quad \Gamma \vdash K = K'}{\Gamma, x : K', \Gamma' \vdash K_1 <_c K_2}$$

Fig. 4.3: The subkinding rules of $T[\mathcal{C}]_{0K}$.

Coercive application rule

$$(CA1)\frac{\Gamma \vdash f : (x:K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) : [c(k_0)/x]K'}$$

$$(CA2)\frac{\Gamma \vdash f = f' : (x:K)K' \qquad \Gamma \vdash k_0 = k_0' : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) = f'(k_0') : [c(k_0)/x]K'}$$

Coercive definition rule

$$(CD)\frac{\Gamma \vdash f : (x:K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$$

Fig. 4.4: The coercive application and definition rules of $T[\mathcal{C}]$.

$T[\mathcal{C}]^*$ is the system obtained from $T[\mathcal{C}]_{0K}$ by adding the *coercive application* and *coercive definition* rules in Figure 4.5.

Coercive application rule

$$(CA^*1)\frac{\Gamma \vdash f : (x:K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f * k_0 : [c(k_0)/x]K'}$$

$$(CA^*2)\frac{\Gamma \vdash f = f' : (x:K)K' \qquad \Gamma \vdash k_0 = k_0' : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f * k_0 = f' * k_0' : [c(k_0)/x]K'}$$

Coercive definition rule

$$(CD^*)\frac{\Gamma \vdash f : (x:K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f * k_0 = f(c(k_0)) : [c(k_0)/x]K'}$$

Fig. 4.5: The coercive application and definition rules of $T[\mathcal{C}]^*$.

4.2 Relationship between the Systems

In this section we give an outline how our present proofs of conservative extension and definitional extension are structured, present the relations between the systems T, $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ for a coherent set $\mathcal{C}$ and provide the necessary precisions and improvements with respect to the earlier proof of "conservativity" in [SL02].

We have introduced the intermediate systems $T[\mathcal{C}]_0$ and $T[\mathcal{C}]_{0K}$ in our formulation. Obviously T, $T[\mathcal{C}]_0$, $T[\mathcal{C}]_{0K}$ are subsystems of $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$, so any judgement derivable in one of these systems is derivable in $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$. Since we have proved that $T[\mathcal{C}]_{0K}$ is a conservative extension of T, we will only show the relations between $T[\mathcal{C}]_{0K}$, $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ in the following parts.

From $T[\mathcal{C}]$ to $T[\mathcal{C}]_{0K}$ we will construct a coercion insertion algorithm to insert the coercions into the gaps. Similarly, from $T[\mathcal{C}]^*$ to $T[\mathcal{C}]_{0K}$, we will construct a coercion insertion algorithm to insert the coercions into the places marked by '$*$'. Consider the relation between $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$. Obviously, every derivation in $T[\mathcal{C}]^*$ may be transformed into a derivation in $T[\mathcal{C}]$. One only needs to "erase" all $*$. But in the direction from $T[\mathcal{C}]$ to $T[\mathcal{C}]^*$, the insertion of $*$ meets the following difficulty: nothing in the syntax of $T[\mathcal{C}]$ permits us to distinguish directly coercive and ordinary applications and to guarantee that in different premises of every inference inside a $T[\mathcal{C}]$-derivation the $*$ will be inserted at the same places (and hence the premises will be matching after insertion). We need to construct the algorithm that transforms derivations in $T[\mathcal{C}]$ into derivations in $T[\mathcal{C}]^*$ carefully. These coercion insertion algorithms are our main "tools". Notice that we cannot be sure in advance that they are defined for all derivations.

Insertion Algorithms. For two type theories T_1 and T_2, we write

$$f : T_1 \to T_2$$

if f is a function from the T_1-derivations to T_2-derivations.

We describe four algorithms, to be defined in the next section, which are such functions:

$$\begin{aligned}
\Theta &: T[\mathcal{C}] \to T[\mathcal{C}]_{0K} \\
\Theta^* &: T[\mathcal{C}]^* \to T[\mathcal{C}]_{0K} \\
\theta_1 &: T[\mathcal{C}]^* \to T[\mathcal{C}] \\
\theta_2 &: T[\mathcal{C}] \to T[\mathcal{C}]^*
\end{aligned}$$

- The algorithm Θ replaces the derivations of $\Gamma \vdash K_1 <_c K_2$ in the premises of coercive rules (CA1)(CA2)(CD) by derivations of $\Gamma \vdash c : (K_1)K_2$ and replaces the coercive applications by several ordinary applications.

- The algorithm Θ^* replaces the derivations of $\Gamma \vdash K_1 <_c K_2$ in the premises of coercive rules (CA*1)(CA*2)(CD*) by derivations of $\Gamma \vdash c : (K_1)K_2$ and replaces the coercive applications by several ordinary applications.

- The algorithm θ_1 replaces coercive applications of the form $f * a$ in $T[\mathcal{C}]^*$ by coercive applications $f(a)$ in $T[\mathcal{C}]$.

- The algorithm θ_2 replaces coercive applications in $T[\mathcal{C}]$ derivations by coercive applications in $T[\mathcal{C}]^*$, by inserting $*$ into appropriate places.

One would expect Θ to be composition of θ_2 and Θ^* but to assure this both θ_2 and Θ^* have to be defined. The relations are as that shown in Fig.4.6.

$$\begin{tikzcd}
T[\mathcal{C}] \arrow[d, shift left, "\theta_2"] \arrow[dr, "\Theta"] & & \\
T[\mathcal{C}]^* \arrow[u, shift left, "\theta_1"] \arrow[r, "\Theta^*"] & T[\mathcal{C}]_{0K} \arrow[r, "conservative"] & T
\end{tikzcd}$$

Fig. 4.6: Relations between $T[\mathcal{C}]$, $T[\mathcal{C}]^*$, $T[\mathcal{C}]_{0K}$ and T

Among the four transformations, the definition of θ_1 is the most easy one. It can be defined in the following way, and trivially be proved total:

θ_1 is a transformation that takes a derivation d in $T[\mathcal{C}]^*$, and returns a derivation $\theta_1(d)$ in $T[\mathcal{C}]$. It is defined by removing all the star-calculus '*' in d.

The definitions of Θ, θ_2 and Θ^* are much more difficult, after giving the definitions, we need to proof that Θ and Θ^* are total first , with inductive proof. Then we show that with some results of Θ and Θ^*, θ_2 is total as well. We will show the definitions of them in the next section.

Remark 4.7. *We need point out that the symbols for the algorithms in this thesis are slightly different with those we have in [LSX13].* θ_1 *is* θ *in [LSX13] and* θ_2 *is* θ^* *in [LSX13].*

4.3 Coercion Insertion Algorithms

In this section, we will present the algorithms for Θ, Θ^* and θ_2. Since they are very similar and related, we will mainly present the transformation Θ, and show the differences in Θ^* and θ_2. Also, we will show some main results of these transformations.

4.3.1 Basic Ideas of the Coercion Insertion Algorithms

In this subsection we shall first consider the problems arising with the definition of Θ and then explain the differences in the cases of Θ^* and θ_2.

The leading idea is to define a transformation Θ which transforms every derivation in $T[\mathcal{C}]$ into a derivation in $T[\mathcal{C}]_{0K}$, according to the description above.

Θ has to transform every application of a coercive application or definition rule into a certain inference (using ordinary non-coercive application) where the derivation of the subkinding judgement in the premise is replaced by the derivation of corresponding coercion and in the conclusion the coercion is inserted between the function and its argument. Otherwise we just keep the basic form of the rule. More precisely, for example, consider a derivation in $T[\mathcal{C}]$ that ends with a coercive application rule:

$$\frac{\Gamma \vdash f : (x:K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) : [c(k_0)/x]K'}$$

We assume that there are derivations in T which can derive judgements $\Gamma \vdash f : (x:K)K'$, $\Gamma \vdash k : K_0$, $\Gamma \vdash c : (K_0)K$. Then we could obtain a derivation in T as follows:

$$\frac{\Gamma \vdash f : (x:K)K' \qquad \dfrac{\Gamma \vdash k : K_0 \quad \Gamma \vdash c : (K_0)K}{\Gamma \vdash c(k_0) : K_0}}{\Gamma \vdash f(c(k_0)) : [c(k_0)/x]K'}$$

The coercive equality application rule (CA2) and coercive definition rule (CD) can be dealt with in the same way.

Now the question is how to extend the transformation to the whole derivation. It is natural to start from the leaves of the derivation tree, and move to the root (conclusion). When a coercive application or definition rule is met, the subkinding judgements are to be replaced by the derivation of the coercion terms and the rules modified as described in the previous paragraph.

To make this plan work, some difficulties must be solved.

The "conceptual core" of all difficulties is that the inserted coercions depend on the derivation. If an expression appears in two places in a derivation in $T[\mathcal{C}]$, the above process might map it to two expressions of T that are not identical. For example, a derivation that ends with e.g. the rule

$$\frac{\Gamma \overset{d_1}{\vdash} K = K' \quad \Gamma \overset{d_2}{\vdash} K' = K''}{\Gamma \vdash K = K''}$$

Under the transformation Θ, d_1 and d_2 become derivations $\Theta(d_1)$ and $\Theta(d_2)$ of, say, $\Gamma_1 \overset{\Theta(d_1)}{\vdash} K_1 = K_1'$ and $\Gamma_2 \overset{\Theta(d_2)}{\vdash} K_2' = K_2''$. We need to show that the corresponding kinds of contexts Γ_1, Γ_2 are equal in T(see Definition 3.5), and K_1', K_2' are equal in context Γ_1. If they are equal in T, we can complete the derivation in T[1]:

$$\frac{\Gamma_1 \overset{\Theta(d_1)}{\vdash} K_1 = K_1' \qquad \dfrac{\dfrac{\Gamma_2 \overset{\Theta(d_2)}{\vdash} K_2' = K_2'' \quad \vdash \Gamma_2 = \Gamma_1}{\Gamma_1 \vdash K_2' = K_2''} \quad \Gamma_1 \vdash K_2' = K_1'}{\Gamma_1 \vdash K_1' = K_2''}}{\Gamma_1 \vdash K_1 = K_2''}$$

[1] A part of this derivation uses the inference rule showed in Remark 4.1

Let's think of the following example for a more explicit description:

Example 4.8. *Consider vector type $Vec : (\mathbf{Type})(Nat)\mathbf{Type}$, then $Vec(Nat, n)$ is a vector type of Nat with length $n : Nat$. Suppose we have coercion $Even <_c Nat$ and $c' = c : (Even)Nat$, we can derive $Even <_{c'} Nat$ by the congruence rule. If we have $e : Even$, then consider the derivation:*

$$\frac{\Gamma, x : Vec(Nat, e), \Gamma' \overset{d_1}{\vdash} K = K' \quad \Gamma, x : Vec(Nat, e), \Gamma' \overset{d_2}{\vdash} K' = K''}{\Gamma, x : Vec(Nat, e), \Gamma' \vdash K = K''}$$

When we apply Θ on d_1 and d_2, we need to insert the gaps in the derivations, for example in $x : Vec(Nat, e)$. We might face the case that we insert c in $\Theta(d_1)$, but c' in $\Theta(d_2)$. Also we might have other different insertions in other places. Hence we can get

$$conc(\Theta(d_1)) \equiv \Gamma_1, x : Vec(Nat, c(e)), \Gamma'_1 \vdash K_1 = K'_1$$

$$conc(\Theta(d_2)) \equiv \Gamma_2, x : Vec(Nat, c'(e)), \Gamma'_2 \vdash K'_2 = K''_2$$

If we consider θ_2 and Θ^* instead of Θ the difficulties will go partly to the verification of correctness of the definition of Θ^* and partly to such verification for θ_2. Indeed, to proceed with θ_2 we need to verify that $*$-symbols are inserted in the same places of Γ and K' in the left premise and in Γ and K' of the right premise. In case of Θ^* we need not bother about places where $*$ is inserted, but we need to verify that coercion terms that are inserted in left and right premises are equal.

Coherence, as defined in Definition 4.2, is the key to solving *all* these problems. Suppose a gap in the same expression is filled with a coercion c_1 at one point in a derivation, and a coercion c_2 at another point. Then c_1 and c_2 may not be identical, but coherence is used to ensure that they will be judgementally *equal* in T and that after filling the gaps the expressions are equal in T. Of course, to implement this, a carefully planned induction is needed.

In fact, another element of coherence, the condition that there is no coercions of the form $Q <_c Q$, plays its role here as well. This is one of the main reasons why the part of the proof concerning the properties of θ_2 cannot be separated from the rest. Before we may show that θ_2 (insertion of $*$) is inverse to θ_1 we need to prove that it is defined for all derivations. This in its turn requires the proof that $*$ is inserted in the same places in the matching parts of different premises. Thus we need to show that there is no coercions $Q <_c Q$ in $T[\mathcal{C}]$, for otherwise it would be possible that an application $f(a)$ is considered as coercive in one place and as non-coercive in another. (In that case, θ_2 could insert $*$ in one branch, but not in the other.) This proof uses the fact that Θ is defined for all derivations.

Several less conceptual (more technical) challenges concern the organization of the inductive proof itself.

An important part of the inductive proof includes the lemmas about presupposed judgements. The reason is that the "common part" of the premise of a rule is a presupposed judgement of both. We prove that if Θ is defined for certain derivation d of $\Gamma \vdash J$ then it is defined for the derivations of presupposed judgements of $\Gamma \vdash J$. (The same for Θ^*, θ_2.) Another group of algorithms and lemmas takes care of the rules like substitution, weakening and contextual retyping which make direct inductive proof difficult. We define a canonical form of derivations, in which these rules only occur after an introduction of a subtyping judgement (from $\Gamma \vdash A <_c B \in \mathcal{C}$), and an algorithm $\mathbf{E}$ that moves these rules up to the "permitted" position. Then we show that if Θ is defined for d, it is also defined for the derivation $\mathbf{E}(d)$ in canonical form (similarly for Θ^* and θ_2.).The proofs use induction on the number of elimination steps.

In this thesis, when proving the properties of the coercive subtyping extension $T[\mathcal{C}]$, we have restricted T to be the known type theories such as UTT and Martin-Löf's type theory. The reason for this is to guarantee that the rule forms are preserved by coercion insertions, an aspect that was not sufficiently investigated in [SL02]. For example, let the derivation d in $T[\mathcal{C}]$

end by some rule r of T (r is not one of coercive rules), $d \equiv \dfrac{\dfrac{d_1}{J_1} \quad \cdots \quad \dfrac{d_n}{J_n}}{r(J_1, ..., J_n)}$, and $\Theta(d_1)$,... $\Theta(d_n)$ be defined. If $\Theta(d)$ is defined, it is supposed to end with an application of the same rule r. This assumes that, if J_i is a premise of an instance of r, then the conclusion of $\Theta(d_i)$ may be used as a premise of an instance of r. More precisely, some "adjustment" using the provable equalities of T may be permitted before r is applied, as we have seen in the previous part, but it has to be proved (by analysis of r) that the form of the conclusion of $\Theta(d_i)$ is appropriate[2].

Example 4.9. *Consider, for example, the elimination operator for an inductive type in UTT. Let it be, for simplicity, just the elimination operator $E[Nat]$ for the type of natural numbers Nat. Its kind is (omitting obvious occurrences of El):*

$$(C : (Nat)Type)(a : C(0))(f : (x : Nat)(y : C(x))C(succ(x)))(z : Nat)C(z)$$

The elimination operators are used in corresponding computation rules. In particular, there are two computation rules for $E[Nat]$,

$$(((E[Nat]C)a)f)0 = a : C(0),$$

$$(((E[Nat]C)a)f)(succ(n)) = (f(n))((((E[Nat]C)a)f)n),$$

corresponding to the standard computation steps of recursion. To be able to apply the same rules after coercion insertion the structure has to be preserved, in particular, no coercions are inserted between C and z, between succ and x, etc. *Notice that in case of θ_2, when only $*$ are inserted, the verification that no $*$ is inserted between C and z is still necessary.*

To take care of this and similar cases, the coherence at the kind level (i.e., non-derivability of the statements of the form $\Gamma \vdash K <_c K$) is used. The lemma with appropriate clause (Lemma 4.27) is used in the proof of

[2] Similar adjustment is necessary in case of Θ^*. In case of θ_2 only verification of matching.

the main theorem[3]. This is an illustration of the fact that the proofs are interconnected because this lemma uses the assumption that Θ is defined (to prove the absence of the judgements $\Gamma \vdash K <_c K$ in subderivations.)

In case of known type theories all rules can be inspected (as we did for UTT and Martin-Löf's type theory). For arbitrary T, a general condition on rule forms has to be elaborated, but we have not yet accomplished this task.

4.3.2 Transformations of Derivations: Exact Formulation

In this subsection, we will give the exact formulation of the transformations, and list the main results about them. Once again, we'll mainly focus on the definition of Θ, and show the difference for Θ^* and θ_2.

Formal Definitions of Transformations of Derivations

Definitions of $\Theta\colon T[\mathcal{C}] \to T[\mathcal{C}]_{0K}$ *and* $\Theta^*\colon T[\mathcal{C}]^* \to T[\mathcal{C}]_{0K}$. Θ and Θ^* are defined by induction on derivations d in $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$, respectively. In the following, we consider the cases in Θ's definition; for Θ^*, it is similar.

1. If d is already a derivation in $T[\mathcal{C}]_{0K}$, then $\Theta(d) \equiv (d)$.
2. If d ends with an instance of introduction of basic coercion in $\mathcal{C}$, then $\Theta(d) \equiv (d)$.
3. If d ends in an instance of a rule R with only one premise, say $d \equiv \dfrac{\dfrac{d_1}{J}}{R(J)}$ where $J \equiv \mathit{conc}(d_1)$, then

$$\dfrac{\dfrac{\Theta(d_1)}{(\mathit{conc}(\Theta(d_1)))}}{R(\mathit{conc}(\Theta(d_1)))}.$$

[3] This clause was first explicitly formulated by Marie-Magdeleine [MM09] and we are not going to focus on this part of proof in this thesis.

4. Suppose d ends by rule R with more than one premise, but not the coercion application or definition rules. Let $d \equiv \dfrac{\frac{d_1}{J_1} \cdots \frac{d_k}{J_k}}{R(J_k)}$ $(J_i \equiv conc(d_i), i = 1, ..., k)$,

$$\Theta(d) \equiv \cfrac{\cfrac{\cfrac{\cfrac{\Theta(d_1)}{conc(\Theta(d_1))} \cdots \cfrac{\Theta d_k}{conc(\Theta(d_k))} \quad \cfrac{\mathbf{?T}[\mathcal{C}]_{\mathbf{0K}} - \textbf{derivations}}{\textbf{Equalities}}}{\textbf{=-transitivity and context replacement}}}{J'_1 \ \cdots \ \cdots \ J'_k}}{R(J'_1 \ \cdots \ \cdots \ J'_k)}$$

   $\Theta(d)$ is defined only if the $?T[\mathcal{C}]_{0k} - \mathit{derivations}$ for required equalities exist.

5. Suppose $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f : (x : M)N \quad \Gamma \overset{d_2}{\vdash} k : K \quad \Gamma \overset{d_3}{\vdash} K <_c M}{\Gamma \vdash f(k) : [c(k)/x]N}$.
   Applying Θ to the derivations d_1, d_2, d_3, we get derivations $\Delta_1 \overset{\Theta(d_1)}{\vdash} f_1 : (x : M_1)N_1$, $\Delta_2 \overset{\Theta(d_2)}{\vdash} k_2 : K_2$ and $\Delta_3 \overset{\Theta(d_3)}{\vdash} K_3 <_{c_3} M_3$.
   Then $\Theta(d)$ is the derivation

$$\Theta(d) \equiv \cfrac{\overset{\Theta(d_1)}{\Delta_1 \vdash f_1 : (x:M_1)N_1} \quad \cfrac{\cfrac{\cfrac{\overset{co(\Theta(d_3))}{\Delta_3 \vdash c_3 : (K_3)M_3} \quad \cfrac{?_1}{\vdash \Delta_1 = \Delta_3}}{\Delta_1 \vdash c_3 : (K_3)M_3} \quad \cfrac{\cfrac{\overset{\Theta(d_2)}{\Delta_2 \vdash k_2 : K_2} \quad \cfrac{?_2}{\vdash \Delta_2 = \Delta_1}}{\Delta_1 \vdash k_2 : K_2} \quad \cfrac{?_3}{\Delta_1 \vdash K_2 = K_3}}{\Delta_1 \vdash k_2 : K_3}}{\Delta_1 \vdash c_3(k_2) : M_3} \quad \cfrac{?_4}{\Delta_1 \vdash M_1 = M_3}}{\Delta_1 \vdash c_3(k_2) : M_1}}{\Delta_1 \vdash f_1(c_3(k_2)) : [c_3(k_2)/x]N_1}$$

   Here $\Theta(d)$ is defined only if derivations $?_1$, $?_2$, $?_3$ and $?_4$ of the required equalities exist.

6. The cases where d ends in an instance of coercive application rule for equality (CA2) and coercive definition rule (CD) are handled similarly.

It can be useful to do the "adjustment" of the premises using equalities in a deterministic way, for example, to make the expressions that occur in the premises more to the right equal to the leftmost occurrence (like Γ_2 and Γ_3 were made equal to Γ_1 above). We shall assume, for certainty, that this convention is applied to both Θ and Θ^* in the same way.

Definitions of $\theta_1 \colon T[\mathcal{C}]^* \to T[\mathcal{C}]$ *and* $\theta_2 \colon T[\mathcal{C}] \to T[\mathcal{C}]^*$. The transformation θ_1 replaces coercive applications in $T[\mathcal{C}]^*$ by coercive applications in $T[\mathcal{C}]$ by erasing $*$. Its well-definedness and hence totality can be trivially verified.

The transformation θ_2 (insertion of $*$ into derivations in $T[\mathcal{C}]$) has the same cases as Θ and Θ^*. There are the following differences in treatment of these cases:

- Coercion application rules in $T[\mathcal{C}]$ are not replaced by ordinary applications but by coercive applications in $T[\mathcal{C}]^*$.
- No context retyping rules, =-transitivity etc. are inserted.

Main Results

We summarize the main result about the algorithms as follows, showing the relationship between systems $T[\mathcal{C}]$, $T[\mathcal{C}]^*$, $T[\mathcal{C}]_{0K}$ and T. Their proofs are given in the next section.

1. (totality of Θ and Θ^*)
   (a) Theorem 4.28: Θ is a well-defined total function.
   (b) Theorem 4.30: Θ^* is a well-defined total function.
2. (conservativity)
   (a) Theorem 4.33: $T[\mathcal{C}]^*$ is a conservative extension of $T[\mathcal{C}]_{0K}$.
   (b) Corollary 4.34: $T[\mathcal{C}]^*$ is a conservative extension of T.
3. (definitional equivalence)
   (a) Theorem 4.29: for any $T[\mathcal{C}]$-derivation d, $\Theta(d) \sim_{T[\mathcal{C}]} d$.
   (b) Theorem 4.31: for any $T[\mathcal{C}]^*$-derivation d, $\Theta^*(d) \sim_{T[\mathcal{C}]^*} d$.
4. (equivalence between $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$)

(a) Theorem 4.36: θ_2 is a well-defined total function.

(b) Theorem 4.37: θ_1 is the inverse of θ_2 (with respect to the identity derivations).

(c) Corollary 4.38: the type theories $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ are equivalent.

(d) Corollary 4.39: the composition $\Theta^* \circ \theta_2$ is defined and equal to Θ.

Remark 4.10. *Among the points above, point 2 (conservativity) and 3 (definitional equivalence) are the real main properties we want to show. 2(b) and 3(b) form what was called "definitional extension" in Chapter 3 (Section 3.5)*

4.4 The Proof of The Theorems

The key point and most difficulty part of the theorems proof is that Θ, Θ^* and θ_2 are well defined total functions. Since the totality proof of Θ^* and θ_2 are very similar with that of Θ, in subsection 4.4.1, we will only focus on showing the totality of Θ. In subsection 4.4.2, we will present the other theorems for our result.

4.4.1 Totality of The Transformations

Basic Idea

The basic idea of the totality proof is that transformations, somehow, "go through" the presuppositions. More precisely, consider rule (2.2), for example:

$$\frac{\Gamma \overset{d_1}{\vdash} K = K' \quad \Gamma \overset{d_2}{\vdash} K' = K''}{\Gamma \vdash K = K''}$$

Under the transformation Θ, d_1 and d_2 become derivations $\Theta(d_1)$ and $\Theta(d_2)$ of, say, $\Gamma_1 \overset{\Theta(d_1)}{\vdash} K_1 = K_1'$ and $\Gamma_2 \overset{\Theta(d_2)}{\vdash} K_2' = K_2''$. We need to show that the corresponding kinds of contexts Γ_1, Γ_2 are equal in $T[\mathcal{C}]_{0K}$, and K_1', K_2' are equal in context Γ_1, so that we can make the following derivation:

$$\dfrac{\Gamma_1 \overset{\Theta(d_1)}{\vdash} K_1 = K_1' \qquad \dfrac{\dfrac{\Gamma_2 \overset{\Theta(d_2)}{\vdash} K_2' = K_2'' \quad \vdash \Gamma_2 = \Gamma_1}{\Gamma_1 \vdash K_2' = K_2''} \qquad \Gamma_1 \vdash K_2' = K_1'}{\Gamma_1 \vdash K_1' = K_2''}}{\Gamma_1 \vdash K_1 = K_2''}$$

On one hand $\Gamma_1 \overset{\Theta(d_1)}{\vdash} K_1 = K_1'$ has presupposed judgement $\Gamma_1 \vdash K_1'$ **kind**, $\Gamma_2 \overset{\Theta(d_2)}{\vdash} K_2' = K_2''$ has presupposed judgement $\Gamma_2 \vdash K_2'$ **kind**. On the other hand, $\Gamma \overset{d_1}{\vdash} K = K'$ has presupposed judgement $\Gamma \overset{d_1'}{\vdash} K'$ **kind**, $\Gamma \overset{d_2}{\vdash} K' = K''$ has presupposed judgement $\Gamma \overset{d_2'}{\vdash} K'$ **kind**, and $\Theta(d_1')$ and $\Theta(d_2')$ would be defined, say $\Delta_1 \overset{\Theta(d_1')}{\vdash} M_1'$ **kind** and $\Delta_2 \overset{\Theta(d_2')}{\vdash} M_2'$ **kind**.

With lemma 4.26, we can show the following equality of judgements:

$$(\Gamma_1 \vdash K_1' \ \mathbf{kind}) = (\Delta_1 \vdash M_1' \ \mathbf{kind})$$

$$(\Gamma_2 \vdash K_2' \ \mathbf{kind}) = (\Delta_2 \vdash M_2' \ \mathbf{kind})$$

Meanwhile, with lemma 4.27, we can show,

$$(\Delta_1 \vdash M_1' \ \mathbf{kind}) = (\Delta_2 \vdash M_2' \ \mathbf{kind})$$

Hence, we have,

$$(\Gamma_1 \vdash K_1' \ \mathbf{kind}) = (\Gamma_2 \vdash K_2' \ \mathbf{kind})$$

By the definition of the equality of judgements, we can get the corresponding equalities we need.

The Systems T^-, $T[\mathcal{C}]_0^-$, $T[\mathcal{C}]_{0K}^-$, $T[\mathcal{C}]^-$ and $T[\mathcal{C}]^{*-}$

As we have mentioned above, in order to make our theorems proofs go through, we need to eliminate the weakening, substitution, context retyping rules. Hence, we introduce the following intermediate systems without these rules.

Definition 4.11. *We give the definition for systems T^-, $T[\mathcal{C}]_0^-$, $T[\mathcal{C}]_{0K}^-$, $T[\mathcal{C}]^-$ and $T[\mathcal{C}]^{*-}$.*

1. *The systems T^- and $T[\mathcal{C}]_0^-$ are obtained from system T and $T[\mathcal{C}]_0$ respectively, by removing the rules (1.4), (3.3), and (4.1)-(4.7).*
2. *The systems $T[\mathcal{C}]_{0K}^-$, $T[\mathcal{C}]^-$ and $T[\mathcal{C}]^{*-}$ are obtained from $T[\mathcal{C}]_{0K}$, $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$, respectively, by removing rules (1.4), (3.3), (4.1)-(4.7), (SK9), (SK10) and (SK11).*

We need to point out that in $T[\mathcal{C}]_0^-$, $T[\mathcal{C}]_{0K}^-$, $T[\mathcal{C}]^-$ and $T[\mathcal{C}]^{*-}$ we still have the weakening context-retyping and substitution rules for subtyping (ST6, ST7 and ST8) These are different with the definitions $T[\mathcal{C}]_0^-$, $T[\mathcal{C}]_{0K}^-$ and $T[\mathcal{C}]^-$ are different with the definitions in [SL02]. (ST6) (ST7) and (ST8) are not included in the definitions of these systems in [SL02]. Elimination of these three rules was problematic in [SL02] and we find that keeping them doesn't affect our proof.

Now, we will give algorithms for weakening, substitution and context retyping in $T[\mathcal{C}]_{0K}^-$, $T[\mathcal{C}]^-$ and $T[\mathcal{C}]^{*-}$. At the same time, we will propose presupposition algorithms in these systems for the presupposed judgements. All the algorithms are constructed inductively on the derivations. The order of these algorithms is not that straightforward, we find that algorithms for some rules depend on some others, and some have to be given simultaneously. Hence we have to deal them carefully in a right order.

First of all, the algorithm for the weakening rules ((1.4) and (SK10)) could be proposed. We can also give the algorithms for statement 1-3 of presuppositions in Definition 3.4. Next, the algorithms for the substitution rules (4.1) - (4.5) and (SK7) could be proposed together. However, the rest algorithms for substitution (4.6) (4.7)(let's call them equality substitution rules), context retyping (3.3) (SK11) and statement 4-8 of presuppositions in Definition 3.4 depend on each other. Figure 4.7 shows the relation order of the algorithms.

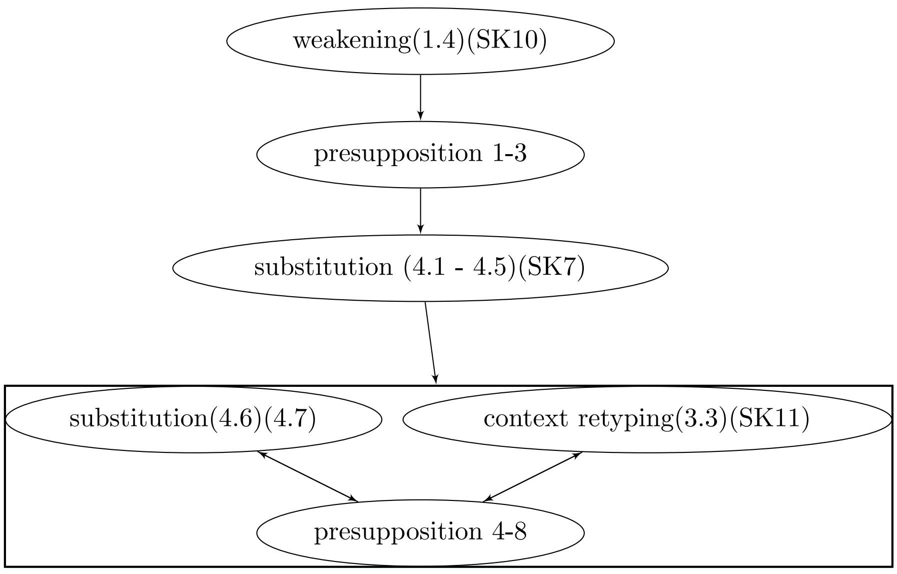

Fig. 4.7: The algorithms 1

To show the difficulties more precisely, we can consider the following cases in details.

1. To give the algorithm for presupposed judgement statement 4 of Definition 3.4, which says that $\Gamma \vdash K_1 = K_2$ has presuppositions $\Gamma \vdash K_1$ **kind** and $\Gamma \vdash K_2$ **kind**, we need to consider the case of rule (6.2):

$$\frac{\Gamma, x : K_1 \overset{d_1}{\vdash} K_2 = K_2' \quad \Gamma \overset{d_2}{\vdash} K_1 = K_1'}{\Gamma \vdash (x : K_1)K_2 = (x : K_1')K_2'}.$$

By induction we obtain the derivations of $\Gamma, x : K_1 \vdash K_2$ **kind** and $\Gamma, x : K_1 \vdash K_2'$ **kind**. We immediately derive the presupposed judgement $\Gamma \vdash (x : K_1)K_2$ **kind**

$$\frac{\Gamma, x : K_1 \vdash K_2 \ \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 \ \ \mathbf{kind}},$$

but to derive another presupposed judgement $\Gamma \vdash (x : K_1')K_2'$ **kind** we need to apply context-retyping first

$$\frac{\dfrac{\Gamma, x : K_1 \vdash K_2' \ \ \mathbf{kind} \quad \Gamma \vdash K_1 = K_1'}{\Gamma, x : K_1' \vdash K_2' \ \ \mathbf{kind}}}{\Gamma \vdash (x : K_1')K_2' \ \ \mathbf{kind}}$$

2. To give the algorithm for context retyping rule

$$\frac{\Gamma, x : K, \Gamma' \overset{d}{\vdash} J \quad \Gamma \overset{d'}{\vdash} K = K'}{\Gamma, x : K', \Gamma' \vdash J}$$

where J is of form: **valid**, K_0 **kind**, $k_0 : K_0$, $K_1 = K_2$, $k_1 = k_2 : K_0$, *or* $K_1 <_c K_2$ (this rule combines (3.3) and (SK11)).

We need to consider the following case of rule (1.2) for derivation d:

$$\frac{\Gamma_1 \overset{d_1}{\vdash} K_1 \ \ \mathbf{kind}}{\Gamma_1, y : K_1 \vdash \mathbf{valid}}(y \notin FV(\Gamma))$$

In this case, it implies that $\Gamma, x : K, \Gamma' \vdash J$ is syntactically equal to $\Gamma_1, y : K_1 \vdash$ **valid**. When Γ' is not an empty context, we could get the derivation for $\Gamma, x : K', \Gamma' \vdash J$ by induction. But if Γ' is an empty context, we need to show that there's derivation for $\Gamma, x : K' \vdash$ **valid**. To generate this, we need to use the presupposed judgement $\Gamma \vdash K'$ **kind** of $\Gamma \vdash K = K'$ in order to apply rule (1.2).

$$\frac{\Gamma \vdash K' \ \ \mathbf{kind}}{\Gamma, x : K' \vdash \mathbf{valid}}$$

The above shows that the algorithms for context retyping and presupposition statement 4-8 of Definition 3.4 depend on each other. Similarly, when we consider the equality substitution rules:

$$\frac{\Gamma, x : K, \Gamma' \vdash K' \ \mathbf{kind} \quad \Gamma \vdash k = k' : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k'/x]K'}$$

$$\frac{\Gamma, x : K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]k' = [k_2/x]k' : [k_1/x]K'}$$

The algorithm for them and the presupposition statement 4-8 of Definition 3.4 depend on each other as well.

To solve this problem, instead of giving all these algorithms together, we will consider the following tricky way which makes the order of all the algorithms clearer.

Rather than giving the algorithms of equality substitution rules and context retyping rule directly, we introduce some weaker rules for them, called *weak equality substitution* rules and *weak context retyping* rule.

$$\frac{\Gamma, x : K, \Gamma' \vdash K' \ \mathbf{kind} \quad \Gamma \vdash k_1 = k_2 : K \quad \Gamma \vdash k_1 : K \quad \Gamma \vdash k_2 : K}{\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]K' = [k_2/x]K'} \quad \textit{(weak-sub1)}$$

$$\frac{\Gamma, x : K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k_1 = k_2 : K \quad \Gamma \vdash k_1 : K \quad \Gamma \vdash k_2 : K}{\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]k' = [k_2/x]k' : [k_1/x]K'} \quad \textit{(weak-sub2)}$$

$$\frac{\Gamma, x : K, \Gamma' \vdash J \quad \Gamma \vdash K = K' \quad \Gamma \vdash K \ \mathbf{kind} \quad \Gamma \vdash K' \ \mathbf{kind}}{\Gamma, x : K', \Gamma' \vdash J} \quad \textit{(weak-ctx)}$$

(J is of form: **valid**, K_0 **kind**, $k_0 : K_0$, $K_1 = K_2$, $k_1 = k_2 : K_0$, *or* $K_1 <_c K_2$).

The differences between these weaker rules and original rules (4.6), (4.7), (3.3) and (SK11) are putting some presupposed judgements of the premises as the new rules' premises.

Now, the algorithms for these rules don't require the presupposition algorithms for statement 4-8 of Definition 3.4. We can give algorithm for these three weaker rules first, and then the algorithms for statement 4-8 of Definition 3.4. Finally, with the presupposition algorithm and the algorithms for these weak rules together, we can present the algorithms for rules (4.6), (4.7), (3.3) and (SK11). Figure 4.8 shows the order of the algorithms which are given in the Appendix A.

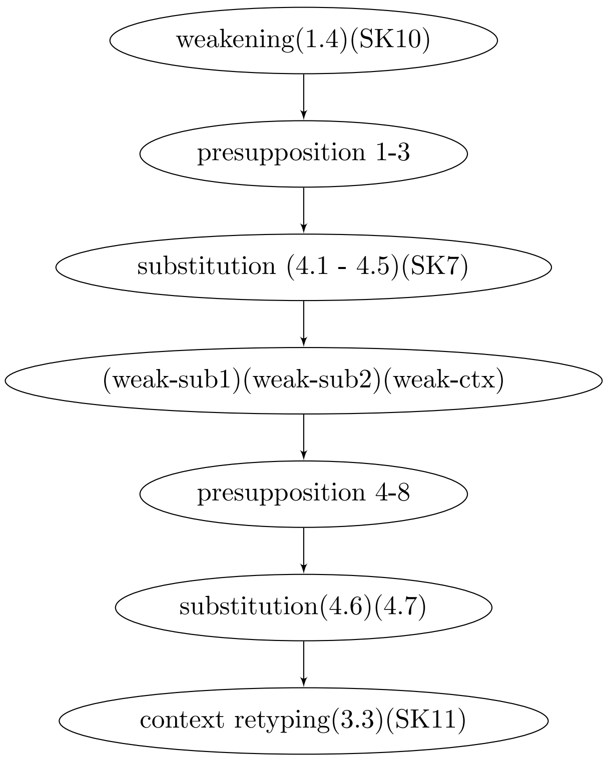
weakening(1.4)(SK10)
presupposition 1-3
substitution (4.1 - 4.5)(SK7)
(weak-sub1)(weak-sub2)(weak-ctx)
presupposition 4-8
substitution(4.6)(4.7)
context retyping(3.3)(SK11)

Fig. 4.8: The algorithms 2

We have the following lemmas for $T[\mathcal{C}]$, and we can give similar lemmas for systems $T[\mathcal{C}]^*$ and $T[\mathcal{C}]_{0K}$. And in these different systems, we will still use the same name for the corresponding algorithms with different domains.

Lemma 4.12. *(**Algorithms for weakening**) In $T[\mathcal{C}]^-$, there is an algorithm to derive the rules (1.4) and (SK10).*

Proof. By the algorithm A.2 in Appendix. □

The following *presupposition algorithms* take a derivation of a judgement J and return derivations of the presuppositions of J.

Lemma 4.13. ***(Presupposition algorithms 1)*** *There exist algorithms pre_1, pre_2, pre_3 from derivations of $T[\mathcal{C}]^-$ to derivations of $T[\mathcal{C}]^-$ that satisfy the following properties.*

1. *If d is a derivation of $\Gamma_1, \Gamma_2 \vdash J$, then $pre_1(d, \Gamma_1)$ is a derivation of $\Gamma_1 \vdash$* **valid***;*
2. *d is a derivation of $\Gamma_1, x : K_0, \Gamma_2 \vdash J$, then $pre_2(d, \Gamma_1)$ is a derivation of $\Gamma_1 \vdash K_0$* **kind***;*
3. *If d is a derivation of $\Gamma \vdash (x : K_1)K_2$* **kind***, then $pre_3(d)$ is a derivation of $\Gamma, x : K_1 \vdash K_2$* **kind***;*

Proof. By the algorithm A.3 in Appendix. □

Lemma 4.14. ***(Algorithms for substitution rules)*** *In $T[\mathcal{C}]^-$, there are algorithms to eliminate the substitution rules (4.1-4.5, SK7).*

Proof. By the algorithm A.4 in Appendix. □

Lemma 4.15. *In $T[\mathcal{C}]^-$, there are algorithms for rules (weak-sub1) and (weak-sub2)*

Proof. By the algorithm A.5 in Appendix. □

Lemma 4.16. *In $T[\mathcal{C}]^-$, there is algorithm for the rule (weak-ctx).*

Proof. By the algorithm A.6 in Appendix. □

Lemma 4.17. ***(presupposition algorithms 2)*** *There exist algorithms $pre_4^1, pre_4^2, \ldots, co$ from derivations of $T[\mathcal{C}]^-$ to derivations of $T[\mathcal{C}]^-$ that satisfy the following properties.*

1. *If d is a derivation of $\Gamma \vdash K_1 = K_2$, then $pre_4^1(d)$ is a derivation of $\Gamma \vdash K_1$* **kind** *and $pre_4^2(d)$ is a derivation of $\Gamma \vdash K_2$* **kind***;*
2. *If d is a derivation of $\Gamma \vdash k_1 = k_2 : K$, then $pre_5^1(d)$ is a derivation of $\Gamma \vdash k_1 : K$ and $pre_5^2(d)$ is a derivation of $\Gamma \vdash k_2 : K$;*

3. *If d is a derivation of $\Gamma \vdash \Sigma : K$, then $pre_6(d)$ is a derivation of $\Gamma \vdash K$ **kind** (Σ denotes term or term equality here);*

4. *If d is a derivation of $\Gamma \vdash A <_c B : \mathbf{Type}$, then $pre_7^1(d)$ is a derivation of $\Gamma \vdash A : \mathbf{Type}$, $pre_7^2(d)$ is a derivation of $\Gamma \vdash B : \mathbf{Type}$ and $co_t(d)$ is a derivation of $\Gamma \vdash c : (A)B$;*

5. *If d is a derivation of $\Gamma \vdash K_1 <_c K_2$, then $pre_8^1(d)$ is a derivation of $\Gamma \vdash K_1$ **kind**, $pre_8^2(d)$ is a derivation of $\Gamma \vdash K_2$ **kind** and $co(d)$ is a derivation of $\Gamma \vdash c : (K_1)K_2$.*

Proof. By the algorithm A.7 in Appendix. □

Now, applying pre_4^1, pre_4^2, pre_5^1 and pre_5^2 on lemma 4.15 and 4.16, we can give algorithms for rules (4.6), (4.7) , (3.3) and (SK11).

Lemma 4.18. ***(Algorithm for equality substitution rules)*** *In $T[\mathcal{C}]^-$, there are algorithms for the equality substitution rules (4.6) and (4.7).*

Proof. By the algorithm A.8 in Appendix. □

Lemma 4.19. ***(Algorithm for context retyping rules)*** *In $T[\mathcal{C}]^-$, there are algorithms for the rule of context replacement (3.3) and (SK11).*

Proof. By the algorithm A.9 in Appendix. □

Theorem 4.20. *There is an algorithm $\mathbf{E}$ that transforms every derivation d in $T[\mathcal{C}]$ into a derivation of the same judgement in $T[\mathcal{C}]^-$. Further, if d is a derivation in $T[\mathcal{C}]_{0K}$, then $\mathbf{E}(d)$ is a derivation in $T[\mathcal{C}]_{0K}^-$.*

Proof. Proved by lemmas 4.12, 4.14, 4.18 and 4.19. □

Colloary 4.21. *For any derivation d in $T[\mathcal{C}]$(or $T[\mathcal{C}]_{0K}$), $conc(\mathbf{E}(d)) \equiv d$.*

Transitivity Elimination

Theorem 4.22. (Elimination of transitivity of subkinding in $T[\mathcal{C}]_{0K}$.) *There is an algorithm, transforming every derivation of the judgement $\Gamma \vdash K <_c K'$ in $T[\mathcal{C}]_{0K}$ into a derivation of the judgement $\Gamma \vdash K <_{c'} K'$ in the same calculus not containing rules SK6, SK7, SK8.*

Proof. By algorithms A.14, A.15 and A.16. □

Colloary 4.23. *In $T[\mathcal{C}]_{0K}$, coherence holds in the following sense:*

1. *if $\Gamma \vdash K = K'$ in $T[\mathcal{C}]_{0K}$, then $\Gamma \vdash K <_c K'$ is not derivable in $T[\mathcal{C}]_{0K}$;*

2. *if $\Gamma \vdash K <_c K'$ and $\Gamma \vdash K <_{c'} K'$ in $T[\mathcal{C}]_{0K}$, then $\Gamma \vdash c = c' : (K)K'$*

Presupposition Lemmas of Derivations

Lemma 4.24. *Let d be a derivation, and suppose $\Theta(d)$ is defined. Then:*

1. *conc$(\Theta(d))$ and conc(d) are judgements of the same form.*

2. *If d_0 is a sub-derivation of d, then $\Theta(d_0)$ is defined.*

3. *If d_0 is a sub-derivation of d, d_0' is another derivation of conc(d_0), $\Theta(d_0) \sim \Theta(d_0')$, and d' is obtained from d by replacing d_0 with d_0', then $\Theta(d')$ is defined and $\Theta(d) \sim \Theta(d')$.*

Proof. Induction on d and use the definition of Θ. □

Lemma 4.25. *Let d be a derivation in $T[\mathcal{C}]$ and $\mathbf{E}$ be the algorithm from lemma 4.20. If $\Theta(d)$ is defined, then $\Theta(\mathbf{E}(d))$ is, and $\Theta(\mathbf{E}(d)) \sim \Theta(d)$.*

Proof. Structural induction on the derivation of d. If d doesn't end by weakening, context-retyping or substitution rules, simply apply induction hypothesis of the premises. Otherwise, assume d has the form $\frac{d_1 \dots d_k}{J} r$, where

d_1,...,d_k are the derivations of the premises. In standard cases the derivation d_1 may be written as $\dfrac{\dfrac{D_1}{J_1}\cdots\dfrac{D_m}{J_m}}{J'}R$. r is the rule to be eliminated and R is the last rule of d_1. Then

$$\mathbf{E}(d) \equiv \mathbf{E}(\frac{\dfrac{D_1...D_m}{J'}R \quad d_2\ ...\ d_k}{J}r) \equiv \frac{\mathbf{E}(\dfrac{D_1 d_2...d_k}{J_1}r)\ ...\ \mathbf{E}(\dfrac{D_m d_2...d_k}{J_m}r)}{J}R$$

Since $\Theta(d)$ is defined, by lemma 4.24, we have that $\Theta(d_1), ...\Theta(d_k), \Theta(D_1), ...\Theta(D_m)$ are defined and get equalities of the corresponding kinds and contexts in these derivations. With these equalities, we could show that $\Theta(E(d))$ is defined. Then by I.H., $\Theta(E(d)) \sim \Theta(d)$.

There are some special cases where the end of d has a different structure, and we can deal them with the equalities generated from the inductive hypothesis and the definition of Θ. Take *weakening* as an example, other cases are similar. If $d \equiv \dfrac{\Gamma,\Gamma' \overset{d_1}{\vdash} J \quad \Gamma,\Gamma'' \overset{d_2}{\vdash} \mathbf{valid}}{\Gamma,\Gamma'',\Gamma' \vdash J}$. The only exceptional case is $d_1 \equiv \dfrac{\Gamma_0 \overset{d_0}{\vdash} K_0\ \mathbf{kind}}{\Gamma_0, x : K_0 \vdash \mathbf{valid}}$, $\Gamma \equiv \Gamma_0, x : K$ and $\Gamma' \equiv <>$. In this case, $\mathbf{E}(d) \equiv d_2$. Since $\Theta(d)$ is defined, for the subderivation d_1 and d_2, $\Theta(d_1)$ and $\Theta(d_2)$ are defined. By the definition of $\Theta(d)$, we have $\Delta_1 \overset{\Theta(d_1)}{\vdash} \mathbf{valid}$, $\Delta_2, \Delta_2'' \overset{\Theta(d_2)}{\vdash} \mathbf{valid}$, $\vdash \Delta_1 = \Delta_2$ $conc(\Theta(d)) \equiv (\Delta_1, \Delta_2'' \vdash \mathbf{valid}) = (\Delta_2, \Delta_2'' \vdash \mathbf{valid})$. Also we have $\Theta(\mathbf{E}(d)) \equiv \Theta(d_2)$. So $conc(\mathbf{E}(\Theta(d))) \equiv (\Delta_2, \Delta_2'' \vdash \mathbf{valid}) = conc(\Theta(d))$ Hence we can get $\Theta(\mathbf{E}(d)) \sim \Theta(d)$. □

The following lemma shows that, if we wish to apply a presupposition algorithm and Θ to a given derivation, then the order in which we apply the two is not important.

Lemma 4.26. *(presupposition lemma)Let d be a derivation in $T[\mathcal{C}]^-$, and suppose $\Theta(d)$ is defined.*

1. *If d is a derivation of $\Gamma_1, \Gamma_2 \vdash J$, then $\Theta(pre_1(d, \Gamma_1))$ is defined and*

$$\Theta(pre_1(d, \Gamma_1)) \sim pre_1(\mathbf{E}(\Theta(d)), \Theta_d(\Gamma_1))$$

where, if Γ_1 has length n, then $\Theta_d(\Gamma_1)$ consists of the first n entries in the context of $conc(\Theta(d))$.

2. *If d is a derivation of $\Gamma_1, x : K, \Gamma_2 \vdash J$, then $\Theta(pre_2(d, \Gamma_1))$ is defined and*

   $$\Theta(pre_2(d, \Gamma_1)) \sim pre_2(\mathbf{E}(\Theta(d)), \Theta_d(\Gamma_1))$$

   where $\Theta_d(\Gamma_1)$ is as above.

3. *If d is a derivation of $\Gamma \vdash (x : K_1)K_2$ **kind**, then $\Theta(pre_3(d))$ is defined and*

   $\Theta(pre_3(d)) \sim pre_3(\mathbf{E}(\Theta(d)))$;

4. *If d is a derivation of $\Gamma \vdash K_1 = K_2$, then $\Theta(pre_4^1(d))$ and $\Theta(pre_4^2(d))$ are defined and*

   $\Theta(pre_4^1(d)) \sim pre_4^1(\mathbf{E}(\Theta(d)))$,

   $\Theta(pre_4^2(d)) \sim pre_4^2(\mathbf{E}(\Theta(d)))$;

5. *If d is a derivation of $\Gamma \vdash k_1 = k_2 : K$, then $\Theta(pre_5^1(d))$ and $\Theta(pre_5^2(d))$ are defined and*

   $\Theta(pre_5^1(d)) \sim pre_5^1(\mathbf{E}(\Theta(d)))$,

   $\Theta(pre_5^2(d)) \sim pre_5^2(\mathbf{E}(\Theta(d)))$;

6. *If d is a derivation of $\Gamma \vdash \Sigma : K$, then $\Theta(pre_6(d))$ is defined and*

   $\Theta(pre_6(d)) \sim pre_6(\mathbf{E}(\Theta(d)))$;

7. *If d is a derivation of $\Gamma \vdash A <_c B :$ **Type**, then $\Theta(pre_7^1(d))$, $\Theta(pre_7^2(d))$ and $\Theta(co_t(d))$ are defined and*

   $\Theta(pre_7^1(d)) \sim pre_7^1(\mathbf{E}(\Theta(d)))$,

   $\Theta(pre_7^2(d)) \sim pre_7^2(\mathbf{E}(\Theta(d)))$,

   $\Theta(co_t(d)) \sim co_t(\mathbf{E}(\Theta(d)))$;

8. *If d is a derivation of $\Gamma \vdash K_1 <_c K_2$, then $\Theta(pre_8^1(d))$, $\Theta(pre_8^2(d))$ and $\Theta(co(d))$ are defined and*

   $\Theta(pre_8^1(d)) \sim pre_8^1(\mathbf{E}(\Theta(d)))$,

   $\Theta(pre_8^2(d)) \sim pre_8^2(\mathbf{E}(\Theta(d)))$,

   $\Theta(co(d)) \sim co(\mathbf{E}(\Theta(d)))$.

Proof. Structural induction on derivation d, following the cases in the algorithm A.3 and A.7. In some cases, we need to use equalities generated from the induction hypothesis and the definition of Θ. Without losing generality, we show the proof of two typical cases in proving pre_6 and the other cases are similar.

1. Rule(3.4), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k : K}{\Gamma \vdash k = k : K}$. By definition $pre_6(d) \equiv pre_6(d_1)$.
   Since $\Theta(d)$ is defined, for the subderivation d_1, $\Theta(d_1)$ is defined, and by the definition, $\Theta(d) \equiv \dfrac{\Delta \overset{\Theta(d_1)}{\vdash} m : M}{\Delta \vdash m = m : M}$. By I.H. $\Theta(pre_6(d_1))$ is defined, hence $\Theta(pre_6(d))$ is defined, and $\Theta(pre_6(d_1)) \sim pre_6(\mathbf{E}(\Theta(d_1)))$. With Lemma 4.25, we have:

$$\begin{aligned} conc(\Theta(pre_6(d)) &\equiv conc(\Theta(pre_6(d_1))) \\ &= conc(pre_6(\mathbf{E}(\Theta(d_1)))) \\ &\equiv (\Delta \vdash M\ \ \mathbf{kind}) \\ &\equiv conc(pre_6(\mathbf{E}(\Theta(d)))) \end{aligned}$$

   So we have $\Theta(pre_6(d)) \sim pre_6(\mathbf{E}(\Theta(d)))$.

2. Rule(6.7), $d \equiv \dfrac{\Gamma, x : K \overset{d_1}{\vdash} k' : K' \quad \Gamma \overset{d_2}{\vdash} k : K}{\Gamma \vdash ([x : K]k')(k) = [k/x]k' : [k/x]K'}$.
   By definition $pre_6(d) \equiv sub_2(pre_6(d_1)), d_2)$, put in another way, we can consider $pre_6(d)$ as $pre_6(d) \equiv \mathbf{E}(d')$,where $d' \equiv \dfrac{\overset{pre_6(d_1)}{\Gamma, x : K \vdash K'\ \mathbf{kind}} \quad \Gamma \overset{d_2}{\vdash} k : K}{\Gamma \vdash [k/x]K'\ \mathbf{kind}}$
   Since $\Theta(d)$ is defined, $\Theta(d_1)$ and $\Theta(d_2)$ are defined. We have $\Delta_1, x : M_1 \overset{\Theta(d_1)}{\vdash} m' : M_1'$, $\quad \Delta_2 \overset{\Theta(d_2)}{\vdash} m : M_2$, $\quad \vdash \Delta_1 = \Delta_2$, $\quad \Delta_1 \vdash M_1 = M_2$

$conc(\Theta(d)) \equiv (\Delta_1 \vdash [x : M_1]m'(m) = [m/x]m' : [m/x]M_1')$

By I.H. $\Theta(pre_6(d_1))$ is defined and $\Theta(pre_6(d_1)) \sim pre_6(\mathbf{E}(\Theta(d_1)))$.

Let $\Theta(pre_6(d_1))$ be the derivation $\overset{\Theta(pre_6(d_1))}{\Pi, x : N \vdash N'}$ **kind**, we have

$(\Delta_1, x : M_1 \vdash M_1'\ \ \mathbf{kind}) = (\Pi, x : N \vdash N'\ \ \mathbf{kind})$

which means

$\vdash \Delta_1 = \Pi, \qquad \Delta_1 \vdash M_1 = M, \qquad \Delta_1, x : M_1 \vdash M_1' = N'$

So we can derive $\vdash \Pi = \Delta_2$ and $\Pi \vdash N = M_2$, hence $\Theta(d')$ is defined, $conc(\Theta(d')) \equiv \Pi \vdash [m/x]N'$, $\Theta(d') \sim pre_6(\mathbf{E}(\Theta(d)))$.

By lemma 4.25, $\Theta(\mathbf{E}(d'))$ is defined and $\Theta(\mathbf{E}(d')) \sim \Theta(d') \sim pre_6(\mathbf{E}(\Theta(d)))$. It means that $\Theta(pre_6(d))$ is defined and $\Theta(pre_6(d)) \sim pre_6(\mathbf{E}(\Theta(d)))$.

□

Lemma 4.27. *Suppose d_1 and d_2 are two derivations in $T[\mathcal{C}]^-$, and $\Theta(d_1)$ and $\Theta(d_2)$ are defined.*

1. *If d_1 and d_2 are both derivations of $\Gamma \vdash$ **valid**, then*

   $\Theta(d_1) \sim \Theta(d_2)$

2. *If d_1 and d_2 are both derivations of $\Gamma \vdash K$ **kind**, then*

   $\Theta(d_1) \sim \Theta(d_2)$

3. *If d_1 is a derivation of $\Gamma \vdash k : K_1$, d_2 is a derivation of $\Gamma \vdash k : K_2$, then*

   $\Theta(d_1) \sim \Theta(d_2)$

4. *If d is a derivation of $\Gamma \vdash K_1 <_c K_2$ and $\Theta(d)$ is defined, then the judgement $\Theta_d(\Gamma) \vdash \Theta_d(K_1) = \Theta_d(K_2)$ is not derivable in T, where $\Theta_d(\Gamma)$, $\Theta_d(K_1)$, $\Theta_d(K_2)$ denote the corresponding part of the judgement $conc(\Theta(d))$.*

5. *If d_1 is a derivation of $\Gamma \vdash K_1 <_c K_2$ and d_2 is a derivation of $\Gamma \vdash K_1 <_{c'} K_2$ in $T[\mathcal{C}]$ and $\Theta(d_1)$, $\Theta(d_2)$ are defined then the equality $\Theta_{d_i}(\Gamma) \vdash \Theta_{d_1}(c) = \Theta_{d_2}(c') : (\Theta_{d_i}(K_1))\Theta_{d_i}(K_2)$ is derivable in T. Here Θ_{d_i} (i = 1, 2) are defined as Θ_d above.*

Proof. For 1-3, induction on the sum of the sizes of the two derivations. It is trivial when d_1 and d_2 are derivations in T^-. If d_1 and d_2 end with the same rule, and this rule is not (CA1), then the proof presents no difficulty.

If they both end with (CA1) with different coercions inserted in the third premise, we can show that the transformation of the two coercions are equal by coherence (Corollary 4.23).

The only difficult case that remains is that one derivation ends with the application rule (6.5), while the other ends with the coercive application rule (CA1). We shall prove that this case is impossible.

Suppose

$$d_1 \equiv \frac{\Gamma \overset{d_1'}{\vdash} f : (x : K_1)K_1' \quad \Gamma \overset{d_1''}{\vdash} k : K_1}{\Gamma \vdash f(k) : [k/x]K_1'}$$

$$d_2 \equiv \frac{\Gamma \overset{d_2'}{\vdash} f : (x : K_2)K_2' \quad \Gamma \overset{d_2''}{\vdash} k : K_0 \quad \Gamma \overset{d_2'''}{\vdash} K_0 <_c K_2}{\Gamma \vdash f(k) : [c(k)/x]K_2'}$$

By definition of Θ, for $\Theta(d_1)$, there are derivations

$\Delta_1 \overset{\Theta(d_1')}{\vdash} f_1 : (x : M_1)M_1'$, $\Delta_2 \overset{\Theta(d_1'')}{\vdash} m : M_2$

and equalities

$\vdash \Delta_1 = \Delta_2$, $\Delta_1 \vdash M_1 = M_2$;

for $\Theta(d_2)$, there are derivations

$\Sigma_1 \overset{\Theta(d_2')}{\vdash} f_2 : (x : N_1)N_1'$, $\Sigma_2 \overset{\Theta(d_2'')}{\vdash} n : N_0$, $\Sigma_3 \overset{\Theta(d_2''')}{\vdash} N_0' <_{c'} N_3$,

and equalities

$\vdash \Sigma_1 = \Sigma_2$, $\vdash \Sigma_2 = \Sigma_3$, $\Sigma_1 \vdash N_1 = N_3$, $\Sigma_2 \vdash N_0 = N_0'$.

On the other hand, by the inductive hypothesis, we have $\Theta(d_1') \sim \Theta(d_2')$ and $\Theta(d_1'') \sim \Theta(d_2'')$.

Hence

$(\Delta_1 \vdash f_1 : (x : M_1)M_1') = (\Sigma_1 \vdash f_2 : (x : N_1)N_1')$

$(\Delta_2 \vdash m : M_2) = (\Sigma_2 \vdash n : N_0)$.

With all the equalities we can get both $\Delta_1 \vdash M_1 = N_0$ and $\Delta_1 \vdash N_0 <_{c'} M_1$, which contradicts coherence (Corollary 4.23).

For 4 and 5, we can easy prove them with the fact that $T[\mathcal{C}]_{0K}$ is coherent (corollary 4.23) and conservative over T (corollary 4.6).

$\square$

Theorem 4.28. *(Totality of* Θ*) For every derivation* d *in system* $T[\mathcal{C}]$*,* $\Theta(d)$ *is defined.*

Proof. We only need to check those cases which need derivation equalities (the cases 4-6 in the definition of Θ). Using the presupposition lemmas(lemma 4.26) and lemma 4.27, we can show the equalities we need. Take rule (2.3) and (CA1) as example cases and the other cases are similar.

- Rule(2.3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K = K' \quad \Gamma \overset{d_2}{\vdash} K' = K''}{\Gamma \vdash K = K''}$, with $\Theta(d_1)$ and $\Theta(d_2)$ are defined. Assume $\Delta_1 \overset{\Theta(d_1)}{\vdash} K_1 = K_1'$ and $\Delta_2 \overset{\Theta(d_2)}{\vdash} K_2' = K_2''$. With presupposition algorithms, we have derivations $\Gamma \overset{pre_4^2(d_1)}{\vdash} K'$ **kind**, $\Gamma \overset{pre_4^1(d_2)}{\vdash} K'$ **kind**, $\Delta_1 \overset{pre_4^2(\Theta(d_1))}{\vdash} K_1'$ **kind** and $\Delta_2 \overset{pre_4^1(\Theta(d_2))}{\vdash} K_2'$ **kind**.

  By lemma 4.26 we have

  $pre_4^2(\Theta(d_1)) \sim \Theta(\mathbf{E}(pre_4^2(d_1)))$, $pre_4^1(\Theta(d_2)) \sim \Theta(\mathbf{E}(pre_4^1(d_2)))$,

  and by lemma 4.27 we have

  $\Theta(pre_4^2(d_1)) \sim \Theta(pre_4^1(d_2))$,

  hence we have

$pre_4^2(\mathbf{E}(\Theta(d_1))) \sim pre_4^1(\mathbf{E}(\Theta(d_2)))$,

which means

$(\Delta_1 \vdash K_1' \ \mathbf{kind}) = (\Delta_2 \vdash K_2' \ \mathbf{kind})$.

By definition we have

$\vdash \Delta_1 = \Delta_2$ and $\Delta_1 \vdash K_1' = K_2'$.

Now we could show that $\Theta(d)$ is well-defined in the following way,

$$\Theta(d) \equiv \frac{\overset{\Theta(d_1)}{\Delta_1 \vdash K_1 = K_1'} \quad \dfrac{\Delta_1 \vdash K_1' = K_2' \quad \dfrac{\overset{\Theta(d_2)}{\Delta_2 \vdash K_2' = K_2''} \quad \vdash \Delta_2 = \Delta_1}{\Delta_1 \vdash K_2' = K_2''}}{\Delta_1 \vdash K_1' = K_2''}}{\Delta_1 \vdash K_1 = K_2'}$$

- Rule(CA1) $d \equiv \dfrac{\overset{d_1}{\Gamma \vdash f : (x:M)N} \quad \overset{d_2}{\Gamma \vdash k : K} \quad \overset{d_3}{\Gamma \vdash K <_c M}}{\Gamma \vdash f(k) : [c(k)/x]N}$, with $\Theta(d_1)$, $\Theta(d_2)$ and $\Theta(d_3)$ defined. Assume that

  $\overset{\Theta(d_1)}{\Delta_1 \vdash f_1 : (x : M_1)N_1}$, $\overset{\Theta(d_2)}{\Delta_2 \vdash k_2 : K_2}$ and $\overset{\Theta(d_3)}{\Delta_3 \vdash K_3 <_{c_3} M_3}$.

  With presupposition algorithms, we have derivations

  $\overset{pre_2(pre_3(pre_6(d_1)),\Gamma)}{\Gamma \vdash M \ \mathbf{kind}}$, $\overset{pre_8^2(d_3)}{\Gamma \vdash M \ \mathbf{kind}}$, $\overset{pre_6(d_2)}{\Gamma \vdash K \ \mathbf{kind}}$, $\overset{pre_8^1(d_3)}{\Gamma \vdash K \ \mathbf{kind}}$,

  $\overset{pre_2(pre_3(pre_6(\mathbf{E}(\Theta(d_1)))),\Delta_1)}{\Delta_1 \vdash M_1 \ \mathbf{kind}}$, $\overset{pre_8^2(\mathbf{E}(\Theta(d_3)))}{\Delta_3 \vdash M_3 \ \mathbf{kind}}$, $\overset{pre_6(\mathbf{E}(\Theta(d_2)))}{\Delta_2 \vdash K_2 \ \mathbf{kind}}$ and $\overset{pre_8^1(\mathbf{E}(\Theta(d_3)))}{\Delta_3 \vdash K_3 \ \mathbf{kind}}$

  By lemma 4.26, we have

  $\Theta(pre_2(pre_3(pre_6(d_1)), \Gamma)) \sim pre_2(pre_3(pre_6(\mathbf{E}(\Theta(d_1)))), \Delta_1)$,

  $\Theta(pre_8^2(d_3)) \sim pre_8^2(\mathbf{E}(\Theta(d_3)))$,

  $\Theta(pre_6(d_2)) \sim pre_6(\mathbf{E}(\Theta(d_2)))$

  $\Theta(pre_8^1(d_3) \sim pre_8^1(\mathbf{E}(\Theta(d_3)))$.

  Other other hand, by lemma 4.27, we have

  $\Theta(pre_2(pre_3(pre_6(d_1)), \Gamma)) \sim \Theta(pre_8^2(d_3))$ and $\Theta(pre_6(d_2)) \sim \Theta(pre_8^1(d_3))$.

  Hence the followings hold

  $pre_2(pre_3(pre_6(\Theta(d_1))), \Delta_1) \sim pre_8^2(\mathbf{E}(\Theta(d_3)))$

$\Theta(pre_6(d_2)) \sim \Theta(\mathbf{E}(pre_8^1(d_3)))$

which means

$(\Delta_1 \vdash M_1 \ \mathbf{kind}) = (\Delta_3 \vdash M_3 \ \mathbf{kind})$

$(\Delta_2 \vdash K_2 \ \mathbf{kind}) = (\Delta_3 \vdash K_3 \ \mathbf{kind})$.

By definition, we have

$\vdash \Delta_1 = \Delta_3$, $\Delta_1 \vdash M_1 = M_3$, $\vdash \Delta_2 = \Delta_3$ and $\Delta_2 \vdash K_2 = K_3$,

then we can get

$\vdash \Delta_2 = \Delta_1$ and $\Delta_1 \vdash K_2 = K_3$.

Recall the case(5) in the definition of Θ in subsection 4.3.2, all the required equalities marked with $?_1 - ?_4$ are proved, so $\Theta(d)$ is well-defined in this case.

$\square$

4.4.2 Other Theorems

With the totality of Θ, we have the following theorem:

Theorem 4.29. *If d is a derivation in $T[\mathcal{C}]$, then $d \sim \Theta(d)$ in $T[\mathcal{C}]$.*

Proof. Induction on the derivation of d, with the totality of Θ (theorem 4.28) and using the equalities in the definition of $\Theta(d)$.

With the same technique in proving the totality of Θ, we can prove the following two theorems.

Theorem 4.30. *For every derivation d in system $T[\mathcal{C}]^*$, $\Theta^*(d)$ is defined.*

Theorem 4.31. *If d is a derivation in $T[\mathcal{C}]^*$, then $d \sim \Theta^*(d)$ in $T[\mathcal{C}]^*$.*

Lemma 4.32. *If J is a judgement in $T[\mathcal{C}]_{0K}$, which has a derivation d in $T[\mathcal{C}]^*$. Then $conc(\Theta^*(d)) \equiv J$.*

Proof. Induction on the derivation d. Since J is a judgement in $T[\mathcal{C}]_{0K}$, the last rule of d can not be coercive application or definition rules ((CA1*) (CA2*) or (CD*)). For all the cases, we can simply make induction on the hypothesis. □

Theorem 4.33. *$T[\mathcal{C}]^*$ is a conservative extension of $T[\mathcal{C}]_{0K}$, in the sense that for every judgement J in $T[\mathcal{C}]_{0K}$, it is derivable in $T[\mathcal{C}]_{0K}$ if and only if it is derivable in $T[\mathcal{C}]^*$.*

Proof.

- (if) Suppose J has a derivation d in $T[\mathcal{C}]^*$. By lemma 4.32, $conc(\Theta^*(d)) \equiv J$, which means that $\Theta^*(d)$ is a derivation of J in $T[\mathcal{C}]_{0K}$.
- (only if) It can be trivially proved since $T[\mathcal{C}]_{0K}$ is a sub-system of $T[\mathcal{C}]^*$

□

With theorem 4.33 and corollary 4.6, we can get:

Colloary 4.34. *$T[\mathcal{C}]^*$ is a conservative extension of T.*

Colloary 4.35. *In $T[\mathcal{C}]$ ($T[\mathcal{C}]^*$) the derivability of $\Gamma \vdash K <_{c_1} K'$ and $\Gamma \vdash K <_{c_2} K'$ implies $\Gamma \vdash c_1 = c_2 : (K)K'$. The judgements of the form $\Gamma \vdash K <_c K$ cannot be derived in $T[\mathcal{C}]$ ($T[\mathcal{C}]^*$).*

Proof. The first part could be proved the same as proposition 3.11.

For the second part, if $\Gamma \vdash K <_c K$ could be derivable in $T[\mathcal{C}]$ ($T[\mathcal{C}]^*$), suppose the derivation is d. Then $\Theta(d)$ ($\Theta^*(d)$) is a derivation in $T[\mathcal{C}]_{0K}$ for a subkinding judgement of two equal kinds, this contradicts 4.23.

□

Next, we will prove the totality of θ_2, the main procedure is almost the same as Θ. One difference we should point out is that since the range of θ_2 is $T[\mathcal{C}]^*$ not $T[\mathcal{C}]_{0K}$, when we prove the lemma for θ_2 which corresponds to lemma 4.27 for Θ, we need to use corollary 4.35 for contradiction.

Theorem 4.36. *For every derivation d in system $T[\mathcal{C}]^*$, $\theta_2(d)$ is defined.*

Theorem 4.37. *θ_2 is the inverse of θ_1 (with respect to the identity transformation on derivations).*

Proof. Induction on the derivations, no case presents any difficulty. $\square$

Colloary 4.38. *The type theories $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ are equivalent.*

Proof. Induction on the derivations, no case presents any difficulty. $\square$

Colloary 4.39. *The composition $\Theta^* \circ \theta_2$ is defined and equal to Θ (with respect to the identity of derivations).*

Proof. Induction on the derivations, by the definition of Θ, Θ^* and θ_2 no case presents any difficulty. $\square$

5. COERCIVE SUBTYPING IN PLASTIC

Plastic [CL01] by Paul Callaghan, is a proof assistant which implements logical framework LF and UTT, coercion has been implemented in Plastic as well.

In this chapter, we will first discuss some useful coercion forms in LF, some of which cannot be trivially done in those type theories in direct syntax form. Then we give an introduction to Plastic with examples showing how inductive types could be defined in Plastic. Finally we will show how coercive subtyping is implemented in Plastic, the existing problems of Callaghan's version and the improvement we have done.

5.1 Coercions in a Logical Framework

In previous chapters, we have studied coercive subtyping in the logical framework LF. Although some proof assistants such as Plastic implements logical frameworks, most of the proof assistants implement type theories directly without implementing a logical framework. For example, in Coq, the Π-types are implemented directly : a Π-type (x:A)B in Coq corresponds to $\Pi(A, [x : A]B)$ in LF. Because of this difference, the coercion mechanism we have studied provides a more general tool than those based on a direct syntax. In particular, several forms of coercions are very useful in practice, as studied by Bailey in his PhD thesis [Bai99], can be captured by our coercion mechanism in LF.

5.1.1 Argument Coercions

This is the usual form of coercions and it is supported by all of the proof assistants that support coercion mechanisms. In a direct syntax, where Π-types are of the form $\Pi x : A.B$, argument coercions are given by the following rules:

$$\frac{f : \Pi x : A.B \quad a : A_0 \quad A_0 <_c A : Type}{f(a) : [c(a)/x]B}$$

and furthermore, f(a) = f (c(a)). When Π is specified in LF, the application operator is defined by means of the elimination operator:

$$app(A, [x : A]B, f, a) = E_\Pi(A, [x : A]B, [G : \Pi(A, [x : A]B)]B[a], [g : (x : A)B]g(a), f)$$

In our system of coercive subtyping, the following is a derivable rule:

$$\frac{f : (A, [x : A]B) \quad a : A_0 \quad A_0 <_c A : Type}{app(A, [x : A]B, f, a) : [c(a)/x]B}$$

and we have $app(A, [x : A]B, f, a) = app(A, [x : A]B, f, c(a))$.

5.1.2 Type Coercions

The so-called type coercions (or 'kind coercions') are those converting non-types into types. For instance, suppose Group is the type of (representations of) groups and $G : Group$. One often says:

for all groups G and for all elements of G, ...

Formally, this is represented as

$$\Pi G : Group.\Pi x : G.......$$

But this is not well-typed: G is not a type! For such applications, Bailey [Bai99] has considered the so-called type coercions that convert non-types to types. Eg., we convert the term G into G's carrier type $El(G)$. In a direct syntax, this has to be introduced separately from the argument coercion mechanism. However, when we consider coercions based on the logical framework, the above term is the following in LF:

$$\Pi(\mathit{Group}, [G : \mathit{Group}]\Pi(G, ...)),$$

which is equal to (by coercive subtyping)

$$\Pi(\mathit{Group}, [G : \mathit{Group}]\Pi(\mathit{El}(G), ...)),$$

where $\mathit{Group} <_{\mathit{El}} U$ with U being a type universe.

Therefore, the type coercions are just special cases of argument coercions in the logical framework.

5.1.3 Function Coercions

Sometimes, when we apply a mismatching function onto an object, we cannot coerce the object to be a matching type but we have coercions for the function which can make the application well-typed. Hence, there is another kind of coercions we would like to consider, converting from functions with a mismatching type into a matching type or even from non functions into functions. This cannot be done in the direct syntax, unless introducing a separate mechanism for function coercion.

For example, if the application $f\ a$ is not well-typed and there is no argument coercion that can be inserted to get it well-typed, we may coerce f into a function whose domain is the type of a with a coercion c, to get the well-typed term $(c(f))a$.

In an LF-based syntax, this is to coerce

$$\mathit{app}(A, B, f, a)$$

into

$$app(A, B, c(f), a)$$

Such function coercions are special cases of argument coercions in the LF-based coercion mechanism.

Compared with the direct syntax, the coercive subtyping in a logical framework provides us a wider range of coercion mechanisms, some of which have been discussed above. In a proof assistant that implements coercions based on a logical framework (like Plastic) these different forms of coercions are supported. In most systems that implement coercions based on a direct syntax (eg Matita), only argument coercions are supported, not type coercions, function coercions or other cases. However, even for latter systems, the above discussions give one a disciplined approach to indicating how further forms of coercions may be implemented.

Remark 5.1. *In Coq [Coq10], it introduces Funclass to help with function coercions. Funclass is a class of functions, whose objects are all terms with a function type; it allows us to write $f(x)$ when f is not a function but can be seen in a certain sense as a function such as bijection, functor, any structure morphism etc. Coq also introduces another abstract Sortclass which is the class of sorts. Objects of Sortclass are the terms whose type is a sort(Prop,Set,Type). It allows to write $x : A$ where A is not a type, but can be seen in a certain sense as a type such as set, group, category etc.*

5.2 Proof Assistant Plastic

Plastic is an implementation of LF and the type theory UTT presented in chapter 9 of [Luo94], with extension of coercive subtyping. It is originally implemented by Paul Callaghan [CL01]. In this section, we will present the basic syntax of Plastic, and give some examples to show how different inductive types could be defined in Plastic.

5.2.1 Declaration and Definition

In Plastic, we can declare terms to be of a given kind in the following way:

```
> [<name>,...,<name> : <kind>];
```

Or define a term in the following way:

```
> [<name> = <term>];
```

Example 5.2. *Here are some simple examples of declarations and definitions.*

```
> [A:Type];
> [a:A];
> [B=A];
> [b=a];
```

where Type is a kind as in LF, and all the terms on the right of a declaration should be a kind. One may notice that we write $a : A$ where A is a type, this is a short notation in Plastic, we will coerce A to $El(A)$ with kind coercion. It is actually the same as writing:

```
> [a:El A];
```

5.2.2 Product of Kinds

In LF, if we have two kinds K and K', we could have dependent product kind $(x : K)K'(x)$. In Plastic, we could define them in the following way,

```
(x:<term>) <term>
```

where $\langle term \rangle$ is a kind or an object of **Type** which would be coerced to be an object of kind. If it is non-dependent we could simply write

```
(_:<term>) <term>
```

or

```
<term> -> <term>
```

In LF, if $k(x) : K'(x)$, we can also define a term $[x : K]k(x)$ to be of kind $(x : K)K'(x)$. In Plastic, the notation of this is:

```
[x:<term>] <term>
```

$\langle term \rangle$ has the same requirement as the ones in the product kind.

If we want to apply a term f of kind $(x : K)K'(x)$ to a term $a : K$ in Plastic, we simply write $f\ a$ or $f(a)$

Example 5.3. *We could define some product kinds and terms of product kinds:*

```
> [A:Type];
> [B:A->Type]
> [pAB : (x: A) (B x)];
> [id = [x:A]x];
```

The same as above, A and B x will be coerced to $El(A)$ *and* $El(B(x))$. *And id will be of kind* $(x : El(A))El(A(x))$.

5.2.3 Inductive Types

The syntax for defining inductive types in Plastic is like this:

```
> Inductive
>     <paramerters>
>     [<name> : <kind>]
>    Constructors
>     [<name> : <kind>]
>     ......
```

Once an inductive type $\langle name \rangle$ is defined, the elimination operator $E\_\langle name \rangle$ will be generated automatically by Luo's algorithm in chapter 9 of [Luo94].

Example 5.4. *We could define the Nat and List in Plastic as follows:*

1. *Nat is defined as:*

   ```
   > Inductive
   >    [Nat:Type]
   >   Constructors
   >    [zero:Nat]
   >    [succ:(n:Nat)Nat];
   ```

   Nat will be a type with two constructors:

   ```
   zero : Nat
   succ : (n:Nat)Nat
   ```

   The elimination operator for E_Nat is generated as well:

   ```
   E_Nat : (C_Nat:El Nat->Type)
           El(C_Nat zero)
           ->((n:El Nat)El(C_Nat n)->El(C_Nat(succ n))
           ->(z:El Nat)El(C_Nat z)
   ```

2. *List of any type A is defined as:*

   ```
   > Inductive
   >    [A:Type]
   >    [List:Type]
   >   Constructors
   >    [nil:List]
   >    [cons: (a:El A)(l:List)List];
   ```

   We will get two constructors for List

```
nil  : List
cons : (a:El A)(l:List)List
```

The elimination rule of List E_List is generated as well:

```
E_List : (A:Type)
         (C_List:El(List A)->Type)
         El (C_List (nil A))
         ->
         ((a:El A)(l:El (List A ))El(C_List l)->El(C_List(cons A a l)))
         -> (z:El(List A))El(C_List z)
```

Example 5.5. *With the terms for elimination rules, we can define other rules for the inductive types:*

1. *For* Π*-types, we have the application rule*

   $$(app) \qquad \frac{\Gamma \vdash M : \Pi(x:A).B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}$$

   With E_Pi, we could define application operator of Π*-type in Plastic*

   ```
   > [ap = [A:Type][B:(_:A)Type][f:Pi A B][x:A] E_Pi A B
         ([G:El (Pi A B)]B x)([g:(x1:A)(B x1)]g x)f];
   ```

2. *For function space type, we have almost the same application rule as* Π *type. With E_Pi_, we could define application rule of Pi_ type*

   ```
   > [ap_ = [A:Type][B:Type][pi:Pi_ A B][x:A]E_Pi_ A B
         ([G:El (Pi_ A B)]B)([f:El A-> El B]f x) pi];
   ```

3. *For* Σ*-types, we have two projection* π_1 *and* π_2

   $$(\pi_1) \qquad \frac{\Gamma \vdash M : \Sigma(x:A).B}{\Gamma \vdash \pi_1(M) : A}$$

$$(\pi_2) \qquad \frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \pi_2(M) : [\pi_1(M)/x]B}$$

With E_Sigma, we could define π_1 and π_2 as well:

```
  >  [pi1 = [A:Type][B:(_:A)Type]
       E_Sigma A B([_:Sigma A B]A)([x:A][y:B x]x)];
  >  [pi2 = [A:Type][B:(_:A)Type][s:Sigma A B]
       E_Sigma A B([s1:Sigma A B]B(pi1 A B s1))([x:A][y:(B x)]y)s];
```

We predefine many other inductive types in the library. To use them, one only simply need to import with the command:

```
> import <libname>
```

5.3 Implementation of Coercions in Plastic

In this section, we will describe how coercive subtyping is implemented in Plastic, show how to use it in various forms of coercion with some examples, then talk about the defects and bugs in Callaghan's implementation and present our improvement.

5.3.1 Different Ways of Using Coercive Subtyping in Plastic

Plastic implements not only the coercions between the simple terms, but also dependent coercions, parameterized coercions and coercion rules. Users could define coercions to handle any combination of the following cases of coercion:

- Plain coercions, of kind $(A)B$.
- Dependent coercions, of kind $(x : A)B(x)$;

- Parameterized coercions, for example, of kind $(l : (n : Nat)(Vec\, n))List$ or as we described before $\pi_1 : (A : Type)(B : (A)Type)(\Sigma(A, B))A$, which are families of coercions.
- Coercion "rules", for example, if we have coercion from A to B, it is able to coerce $List(A)$ to $List(B)$.

For the coercion rules, we call the coercions in the premises to be *prerequisites* of the coercions in the conclusion. For example, in the following rule:

$$\frac{K_1 <_c K_2}{K_1' <_{c'} K_2'}$$

c is a prerequisite of c'.

More precisely, when we apply a term f of kind $(x : K_1)K_2$ on a term a, the coercion insertion goes with the following steps:

1. check the kind of a, let the type be K_1',
   - if K_1' is convertible to K_1, then $f(a)$ is well typed and no is coercion needed.
   - otherwise, we need to go to next step to search for applicable coercions.
2. if there is a coercion c from K_1' to K_1 in the context, then $f(a)$ is typable and equals to $f(c(a))$ of kind $[c(a)/x]K_2$.
3. if there's a parameterized coercion $c_0(p_1, p_2...)$ matches the form from K_1' to K_1 (see Example 5.7), we will go to check whether there're some parameters $m_1, m_2, ...$ makes $c_0(m_1, m_2, ...)$ from K_1' to K_1 . If so, $f(a)$ is of kind $[c(a)/x]K_2$ equals to $f(c(a))$, where $c \equiv c_0(m_1, m_2...)$.
4. if there's a coercion rule c_r matches the form from K_1' to K_1 (see Example 5.8), we will go to check whether the parameters and the prerequisites satisfied . If so, $f(a)$ is of kind $[c(a)/x]K_2$ equals to $f(c(a))$, where c is c_r applied with parameters and prerequisite coercions.

5. otherwise, $f(a)$ is alerted to be not well-typed.

Remark 5.6. *In Plastic, when we consider using coercive subtyping, we not only try to find the coercion for the required source and range kind, but also try to find coercion whose source and range kind are convertible to the requirement. What we say there's a coercion from K_1' to K_1 actually means that there's a coercion from K_0' to K_0, where K_0 is convertible to K_1 and K_0' is convertible to K_1',*

We will use two examples to illustrate how we apply parameterized coercions and coercion rules in more detail.

Example 5.7. *Consider coercion $\Sigma(A, B) <_{\pi_1} A$ for any A, B, where $\pi_1 : (v_1 : Type)(v_2 : (v_1)Type)(\Sigma(v_1, v_2))v_1$. Suppose we have terms $M : Type$, $N : (M)Type$, $a : \Sigma(M, N)$, $f : (M)M$. When we apply f to a, we require a term of type M, but we only have term of type $\Sigma(M, N)$. So we need to insert this gap, put in another word, we need to get a coercion from $\Sigma(M, N)$ to M. In Plastic we record $\Sigma(M, N)$ as "src" (source) and M as "tgt" (target):*

$$src - \ \Sigma(M, N) \qquad tgt - \ M$$

Checking the coercions, we have π_1 with variables, domain ("dom") and range ("rng") of this coercion are:

$$dom - \ \Sigma(v_1, v_2) \qquad rng - \ v_1$$

Comparing M with v_1, we try to match v_1 with M first. Then we compare $\Sigma(M, N)$ with $\Sigma(M, v_2)$, and match v_2 with N. Finally, we get a coercion

$$\pi_1(M, N) : \Sigma(M, N)M$$

which satisfies our coercion requirement, and make $f(a)$ to be well-typed, and equal to $f(\pi_1(M, N)(a)) : [\pi_1(M, N)(a)/x]M$.

Example 5.8. *Consider coercion rule, for any A, B*

$$\frac{A <_c B}{List(A) <_{map(A,B,c)} List(B)}$$

where map is defined as in chapter 3:

$$map : (v_1 : Type)(v_2 : Type)(c_0 : (v_1)v_2)(List(v_1))List(v_2)$$

Suppose we have terms $M, N : Type$, a coercion $M <_c N$, and we want to apply $f : (List(N))List(N)$ to term $a : List(M)$. We need a coercion from $List(M)$ to $List(N)$:

$$src - \ List(M) \qquad tgt - List(N)$$

Checking the coercions, we have map with variables, the coercion domain and range are:

$$dom - \ List(v_1) \qquad rng - List(v_2)$$

Comparing $List(N)$ with $List(v_2)$, we try to match v_2 with N. Then we compare $List(M)$ with $List(v_1)$, match v_1 with M. Now we need a term $c_0 : (v_1)v_2$, and $c : (M)N$ is what we need. Finally we get a coercion

$$map(M, N, c) : (List(M))List(N)$$

which satisfies our coercion requirement, and make

$$f(a) = f(map(M, N, c)(a)) : [map(M, N, c)(a)/x]List(N)$$

5.3.2 Declaring Coercions

The syntax of declaring coercion in Plastic is followed:

```
> Coercion
>   Parameters <decls>
>   Prerequisites <names>
>   =<term>:<type>
```

In this syntax, *term* is for the coercion term, *type* is the type of the term and is optional. *Parameters* and *Prerequisites* are also optional according to different coercions users want to define and we could use them to define parameterized coercion and coercion rules. *Prerequisites* requires some coercions as prerequisites. Once a user defines a coercion, the system will give a meta-variable name with "cx" as prefix to the coercion. The names will be cx1, cx2, cx3, ...

Example 5.9. *Here are examples of different ways to declare coercions where Nat and Bool are types (we omit Plastic code for some inductive data types).*

1. *Plain coercions. If we want to define a plain coercion c from Bool to Nat, we could simply write:*

   ```
   > [c : Bool -> Nat];
   > Coercion = c : Bool -> Nat;
   ```

   or simply

   ```
   > [c: Bool -> Nat];
   > Coercion = c;
   ```

   We also could define a concrete term to be a coercion. Suppose c1 is of kind (Bool)Nat and takes false to zero, and true to one (succ(zero)):

   ```
   > [c1 =  E_Bool ([x:Bool] Nat) (succ(zero)) zero];
   > Coercion = c1;
   ```

2. *Dependent coercions. We could define a function lv from List(Nat) to Vec(Nat,n) (n is the size of the list),*

$$lv(nil(Nat)) = vnil(Nat)$$

$$lv(cons(Nat, x, l)) = vcons(Nat, len(l), x, lv(l))$$

where len:(l:(List(Nat)))Nat gives the length of a List(Nat):

$$len(nil(Nat)) = zero$$

$$len(cons(Nat, x, l)) = succ(len(l))$$

now we could define lv as a coercion

```
> Coercion  = lv;
```

3. *Parameterized coercions. We could define a function vl from Vec(Nat,n) to List(Nat):*

$$vl(vnil(Nat)) = nil(Nat)$$

$$vl(vconst(Nat, n, x, v)) = cons(Nat, x, vl(v))$$

so we could define vl as a parameterized coercion, with parameter n:Nat.

```
> Coercion
>   Parameters [n:Nat]
>   = vl n;
```

4. *Coercion rules. We could define a rule, if there is a coercion from A to B, then we can have a coercion from List A to List B.*

```
> Coercion
>  Parameters [A,B:Type][f:A->B]
>  Prerequisites f
>   = map A B f : List A -> List B;
```

where map is a function maps every element from List(A) to List(B) with function f:(x:A)B defined as in Chapter 3.

Example 5.10. *We could combine the examples above, the following program will work which shows how these coercions work.*

```
> [c : Bool -> Nat];
> Coercion = c ;
> [a : Bool];
> [sa = succ(a)];
> Coercion
>   Parameters [A,B:Type][f:A->B]
>   Prerequisites f
>  = map A B f : List A -> List B;
> [g: (List Nat) -> Nat];
> [b: List Bool];
> [gb = g(b)];
```

Coercion c would be given a meta-variable name $cx1$ in the program, and the coercion rule is $cx2$. $cx2$ takes Bool, Nat and $cx1$ as parameters, and $cx1$ is also the coercion as prerequisite we mentioned above. $g(b)$ would be well typed through the coercion rule, and

$$succ(a) = succ(cx1\ a) : El(Nat)$$

$$g(b) = g(cx2(Bool, Nat, cx1(b))) : El(Nat)$$

Example 5.11. *The Plastic code for Example 3.1 is as follows:*

```
> [Man,Human,Prop:Type];
> [c:Man->Human];
> Coercion = c;
> [run: Human->Prop];
> [John:Man];
> [John_run: run John];
```

Coercion $c : (Man)Human$ *is given a meta-variable name* $cx1$ *in the program, and*

$$run\ John = run(cx1(John)) : El(Human)$$

5.3.3 Transitivity and Coherence

In Plastic, transitivity is implemented for basic coercions, those which do not have premises coercion. For example, if we have coercions $Even <_c Nat$, and $Nat <_{c'} Bool$, then we will generate a new coercion $[x : Even]c'(c(x))$ from $Even$ to $Bool$.

Coherence will be checked when a new coercion c from A to B is introduced, where c could be newly defined or generated by transitivity. To achieve this, Plastic checks to see if there is already a coercion from A to B. If no such coercion exists, we accept the new coercion. If there exists such a coercion c', check whether the two coercion terms c and c' are convertible; if they are convertible, we will do nothing with it; if they are not convertible, reject the coercion.

When parameterized coercions are specified or when coercion rules are used to introduce coercions, coherence checking is undecidable in general. Therefore, we need to show that, for example, certain coercion rules are coherent and hence can be used in practice. Here are some examples, which demonstrate that those parameterized coercions or coercion rules can be used to specify coercions in the considered applications.

- For Σ-types, the first projections π_1 as a parameterized coercion, as specified by the rule (Σ_{π_1}) in Example 5.5 , are coherent. This has been used effectively by many people (eg, [Bai99]) for notational abbreviations in proof development.

- The structural subtyping rules for all of the inductive types that are introduced by the schemata (eg, the rule for the type of lists in Example 3.13) are proved to be coherent ([LL05, LA08]). Such coercions specify natural subtyping relations for inductive types and can be used in many applications.

- There are many other coercion rules useful in various applications, examples of which include

  - the (ξ) rule together with the subtyping propagation rules are coherent and used in providing Intensional Manifest Fields for record types or Σ-types ([Luo09b]).
  - for the dot-types to be discussed below, the associated projection operators, together with the subtyping propagation operators for the dot-types, are coherent ([Luo10]). See the next chapter for dot-types and their implementation.

5.3.4 Problems and Improvement

However, there are some defects of Callaghan's implementation.

1. One can add a new coercion which is convertible to an existing one. For example, if we have defined $A <_c B$, when we have $c' = c$ or $c'' = [x : A](c(x))$, we can still declare c' or c'' to be a coercion. A special case is that one can declare a term $c : (A)B$ to be a coercion as many times as he wants. Since they are convertible to c, coherence still holds. But when we use coercions in Plastic, we will only use the first coercion among these convertible coercions. So the convertible coercions are actually redundant.

2. When we generate a new coercion with the transitivity rule, while there's an existing coercion of the same type, only the coercion generated by transitivity will be rejected, the coercions forming the composition will still be accepted. For instance, if we have coercion $A <_{c_1} B$,

$A <_{c_2} C$, and we want to introduce a new coercion $B <_{c_3} C$, by transitivity $[x : A](c_3(c_1(x)))$ should be a coercion from A to C, since c_2 is already a coercion of such type, $[x : A](c_3(c_1(x)))$ will not be added into context. But c_3 will still be added into the context in the old implementation. It is not reasonable, the newly introduced coercion c_3 violates the coherence with transitivity rule, it should not be accepted as a coercion.

3. There are some problems when the transitivity rule takes more than 3 coercions. For example. If we have $A <_{c_1} B$, and $C <_{c_2} D$, when we introduce another coercion $B <_{c_3} C$, the system will generate coercion $[x : A](c_3(c_1(x)))$ from A to C and coercion $[x : B](c_2(c_3(x)))$ from B to D. But it will never get the coercion from A to D which should be generated as well.

To amend these problems, we have improved coercive subtyping in Plastic according to each problem above as follows:

1. When adding a new coercion, check whether the term has been declared as coercion. If so, just ignore the declaration. If not, and if the coercion is of the same type as an existing coercion, check whether it is convertible to the existing one. If yes, ignore this declaration; if not, deny this declaration. Accept the other cases.

2. When we add a new coercion, we check the coherence of the term and all possible transitivity coercions. If the newly introduced coercion which is generated by transitivity rule is rejected, we will reject the term causing this transitivity as well. For example, if we have $A <_{c_1} C$, $A <_{c_2} B$ and we want to introduce $C <_{c_3} B$. By transitivity, we could generate a new coercion $[x : A]c_3(c_1(x))$ from A to B, but c_2 is already a coercion from A to B and they are not convertible. We will not only reject $[x : A]c_3(c_1(x))$, but also reject coercion c_3. To achieve this, whenever we find a term that violates coherence, i.e. it's of the same type with but not convertible to an existing coercion, algorithm

will fail. The context will be the same with the one before we enter the algorithm. So any coercion which might violate coherence itself or generate some coercion with rules to violate coherence will not be added.

3. The last problem was simply caused by a programming bug. Dealing with the coercion composition recursively as showed in algorithm 5.12 below will solve the problem.

The algorithm for adding a new plain coercion into context and check the transitivity and coherence is giving below:

Algorithm 5.12. *Declare c to be a plain coercion*

1. *Check whether c is of a product type. If c is not a product type, send a warning and stop, otherwise go to the next step.*
2. *Check whether c is already defined as a coercion. If c is already defined as a coercion, send a warning and stop, otherwise go to the next step.*
3. *Get c's type $(A)B$ from the context, check whether there's already a coercion from A to B, if so interrupt and end the algorithm, otherwise go to the next step.*
4. *Get all the existing coercions c_X from type X to A and all the coercions c_Y from type B to type Y. Consider the coercions generated by transitivity from X to B and from A to Y (see Figure 5.1). For the existing X and Y, take $< c\ c_X >= [x : X](c(c_X(x)))$ as coercion from X to B and $< c_Y\ c >= [x : A](c_Y(c(x)))$ as coercion from A to Y, go to step 3 to check the coherence for all these coercion recursively. If there's no such coercions, simply go to the next step.*
5. *Add the coercion c into the context.*

One may find that we add the coercion to the context in the last step. This makes sure that we will add this coercion to the context only when

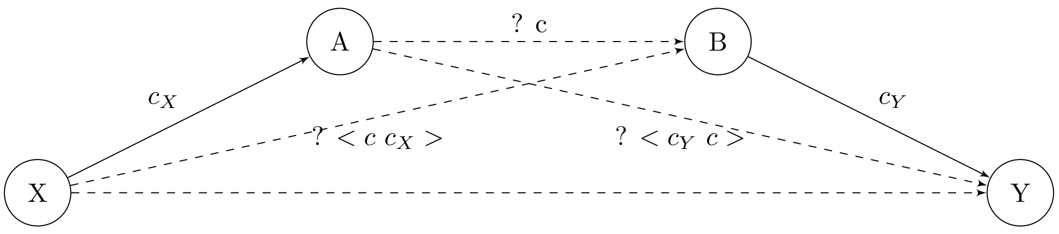

Fig. 5.1: Transitivity

all the coherent checking have been done. When we interrupt and end our algorithm, we do not record any change in the algorithm, and the state of the context will be the same state before starting the algorithm. This guarantees that, if the coercion we try to introduce is problematic (like not coherent), no action will be taken according to this coercion.

Example 5.13. *We will use some examples to show the improvement we have made for coercive subtyping in Plastic.*

1.
```
> [A, B : Type];
> [c1 :  A -> B];
> [c2 = c1];
> Coercion = c1;
> Coercion = c2;
```

In the previous version, both c1 and c2 will be defined as coercions. Now, the system will only add c1 as coercion and tell the user there's an convertible term to c2 which is already a coercion.

2.
```
> [A, B, C : Type];
> [c1 : A ->B ];
> [c2 : A ->C ];
> [c3 : B ->C ];
> Coercion = c1;
> Coercion = c2;
> Coercion = c3;
```

In the previous implementation, when we introduce the last line, the system will tell the user there's already a coercion from A to C which is *not convertible to [x:A]cx3(cx1(x)) (cx1, cx2 and cx3 are meta-variables in Plastic for c1, c2 and c3). However, c3 is still kept as a coercion. Now, we will not keep c3 as coercion in the context for this case.*

3.
```
> [A, B, C, D : Type];
> [c1 : A ->B ];
> [c2 : C ->D ];
> [c3 : B ->C ];
> Coercion = c1;
> Coercion = c2;
> Coercion = c3;
```

In the previous implementation, we will use transitivity to add coercions from A to C and from B to D, but the system cannot add coercion from A to D. After amending the program, the coercion from A to D will also be added.

6. DOT-TYPES WITH COERCIVE SUBTYPING

Dot-types, or sometimes called dot objects or complex types, are special data types. They were introduced by Pustejovsky in the Generative Lexicon Theory [Pus95] and studied by many others, including [Ash11]. Intuitively, a dot-type is formed from two constituent types that present distinct aspects of those objects in the dot-type. For example, a book may be considered to have two aspects: one informational (eg, when it is read) and the other physical (eg, when it is picked up). One may therefore consider a dot-type PHY • INFO whose objects, including books, have both physical and informational aspects. In particular, such objects can be involved in the linguistic phenomenon of copredication and dot-types play a promising role in its analysis and formalization.

Although the meaning of dot-types is intuitively clear, its proper formal account seems surprisingly difficult and tricky (see [Ash08] for a discussion). Researchers have made several proposals to model dot-types formally including, for example, [Ash11, AP05] and [Coo11, Coo07]. Besides discussions on whether the proposed solutions do capture and therefore give successful formal accounts of dot-types, most of these proposals are considered in the Montagovian setting which is based on Church's simply type theory [Chu40]. Compared to simply type theory, modern type theories(MTTs) may be classified into the predicative type theories such as Martin-Löf's type theory [NPS90, ML84] and the impredicative type theories such as the Calculus of Constructions (CC) [CH88] and the Unifying Theory of dependent Types (UTT) [Luo94]. In [Luo10], a formal treatment of dot-types in modern type theories has been proposed with the help of coercive subtyping and it is argued that, because in the formal semantics based on MTTs, common nouns

(CNs) are interpreted as types (rather than predicates as in the Montague semantics), the linguistic phenomena such as copredication can be given satisfactory treatments by means of dot-types.

In this chapter, we present an implementation of dot-types in the proof assistant Plastic [CL01], based on the formalization of dot-types in MTTs. As far as we know, this is the first attempt to implement the dot-types.[1] It allows us to use dot-types in the development of formal semantics in proof assistants and, at the same time, gives us a better understanding of dot-types and their relationship with other data types in type theory.

Dot-types are not ordinary inductive types, as found in the MTT-based proof assistants such as Agda, Coq and Plastic. In particular, for $A \bullet B$ to be a dot-type, the constituent types A and B should not share components (see the main text for the formal definition). In an implementation of dot-types, this special condition of type formation must be checked and adhered to. In order to make sure of this, we have to implement the dot-types as special data types, different from ordinary inductive types. We shall show how this is done in our implementation in Plastic.

6.1 Dot-types in Formal Semantics

In the Generative Lexicon Theory [Pus95], Pustejovsky has introduced the idea of employing dot-types to model various linguistic data that involve objects that have distinct aspects. Typical examples are concerned about copredication, where different aspects of a word are selected when predication comes into force. For example, in the following sentences [Ash11], the words ‘lunch’ and ‘book’ both have two distinct aspects to be selected: in (6.1) a ‘lunch’ was delicious as food and took forever as an event, and in (6.2) a ‘book’ was picked up as a physical object and mastered as an informational

[1] In [Luo11] some examples have been presented in Coq [Coq10] and, since Coq does not support dot-types, Σ-types were used to mimic dot-types, although the author was fully aware of the fact that this is in general impossible and leads to incoherence, since there is no guarantee that the constituent types of a dot-type do not share components.

object.

(6.1) The lunch yesterday was delicious but took forever.

(6.2) John picked up and mastered the mathematical book.

There have been studies of dot-types in various formal systems or semi-formal systems including, for example, [Pus95, AP05, Ash11, Pus11]. Most of these proposals are given in the Montagovian setting where, in particular, common nouns are interpreted as predicates. It has been argued that the way that CNs are interpreted in the Montagovian setting is incompatible with the subtyping postulates that the type of entities has subtypes *Event* (of events), PHY (of physical objects), INFO (of informational objects), etc. This leads to unnecessary difficulties and formal complications when formalizing dot-types. On the other hand, if we interpret CNs as types, as in the formal semantics based on MTTs, the treatment becomes straightforward and satisfactory [Luo10]. (See section 6.2.1 for further details.)

Usually the two aspects involved in a dot-type are incompatible: in the above examples, Food and Event are incompatible and so are the physical and informational objects. This incompatibility of the two aspects that form a dot-type was expressed by Pustejovsky as follows:

> *Dot objects have a property that I will refer to as* ***inherent polysemy****. This is the ability to appear in selectional contexts that are contradictory in type specification.* [Pus05]

In other words, an important feature is that, to form a dot-type $A \bullet B$, its constituent types A and B should not share common parts. For instance,

- PHY • PHY should not be a dot-type because its constituent types are the same type PHY.
- PHY • (PHY • INFO) should not be a dot-type because its constituent types PHY and PHY • INFO share the component PHY.

Put in another way, a dot-type $A \bullet B$ can only be formed if the types A and B do not share any components: it is a dot-type only when the constituent types A and B present different and incompatible aspects of the objects.

This incompatibility is one of the two key features based on which dot-types are introduced in MTTs: it is stipulated that the constituent types of a dot-type do not share components. The other feature is that the relationships between the dot-type and its constituent types are captured by means of coercive subtyping so that the dot-type is the subtype of both of its constituent types. We now turn to the type-theoretic formulation of dot-types.

6.2 Dot-types in Modern Type Theories

In this section, we show how dot-types can be introduced in modern type theories with the help of coercive subtyping [Luo10]. We will first explain informally, in the formal semantics based on MTTs, called *type-theoretical semantics*, henceforth, how to use dot-types to interpret copredication in natural language. Then we will lay down the formal rules of the dot-types in modern type theories.

6.2.1 Dot-types and Coercive Subtyping

Type-Theoretical Semantics

In [Ran94], Ranta has studied various semantic issues of natural languages in Martin-Löf's type theory, introducing the basic ideas of type-theoretical semantics based on MTTs. Unlike Montague grammar in which common nouns like Man and $Human$ are interpreted as functional subsets (or predicates) of entities, in the type-theoretical semantics based on modern type theories, common nouns are interpreted as types. For instance, in Montague grammar, Man and $Human$ are interpreted as objects of type $e \to t$, where e is the type of entities and t the type of propositions. In type-theoretical semantics, the interpretations of Man, $Human$ and $Book$ are types:

$$[[man]], [[human]], [[book]] : Type$$

This is natural in a modern type theory, which is many-sorted in the sense that there are many types like $[[man]]$ and $[[book]]$ consisting of objects standing for different sorts of entities, while the simple type theory may be thought of as single sorted in the sense that there is the type e of all entities. In a type-theoretical semantics, verbs and adjectives are interpreted as predicates. For example, we can have

$$\begin{array}{rcl} [[nice]] & : & [[book]] \to Prop \\ [[read]] & : & [[human]] \to [[book]] \to Prop \end{array}$$

where $Prop$ is the type of propositions. Modified common noun phrases could be interpreted by means of Σ-types of dependent pairs: for instance,

$$[[nice\ book]] = \Sigma([[book]], [[nice]])$$

Dot-type and Coercive Subtyping

Intuitively, a type of two aspects should be a subtype of a single aspect of those two, so a dot-type should be a subtype of its constituent types. For instance, it is natural to think that the type consisting of the objects with both aspects of food and event be a subtype of Food as well as a subtype of Event. Similarly, the type consisting of objects with both physical and informational aspects should be a subtype of the type PHY of physical objects and a subtype of the type of informational aspect:

$$\textsc{Phy} \bullet \textsc{Info} <_{c_1} \textsc{Phy}$$

$$\textsc{Phy} \bullet \textsc{Info} <_{c_2} \textsc{Info}$$

Consider sentence (6.2) again. In a type-theoretical semantics, we may assume that

$$\begin{array}{rcl} [[\mathit{book}]] & < & \textsc{Phy} \bullet \textsc{Info} \\ [[\mathit{pick\ up}]] & : & [[\mathit{human}]] \to \textsc{Phy} \to \mathit{Prop} \\ [[\mathit{master}]] & : & [[\mathit{human}]] \to \textsc{Info} \to \mathit{Prop} \end{array}$$

Because of the above subtyping relationship (and contravariance of subtyping for the function types), we have

$$\begin{array}{rcl} [[\mathit{pick\ up}]] & : & [[\mathit{human}]] \to \textsc{Phy} \to \mathit{Prop} \\ & < & [[\mathit{human}]] \to \textsc{Phy} \bullet \textsc{Info} \to \mathit{Prop} \\ & < & [[\mathit{human}]] \to [[\mathit{book}]] \to \mathit{Prop} \end{array}$$

$$\begin{array}{rcl} [[\mathit{master}]] & : & [[\mathit{human}]] \to \textsc{Info} \to \mathit{Prop} \\ & < & [[\mathit{human}]] \to \textsc{Phy} \bullet \textsc{Info} \to \mathit{Prop} \\ & < & [[\mathit{human}]] \to [[\mathit{book}]] \to \mathit{Prop} \end{array}$$

Therefore, $[[\mathit{pick\ up}]]$ and $[[\mathit{master}]]$ can both be used in a context where terms of type $[[\mathit{human}]] \to [[\mathit{book}]] \to \mathit{Prop}$ are required and the interpretation of the sentence (6.2) can proceed as intended.

However, as we mentioned above, there are some difficulties if we do the same thing in Montagovian settings. Take the example of “heavy book”. In Montague semantics, we should have

$$\begin{array}{rcl} [[\mathit{heavy}]] & : & (\textsc{Phy} \to t) \to (\textsc{Phy} \to t) \\ [[\mathit{book}]] & : & \textsc{Phy} \bullet \textsc{Info} \to t \end{array}$$

In order to interpret “heavy book” as [[heavy]]([[book]]), we need

$$\textsc{Phy} \bullet \textsc{Info} \to t < \textsc{Phy} \to t$$

By contravariance, we need

$$\textsc{Phy} < \textsc{Phy} \bullet \textsc{Info}$$

But this is not the case, the subtype relation is actually in another way around.

6.2.2 Dot-types in Type Theory: a Formal Formulation

In the following, we present a type-theoretic treatment of dot-types with the help of coercive subtyping. There are two important ingredients in this type-theoretic definition:

1. The constituent types of a dot-type should not share common components.
2. A dot-type, if well-formed, should be a subtype of both of its constituent types.

Because of (1), the first and the most important thing is to define the notion of component and, when doing this, because of (2), the set of components of a type should contain its constituents and their super-types. This is formally given by means of the following definition.

Definition 6.1 (component)**.** *Let T :* **Type** *be a type in the empty context. Then, $\mathscr{C}(T)$, the set of components of T, is defined as:*

$$\mathscr{C}(T) =_{def} \begin{cases} SUP(T) & \textit{if the normal form of } T \textit{ is not of the form } X \bullet Y \\ \mathscr{C}(T_1) \cup \mathscr{C}(T_2) & \textit{if the normal form of } T \textit{ is } T_1 \bullet T_2 \end{cases}$$

where $SUP(T) = \{T' | T \leqslant T'\}$.

Now, we give the formal rules for the dot-types in Figure 6.1. Note that, in the formation rule, we require that the constituent types do not share

Formation Rule

$$\frac{\Gamma \vdash valid \quad \langle\rangle \vdash A\colon \mathbf{Type} \quad \langle\rangle \vdash B\colon \mathbf{Type} \quad \mathscr{C}(A) \cap \mathscr{C}(B) = \emptyset}{\Gamma \vdash A \bullet B\colon \mathbf{Type}}$$

Introduction Rule

$$\frac{\Gamma \vdash a\colon A \quad \Gamma \vdash b\colon B \quad \Gamma \vdash A \bullet B\colon \mathbf{Type}}{\Gamma \vdash \langle a, b\rangle\colon A \bullet B}$$

Elimination Rules

$$\frac{\Gamma \vdash c\colon A \bullet B}{\Gamma \vdash p_1(c)\colon A} \qquad \frac{\Gamma \vdash c\colon A \bullet B}{\Gamma \vdash p_2(c)\colon B}$$

Computation Rules

$$\frac{\Gamma \vdash a\colon A \quad \Gamma \vdash b\colon B \quad \Gamma \vdash A \bullet B\colon \mathbf{Type}}{\Gamma \vdash p_1(\langle a, b\rangle) = a\colon A} \qquad \frac{\Gamma \vdash a\colon A \quad \Gamma \vdash b\colon B \quad \Gamma \vdash A \bullet B\colon \mathbf{Type}}{\Gamma \vdash p_2(\langle a, b\rangle) = b\colon B}$$

Projections as Coercions

$$\frac{\Gamma \vdash A \bullet B\colon \mathbf{Type}}{\Gamma \vdash A \bullet B <_{p_1} A\colon \mathbf{Type}} \qquad \frac{\Gamma \vdash A \bullet B\colon \mathbf{Type}}{\Gamma \vdash A \bullet B <_{p_2} B\colon \mathbf{Type}}$$

Coercion Propagation

$$\frac{\Gamma \vdash A \bullet B\colon \mathbf{Type} \quad \Gamma \vdash A' \bullet B'\colon \mathbf{Type} \quad \Gamma \vdash A <_{c_1} A'\colon \mathbf{Type} \quad \Gamma \vdash B = B'\colon \mathbf{Type}}{\Gamma \vdash A \bullet B <_{d_1[c_1]} A' \bullet B'\colon \mathbf{Type}}$$

where $d_1[c_1] \equiv [x : A * B]\langle c_1(p_1(x)), p_2(x)\rangle$

$$\frac{\Gamma \vdash A \bullet B\colon \mathbf{Type} \quad \Gamma \vdash A' \bullet B'\colon \mathbf{Type} \quad \Gamma \vdash A = A'\colon \mathbf{Type} \quad \Gamma \vdash B <_{c_2} B'\colon \mathbf{Type}}{\Gamma \vdash A \bullet B <_{d_2[c_2]} A' \bullet B'\colon \mathbf{Type}}$$

where $d_2[c_2] \equiv [x : A * B]\langle p_1(x), c_2(p_2(x))\rangle$

$$\frac{\Gamma \vdash A \bullet B\colon \mathbf{Type} \quad \Gamma \vdash A' \bullet B'\colon \mathbf{Type} \quad \Gamma \vdash A <_{c_1} A'\colon \mathbf{Type} \quad \Gamma \vdash B <_{c_2} B'\colon \mathbf{Type}}{\Gamma \vdash A \bullet B <_{d[c_1, c_2]} A' \bullet B'\colon \mathbf{Type}}$$

where $d[c_1, c_2] \equiv [x : A * B]\langle c_1(p_1(x)), c_2(p_2(x))\rangle$

Fig. 6.1: The rules of Dot-type

common components:

$$\mathscr{C}(A) \cap \mathscr{C}(B) = \emptyset$$

According to the rules in Figure 6.1, $A \bullet B$ is a subtype of A and a subtype of B. In other words, an object of the dot-type $A \bullet B$ can be regarded as an object of type A, in a context requiring an object of A, and can also be regarded as an object of type B in a context requiring an object of B.

Finally, the subtyping relations are propagated through the dot-types, by means of the coercions d_1, d_2 and d as specified in the last three rules in Figure 6.1.

Propagations

To explain the propagation rules, we first illustrate the coercion relations below:

$$\begin{tikzcd}
 & A \arrow[rr, "\leqslant"] & & A' \\
A \bullet B \arrow[ur, "<"] \arrow[dr, "<"'] \arrow[rr, "<"] & & A' \bullet B' \arrow[ur, "<"] \arrow[dr, "<"'] & \\
 & B \arrow[rr, "\leqslant"'] & & B'
\end{tikzcd}$$

Let's take a more concrete example: we can think of interpreting the phrase

pick up and read the book

Instead of simply considering book having physical and informational aspect, we might think that a book contains *readable* information, compared to *radio program* which does not have a *readable* informational aspect. So we could interpret

$$\begin{aligned}
[[readable]] &: \textsc{Info} \to Prop \\
[[readable\ info]] &= \Sigma(\textsc{Info}, [[readable]]) \\
[[book]] &< \textsc{Phy} \bullet [[readable\ info]]
\end{aligned}$$

With the coercion relation we have for Σ-types in [LL05],

$$\Sigma(\textsc{Info}, [[readable]]) < \textsc{Info}$$

we have $[[readable\ info]] < \textsc{Info}$ and trivially we have $\textsc{Phy} = \textsc{Phy}$. So we could get the following by propagation rule

$$[[book]] < \textsc{Phy} \bullet [[readable\ info]] < \textsc{Phy} \bullet \textsc{Info}$$

It conforms with the example we've explained above.

Since the constituent types of a well-formed dot-type do not share components, it is straightforward to prove the following coherence property.

Proposition 6.2. *(coherence) The coercions p_1, p_2, d_1, d_2 and d are coherent together.*

Note that coherence is important as it guarantees the correctness of employing the projections p_1 and p_2 and the propagation operator d as coercions, and hence the subtyping relationships $A \bullet B <_{p_1} A$ and $A \bullet B <_{p_2} B$.

If the constituent types of a dot-type shared a common component, coherence would fail, like in product type. For instance, A and $A \bullet B$ share the component A. If $A \bullet (A \bullet B)$ were a dot-type, with the transitivity rule there would be the following two coercions p_1 and $p_2 \circ p_1$:

$$A \bullet (A \bullet B) <_{p_1} A$$

$$A \bullet (A \bullet B) <_{p_2 \circ p_1} A$$

which are between the same types but not equal, coherence would then fail.

One may find that, when a dot-type $A \bullet B$ is well-formed, its behavior is similar to that of a product type $A \times B$: intuitively, its objects are pairs and the projections p_1 and p_2 correspond to the projection operations π_1 and π_2 in the product type, respectively. However, there are two important differences between dot-types and product types:

1. The constituent types of a dot-type do not share components, while in a product type the constituent parts can possibly share component. For instance, $A \times A$ is a well-formed product type, but $A \bullet A$ is not a well-formed dot-type.

2. It is fine for both of the projections p_1 and p_2 for dot-types to be coercions (Proposition 6.2), but for product types, only one of them can be coercions for, otherwise, coherence would fail [Luo04].

6.3 Implementation

We have shown how to formalize the dot-types in type theory in the previous section. As we have proof assistants which have implemented various data types, we would also like to put dot-types into proof assistant. However unlike inductive types such as the product types or Σ-types which could be defined with inductive schemata, dot-type cannot simply be defined in such a way. The main reason is that we need to check whether the constituents of the dot-type share components. Especially in our definition of component, we need to check all the coercion relations of the term and its constituents in the context. This is not covered by existing approaches to define inductive types in the libraries of proof assistants. So we have to proceed in a hard code way: defining dot-types directly in a proof assistant.

In this section, we present how we implement dot-types in the proof assistant Plastic, and show how to use it.

6.3.1 Dot-types in Plastic

As explained above, the dot-types have to be directly implemented in Plastic and, at the same time, the associated subtyping relations have to be specified.

First of all, we should point out that, different with the idea in the theory, dot-types do not exist until they are used in our implementation. More precisely, in the theory, for any types A and B, if they do not share components, $A \bullet B$ is a valid dot-type. But in our implementation, it's a valid dot-type only when we have used it and put it into the context. The reason is that we need to check whether the dot-type is valid and add corresponding coercions, but we do not want to check this for every pairs of data type which involves too much unnecessary work. We call a dot-type is *considered* (or *existing*) if we have used it and put it into the context. There are several more things we need to point out here:

- In the syntax of Plastic, we use $A * B$ to present the dot-type $A \bullet B$ and $dot < a, b >$ to present dot term $< a, b >$. However, in the parts that do not concern with code in this thesis, we still use $A \bullet B$ and $< a, b >$ for description.
- When we declare a new dot-type $A \bullet B$, or a dot term $< a, b >$ where $a : A$ and $b : B$, we will first check whether it is a proper dot-type. If $\mathcal{C}(A) \cap \mathcal{C}(B) = \emptyset$, $A \bullet B$ will be a legal dot-type or $< a, b >$ will be a legal dot term; otherwise, they will be rejected and an error message '*dot-type should not share component*' will be shown
- Once a dot-type $A \bullet B$ or dot term $< a, b >$ is considered to be well-formed (legal), we will consider the coercions generated from the dot-type $A \bullet B$. We will add $[x : A \bullet B]p_1(x)$ and $[x : A \bullet B]p_2(x)$ as coercions from $A \bullet B$ to A and B [2].
- We do not offer a special place for the dot-types in the context, we use the information of the coercions. When we want to find out whether a

[2] The system will automatically assign newly metavariable names $cx1$, $cx2$, ... to the new introduced coercions by the dot-type rules

dot-type $A \bullet B$ is considered, we go to the check the coercions in the context to see if $A \bullet B$ is domain of some coercions.

- Furthermore, we will check the existing coercions of other dot-types to see whether there are cases for coercion propagation and if so, Plastic will add the new coercions generated by the coercion propagation into context.

In the implementation, we define some reductions for the computation rules for the projections p_1 and p_2, and the propagation operators d, d_1, and d_2. Assume $A, B, C, D : Type$, $a : A$, $b : B$, $A <_{c_1} C$, $B <_{c_2} D$, we have:

$$\begin{aligned} p_1(< a, b >) &\ \rhd\ a \\ p_2(< a, b >) &\ \rhd\ b \\ d[c_1, c_2](< a, b >) &\ \rhd\ < c_1(a), c_2(b) > \\ d_1[c_1](< a, b >) &\ \rhd\ < c_1(a), b > \\ d_2[c_2](< a, b >) &\ \rhd\ < a, c_2(b) > \end{aligned}$$

In the following, we present four main algorithms in our implementation. First we need to give an algorithm to calculate the components of a type.

Algorithm 6.3. *(checking components) Given a type A, we will calculate the component of A, $\mathscr{C}(A)$, in the following way.*

1. *Check the form of A to see whether it is a dot-type or not.*
2. *If A is not a dot-type, check all the coercion relations in the context to find out every type T which satisfies $A <_c T$ with some coercion c. $\mathscr{C}(A)$ is the set of all these super types T.*
3. *If A is a dot-type of the form $T_1 \bullet T_2$, $\mathscr{C}(A) = \mathscr{C}(T_1) \cup \mathscr{C}(T_2)$. (The algorithm is called recursively.)*

The second algorithm deals with the introduction of dot-types.

Algorithm 6.4. *When defining a type to be a dot-type $A \bullet B$:*

1. *Check the context to see whether A and B are already defined types. If so, go to the next step; otherwise alert that the type is not defined and end the algorithm.*
2. *Calculate $\mathscr{C}(A)$ and $\mathscr{C}(B)$ to see whether the intersection of these two is empty. If so go to the next step; if not, alert that the constituent types share component and end the algorithm.*
3. *Check the existing coercions, to see whether $A \bullet B$ has already been considered. If so, simply finish the algorithm; otherwise go to the next step.*
4. *Add coercion from $A \bullet B$ to A and from $A \bullet B$ to B into the context and add the coercions generated by transitivity as well.*
5. *Check the existing coercions of dot-types in the context to add coercions introduced by propagation rules. For every existing dot-type $C \bullet D$:*
   - *if there's a coercion c_1 from A to C, and a coercion c_2 from B to D, add a new coercion $[x : A \bullet B]d[c_1, c_2](x)$ from $A \bullet B$ to $C \bullet D$.*
   - *if there's a coercion c_1 from A to C, and B is convertible to D, add a new coercion $[x : A \bullet B]d_1[c_1](x)$ from $A \bullet B$ to $C \bullet D$.*
   - *if there's a coercion c_2 from B to D, and A is convertible to C, add a new coercion $[x : A \bullet B]d_2[c_2](x)$*
   - *otherwise, do nothing.*
6. *Check the transitivity possibilities of the newly generated coercion.*

The third algorithm deals with the introduction of dot-terms (similar to that for dot-type introduction).

Algorithm 6.5. *When defining a term to be a dot term $< a, b >$:*

1. *Check the context to see whether a and b are defined terms. If so take the types of a and b, let say A and B, and go to the next step; otherwise alert the term is not defined and end the algorithm.*

2. *Calculate $\mathscr{C}(A)$ and $\mathscr{C}(B)$ to see whether the intersection of these two is empty. If so go to the next step; if not, alert that the constituent types share component and end the algorithm.*

3. *Check the existing coercions, to see whether $A \bullet B$ has already been considered. If so, simply finish the algorithm; otherwise go to the next step.*

4. *Add coercion from $A \bullet B$ to A and from $A \bullet B$ to B into the context, add the coercions generated by transitivity as well.*

5. *Check the existing coercions of dot-types in the context. For every existing dot-type $C \bullet D$:*

   - *if there's a coercion c_1 from A to C, and a coercion c_2 from B to D, add a new coercion $[x : A \bullet B]d[c_1, c_2](x)$ from $A \bullet B$ to $C \bullet D$.*
   - *if there's a coercion c_1 from A to C, and B is convertible to D, add a new coercion $[x : A \bullet B]d_1[c_1](x)$ from $A \bullet B$ to $C \bullet D$.*
   - *if there's a coercion c_2 from B to D, and A is convertible to C, add a new coercion $[x : A \bullet B]d_2[c_2](x)$*
   - *otherwise, do nothing.*

6. *Check the transitivity possibilities of the newly generated coercion.*

Another part we should take care of is that, since we need to consider the propagation rules of dot-types, when we introduce a new coercion, it links two existing dot-types and generates a new coercion through the propagation rules. So when we introduce a new coercion, we should also check all the dot-types in the context to see whether there're types that satisfy the conditions of propagation rule. If so, we need to add a new coercion for the propagation rule as well.

Algorithm 6.6. *If $A_1 <_c A_2$ is a newly declared coercion or a newly generated coercion by transitivity rule:*

1. *look up the context, find all type pairs (A_1, B_1) and (A_2, B_2), such that $A_1 \bullet B_1$ and $A_2 \bullet B_2$ are existing dot-types:*
   - *if B_1 is convertible to B_2, add coercion d1[c] from $A_1 \bullet B_1$ to $A_2 \bullet B_2$*
   - *if B_1 is coercible to B_2 and c_1 is the coercion from B_1 to B_2, add the coercion $d[c, c_1]$ from $A_1 \bullet B_1$ to $A_2 \bullet B_2$.*
2. *look up the context, find all type pairs (C_1, A_1) and (C_2, A_2), such that $C_1 \bullet A_1$ and $C_2 \bullet A_2$ are existing dot-types:*
   - *if C_1 is convertible to C_2, add coercion d2[c] from $C_1 \bullet A_1$ to $C_2 \bullet A_2$*
   - *if C_1 is coercible to C_2 and c_2 is the coercion from C_1 to C_2, add the coercion $d[c_2, c]$ from $C_1 \bullet A_1$ to $C_2 \bullet A_2$.*
3. *Check the transitivity possibilities of the new generated coercion.*

6.3.2 Examples of Dot-types in Plastic

In this subsection, we will first give some abstract examples to show how to declare a dot-type in Plastic, what we will get from the declaration, and some examples of illegal declaration of dot-types. Then we will give a concrete example to interpret sentences in natural language in Plastic.

Example 6.7. *We can define a dot-type or a dot-term simply in the following way:*

1. *If we have two types A, B which do not share components, we could simply define a type M of type $A * B$ like this:*

   ```
   > [M = A*B];
   ```

   The system will generate two coercions $cx1$ from $A * B$ to A and $cx2$ from $A * B$ to B.

2. *We can also define a dot term . If we have two terms a, b, $a : A$ and $b : B$, we can define a dot term $m =< a, b >$ like this:*

```
> [m = dot<a,b>];
```

Now m is defined to be a dot term dot $< a, b >$ and it is of type $A * B$. The system will generate two coercions $cx1$ from $A * B$ to A and $cx2$ from $A * B$ to B.

Example 6.8. *In the following examples, the types share components in different ways, none of them could be defined as a dot-type or dot term, alert will be shown in all the following cases.*

1. *The two parts are the same*

```
> [M = A*A];
```

2. *$A * C$ and $A * B$ have the same component A*

```
> [M = (A*C)*(A*B)];
```

3. *A is a subtype of B, as definition of component $A \in \mathscr{C}(A) \cap \mathscr{C}(B)$*

```
> [c:A->B];
> Coercion = c;
> [M = A*B];
```

4. *a and b are both of type A, but $A * A$ is not a legal dot-type, so dot $< a, b >$ is not a legal dot term.*

```
> [a,b:A];
> [ab = dot<a,b>];
```

5. *a is of type A and b is of type B, while A is a subtype of B. As shown above, $A * B$ is not a legal dot-type, hence dot $< a, b >$ is not a legal dot term.*

```
> [a:A];
> [b:B];
> [c:A->B];
> Coercion = c;
> [ab = dot<a,b>];
```

Example 6.9. *When we have dot-type $A * B$, $A <_{c1} C$ and $B <_{c2} D$, if we claim $C * D$ to be another dot-type, coercions from the propagation rule will also be added.*

```
> [c1:A->C];
> [c2:B->D];
> Coercion = c1;
> Coercion = c2;
> [M1 = A*B];
> [M2 = C*D];
```

In this example several coercions will be added according to the dot-type rule and transitivity. $cx1$ from $A*B$ to A, $cx2$ from $A*B$ to B, $< c1, cx1 >= [x : (A * B)]cx3(cx1(x))$ by transitivity from $A * B$ to C, $< c2, cx1 >= [x : (A * B)]cx4(cx1(x))$ by transitivity from $A * B$ to D, $cx3$ from $C * D$ to C and $cx4$ from $C * D$ to D. However, we will get one more coercion from the propagation rule, there will be a coercion $cx5$ from $A * B$ to $C * D$ and $cx5 = [x : A * B]d[c1, c2](x)$, where for any dot term dot $< a, b >$ of dot-type $A * B$, $d[c1, c2]dot < a, b >= dot < c1(a), c2(b) >$.

Now, let's use a concrete example to show how we could interpret natural language in Plastic:

Example 6.10. *Let's consider the sentence*

John picked up and mastered a book

We should contain the following data:

```
> [PhyInfo = Phy*Info];
> [cb : Book -> PhyInfo];
> Coercion = cb;
> [John:Human];
> [b:Book];
```

Note that 'b' is an arbitrary object of type Book. The verbs 'pick up' and 'master' are of the following types, where "==>" is Plastic notation for function type :

```
> [pickup : Human ==> (Phy ==> Prop)];
> [master : Human ==> (Info ==> Prop)];
```

With the above, we could interpret the sentences "John picked up a book" and "John mastered a book" separately and then use the predefined connective and: $Prop \to Prop \to Prop$ to connect them. However, this would not satisfy the original sentence since the book picked up and that mastered must be the same book.

We want 'and' to connect 'picked up' with 'mastered', so we consider 'and' as a generic semantic kind: for any type A, [[AND]](A) is of kind $A \to A \to A$. For A being $Human ==> (Book ==> Prop)$ [3],

$$And = [[AND]](Human ==> (Book ==> Prop)).$$

In particular, the term "And pickup master" is well-typed, thanks to the coercive subtyping relations and the contravariance in subtyping function types as explained in section 6.1. Now, interpreting the indefinite article by means of the existential quantifier, the above sentence is interpreted as

[3] An alternative possibility is letting A be $Human ==> (PhyInfo ==> Prop)$

(in a readable notation)

$$\exists b\colon Book.\ And(pickup, master)\ John\ b.$$

The full code is followed:

```
> import Sol_All;
> import FnCoercion;
> import SigmaCoercion;
> [Human, Phy, Info, Book :Type];
> [PhyInfo = Phy*Info];
> [cb:Book -> PhyInfo];
> Coercion = cb;
> [pickup:Human ==> (Phy ==> Prop)];
> [master:Human ==> (Info ==> Prop)];
> [John:Human];
> [b:Book];

("John picked up b and John mastered b")

> [pickup_b = ap_ Book Prop (ap_ Human (Book==>Prop)  pickup John) b];
> [master_b = ap_ Book Prop (ap_ Human (Book==>Prop)  master John) b];
> [sentence1 = and pickup_b master_b];

("John picked up and mastered b")

> [AND: (A:Type) (A->A->A)]
> [And = AND (Human ==> (Book ==> Prop))];
> [sentence2 = ap_ Book Prop (ap_ Human (Book==>Prop)
                        (And pickup master) John) b];

("John picked up and mastered a book")
```

```
> [sentence3 = Ex Book [b:Book](ap_ Book Prop (ap_ Human (Book==>Prop)
                       (And pickup master) John) b)];
```

7. CONCLUSION AND FUTURE WORK

7.1 Summary

In this thesis, we have discussed an interesting issue of type theory – subtyping. We have mainly focused on coercive subtyping which is a simple but powerful abbreviation mechanism for subtype relation. We have pointed out the defect of a previous formulation and study of this issue in [Luo99, SL02] and given a better description and understanding of it.

First, we have given the original description of coercive subtyping by Luo in [Luo99]. Using a counter example, we have shown that it was not a conservative extension and the extended system might even be inconsistent in some special cases. Then we have proposed an adequate formulation $T[\mathcal{C}]$ together with an equivalent middle system $T[\mathcal{C}]^*$ with "star-calculus". We have given algorithms for the implicit coercion insertions and proved that for a Martin-Löf's type system T the system $T[\mathcal{C}]^*$ is a conservative extension.

Next, we've shown the implementation of coercive subtyping in proof assistant Plastic, and our improvement of it. At last, we've presented a special kind of data type call "dot-types", which contains two distinguish aspect. We've shown the formalization of this type in type theory with coercive subtyping, and the implementation of it in Plastic. Different with normal data types, dot-types cannot be simply defined in library of a proof assistant, we embed it directly inside the hard code of Plastic.

7.2 Discussion and Future Work

We still have some interesting problems that we should discuss and work on with in the future:

1. One may find that the proof of the totality of the transformations is very tedious. The main reason is that when we transform a derivation, different branches might be inserted with equal but not syntactically equal coercions. We have to use our presupposition lemmas to prove they are judgementally equal to built a transformation. Like in

$$\frac{\Gamma \overset{d_1}{\vdash} K = K' \quad \Gamma \overset{d_2}{\vdash} K' = K''}{\Gamma \vdash K = K''}$$

Under the transformation Θ, d_1 and d_2 become derivations $\Theta(d_1)$ and $\Theta(d_2)$ of, say, $\Gamma_1 \overset{\Theta(d_1)}{\vdash} K_1 = K_1'$ and $\Gamma_2 \overset{\Theta(d_2)}{\vdash} K_2' = K_2''$. We need to show that the corresponding kinds of contexts Γ_1, Γ_2 are equal in T.

But, if we add one more condition on the system and make some change on the formulation, the things might become easier. Suppose T is system with *normalization property*, and we change the coercion application rule (CA1) to be:

$$\frac{\Gamma \vdash f : (x : K)K' \qquad \Gamma \vdash k_0 : K_0 \qquad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) : [nf(c)(k_0)/x]K'}$$

where $nf(c)$ means the normal form of coercion c ((CA2) and (CD) could be modified accordingly).

Under this formulation, the gaps of the same type should be inserted with the syntactically same coercion. For the rule above, Γ_1 and Γ_2 should be syntactically equal, K_1' and K_2' should be syntactically equal as well. The whole definition of Θ will be much simpler.

Actually, this is like what we have done in the implementation. Although we allow equal coercions, we only use the first one in the context when we need to insert them. It does not have to be the normal form, but still guarantees the insertion to be syntactically unique. And, as the improvement we've presented in 5.3.4, we consider the rest convertible coercions to be redundant and will not add them into the context any more.

However, we have shown that without the normalization condition, we can still prove the theorems. When we consider to add this new restriction, what side effect will it bring us? It's not clear yet.

2. As we have discussed in 3.4.2 and 4.3.1, the reason that the original formulation for type system with coercive subtype $T[\mathcal{R}]$ is problematic, is that, a set of coercive rules $\mathcal{R}$ is too general. Since we want $\mathcal{R}$ to be widely open, some unfriendly rule might get involved. The same problem comes out if we say T is any type theory specified in LF. Hence, we restrict our formulation to use $\mathcal{C}$ – a set of coercive judgements – instead of a set of rules $\mathcal{R}$, and we've only proved the theorems for Martin-löf's type theory and UTT. However, we are wondering whether the restriction is too tight. We still hope we can use a set of rules $\mathcal{R}$ for the basement of coercions. We have mentioned that we can use $\mathcal{R}$ to generate a coherent set of judgements $\mathcal{C}_{\mathcal{R}}$, this is one possible solution. However we would like to see whether we can use $\mathcal{R}$ as a base directly and find out what kind of such set could make us a proper extension.

3. Since we have extended system T with coercive subtyping into system $T[\mathcal{C}]$, we would also like to think about the meta-theoretical properties. To which extent the meta-theoretical properties are extended to the new system? Whether normalization and confluence are still hold? How can the method we've used (like the transformation Θ, Θ^*) help us in studying such properties? All these questions are yet to be studied.

4. We've considered the coercions globally, but, sometimes, we just want the coercions be taken place only in a certain environment. Hence, we would like to consider the following so called *Contextual Coercive Subtyping* [Luo10, Luo11].

$$\frac{\Gamma \vdash A : \textbf{Type} \quad \Gamma \vdash B : \textbf{Type} \quad \Gamma \vdash c : (A)B}{\Gamma, A <_c B \vdash \textbf{valid}}$$

$$\frac{\Gamma, A <_c B, \Gamma' \ \textbf{valid}}{\Gamma, A <_c B, \Gamma' \vdash A <_c B : \textbf{Type}}$$

Further more, we could consider to put the coercions inside the terms in the following way:

$$\frac{\Gamma, A <_c B \vdash J}{\Gamma \vdash \textbf{coercion}\ A <_c B\ \textbf{in}\ J}$$

J could be any judgement, if $J \equiv t : T$, **coercion** should distribute through J, **coercion** $A <_c B$ **in** $(t : T) =$ (**coercion** $A <_c B$ **in** t) : (**coercion** $A <_c B$ **in** T).

$$\frac{\Gamma, A <_c B \vdash J}{\Gamma \vdash (\textbf{coercion}\ A <_c B\ \textbf{in}\ t) : (\textbf{coercion}\ A <_c B\ \textbf{in}\ T)}$$

The use of these subtyping framework and the meta-properties of them will be interesting.

5. We have discussed difference between type system of type assignment and type system with canonical object. We've also discussed subsumptive subtyping and coercive subtyping. It is clear that subsumptive subtyping is not suitable for type system with canonical object, but we are still wondering whether or to what extent we can deploy coercive subtyping for type systems of type assignment. We believe the study of this will also show us the relationship between these two different views of type theory.

6. Although we have amended the implementation of coercive subtyping, it is still based on the original formulation $T[\mathcal{R}]$. We are aiming to improve the implementation to be based on the adequate system $T[\mathcal{C}]$ using a set of rules $\mathcal{R}$ to get a coherent set of judgements $\mathcal{C}$, but there are some difficulties. One of them is that the $\mathcal{C}$ generated from $\mathcal{R}$ might

be an infinite set even if $\mathcal{R}$ is finite (recall the remark 4.4). How to deal with such problems still need to be studied.

BIBLIOGRAPHY

[AC01] D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theor. Comput. Sci.*, 266(1-2):273–309, 2001.

[Agd08] The agda proof assistant (version 2). http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php, 2008.

[AMS07] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! *PLPV07*, 2007.

[AP05] N. Asher and J. Pustejovsky. Word meaning and commonsense metaphysics, 2005.

[Ash08] N. Asher. A type driven theory of predication with complex types. *Fundamenta Infor.*, 84(2), 2008.

[Ash11] N. Asher. *Lexical Meaning in Context: A Web of Words.* Cambridge University Press, 2011.

[Bac88] R. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In A. Avron et al, editor, *Workshop on General Logic. LFCS Report Series, ECS-LFCS-85-52.* Dept. of ComputerScience, University of Edinburgh, 1988.

[Bai99] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1999.

[Bar91] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[Bar92] H. Barendregt. Lambda calculi with types. *Handbook of Logic in Computer Science*, 1992.

[BF99] G. Barthe and M. J. Frade. Constructor subtyping. In S. Doaitse Swierstra, editor, *Proceedings of Programming Languages and Systems, 8 conf. (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1999.

[BT98] G. Betarte and A. Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. In G. Sambin and G. Smith, editors, *Twenty-five Years of Constructive Type Theory.* Oxford University Press, 1998.

[BvR00] G. Barthe and F. van Raamsdonk. Constructor subtyping in the calculus of inductive constructions. In Jerzy Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures, 3rd International Conference (FOSSACS 00)*, volume 1784 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2000.

[Car88] L. Cardelli. Type-checking dependent types and subtypes. *Lecture Notes in Computer Science*, 306:45–57, 1988.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland Publishing Company, 1958.

[CG92] P. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leqslant}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[CH88] Th. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[CL01] P. Callaghan and Z. Luo. An implementation of lf with coercive subtyping and universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

[Coo07] R. Cooper. Copredication, dynamic generalized quantification and lexical innovation by coercion. *Proceedings of GL2007, the Fourth International Workshop on Generative Approaches to the Lexicon*, 2007.

[Coo11] R. Cooper. Copredication, quantification and frames. *Logical Aspects of Computational Linguistics (LACL'2011). LNAI 6736*, 2011.

[Coq10] The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.3), INRIA*, 2010.

[CPM90] Th. Coquand and Ch. Paulin-Mohring. Inductively defined types. *Lecture Notes in Computer Science*, 417, 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[dB80] N. G. de Bruijn. A survey of project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 141–161. Academic Press, NY, 1980.

[Dyb91] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[Gen35] G. Gentzen. Untersuchugen über das logishe schliessen. *Mathematische Zeitschrift*, 39, 1935.

[Geu09] H. Geuvers. Proof assistants: history, ideas and future. In *Sadhana Journal, Academy Proceedings in Engineering Sciences, Spe-*

cial Issue on Interactive Theorem Proving and Proof Checking, volume 34, Februray 2009.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[HHP87] R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. In *Proceedings of Symposium on Logic in Computer Science 1987*, pages 194–204. IEEE Computer Society, 1987.

[How80] W. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. HindIey, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490. Academic Press, NY, 1980.

[JLS96] A. P. Jones, Z. Luo, and S. Soloviev. Some algorithmic and proof-theoretical aspects of coercive subtyping. In Eduardo Gimnez and Christine Paulin-Mohring, editors, *Proceedings of Types for Proofs and Programs, International Workshop TYPES 96*, volume 1512 of *Lecture Notes in Computer Science*, pages 173–195. Springer, 1996.

[Kle52] S. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[LA08] Z. Luo and R. Adams. Structural subtyping for inductive types with functorial equality rules. *Mathematical Structures in Computer Science*, 18(5):931–972, 2008.

[LL05] Z. Luo and Y. Luo. Transitivity in coercive subtyping. *Information and Computation.*, 197(1-2):122–144, 2005.

[LP92] Z. Luo and R. Pollack. Lego proof development system: User manual, 1992.

[LS99] Z. Luo and S. Soloviev. Dependent coercions. *The 8th Inter. Conf. on Category Theory and Computer Science (CTCS'99), Edinburgh,* *Scotland. Electronic Notes in Theoretical Computer Science*, 29, 1999.

[LSX13] Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: Theory and implementation. *Information and Computation*, 223:18–42, February 2013.

[Luo90] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[Luo97] Z. Luo. Coercive subtyping in type theory. In Dirk van Dalen and Marc Bezem, editors, *Proceedings of the 1996 Annual Conference of the European Association for Computer Science Logic, Utrecht (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 276–296. Springer, 1997.

[Luo99] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[Luo04] Y. Luo. *Cohernce and transitivity in coercive subtyping.* PhD thesis, University of Durham, 2004.

[Luo09a] Z. Luo. Dependent record types revisited. In *Proc. of the 1st Inter. Workshop on Modules and Libraries for Proof Assistants (MLPA'09), Montreal. ACM Inter. Conf. Proceeding Series; Vol. 429.*, 2009.

[Luo09b] Z. Luo. Manifest fields and module mechanisms in intensional type theory. In *Types for Proofs and Programs, TYPES'08. LNCS 5497*, 2009.

[Luo10] Z. Luo. Type-theoretical semantics with coercive subtyping. *Semantics and Linguistic Theory 20 (SALT20), Vancouver*, 2010.

[Luo11] Z. Luo. Contextual analysis of word meanings in type-theoretical semantics. *Logical Aspects of Computational Linguistics (LACL'2011). LNAI 6736*, 2011.

[Mat08] The Matita proof assistant. `http://matita.cs.unibo.it/`, 2008.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer Systems and Sciences*, 17:348–375, 1978.

[Mit91] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[ML75] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H.Rose and J.C.Shepherdson, editors, *Logic Colloquium'73*, 1975.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[MM09] L. Marie-Magdeleine. *Sous-typage coercitif en présence de réductions non-standards dans un système aux types dépendants.* PhD thesis, Université de Toulouse, 2009.

[Mon74] R. Montague. Formal philosophy, 1974.

[NPS90] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf 's Type Theory: An Introduction.* Oxford University Press, Oxford, 1990.

[PM93] C. Paulin-Mohring. Inductive definition in the system Coq: rules and properties. In *Proceedings of Inter. Conf. on Typed Lambda Calculi and Application (TLCA'93), LNCS 664*, 1993.

[Pra73] D. Prawitz. Towards a foundation of a general proof theory. In P. Suppes et al, editor, *Logic, Methodology, and Phylosophy of Science IV.* 1973.

[Pra74] D. Prawitz. *Synthese*, 27, 1974.

[Pro12] Proof general. http://proofgeneral.inf.ad.ac.uk, 2012.

[Pus95] J. Pustejovsky. *The Generative Lexicon.* MIT, 1995.

[Pus05] J. Pustejovsky. A survey of dot objects. Manuscript, 2005.

[Pus11] J. Pustejovsky. Mechanisms of coercion in a general theory of selection, 2011.

[Ran94] A. Ranta. *Type-Theoretical Gramma.* Oxford University Press, Oxford, 1994.

[Rey74] J.C. Reynolds. Towards a theory of type structure. *Lecture Notes in Computer Science*, 19, 1974.

[Saï97] A. Saïbi. Typing algorithm in type theory with inheritance. In *Proceedings of 24th Annual Symposium on Principles of Programming Languages (POPL'97)*, 1997.

[Sco70] D. Scott. Constructive validity. *Symp. on Automatic Demonstration, Lecture Notes in Mathematics 125*, 1970.

[SL02] S. Soloviev and Z. Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 113(1-3):297–322, 2002.

[XL12] T. Xue and Z. Luo. Dot-types and their implementaion. *Logical Aspects of Computational Linguistics (LACL'2012). LNCS*, 7351:234–249, 2012.

APPENDIX

A. ALGORITHMS

Notation A.1. *If d is a derivation of a type system specified in LF, ending with a rule labeled (m) which is of form* $\frac{J_1 \; ... \; J_n}{J}$*, and each judgement* J_k *(k=1,...,n) has a derivation* d_k*, then we write d as* $R_{(m)}(d_1, ..., d_n)$*. For a special case, if d is ending with rule (1.3) in Figure 4.1, we write it as* $R_{(m)}(d_1, \Gamma)$*.*

Algorithm A.2. ***(weakening)****If in* $T[\mathcal{C}]^-$*, d is a derivation of* $\Gamma, \Gamma' \vdash J$*, and* d' *is a derivation of* $\Gamma, \Gamma'' \vdash \textbf{valid}$*, then* $wkn(d, d', \Gamma)$ *is a derivation of* $\Gamma, \Gamma'', \Gamma' \vdash J$*. (J is of form:* $\textbf{valid}$*,* K_0 $\textbf{kind}$*,* $k_0 : K_0$*,* $K_1 = K_2$*,* $k_1 = k_2 : K_0$*, or* $K_1 <_c K_2$*)*

The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

1. *rule(1.1),* $d \equiv \frac{}{<>\vdash \textbf{valid}}$.

   It means that $\Gamma \equiv <>$ *and* $\Gamma' \equiv <>$*, hence*

   $wkn(d, d', <>) \equiv d'$

2. *rule(1.2),* $d \equiv \frac{\Gamma_1 \overset{d_1}{\vdash} K \;\; \textbf{kind}}{\Gamma_1, x : K \vdash \textbf{valid}}$

   It means that $\Gamma, \Gamma' \equiv \Gamma_1, x : K$

   (a) $\Gamma' \equiv <>$*, simply*

   $wkn(d, d', \Gamma) \equiv d'$

   (b) *otherwise,* $\Gamma' \equiv (\Gamma_2, x : K)$ *for a* Γ_2*, hence* $\Gamma, \Gamma_2 \equiv \Gamma_1$*,*

   $wkn(d, d', \Gamma) \equiv R_{(1.2)}(wkn(d_1, d', \Gamma))$

3. *rule(SK1),* $d \equiv \frac{\Gamma, \Gamma' \overset{d_1}{\vdash} A <_c B : \textbf{Type}}{\Gamma, \Gamma' \vdash El(A) <_c El(B)}$,

   $wkn(d, d') \equiv R_{(SK1)}(R_{(ST7)}(d_1, d', \Gamma))$

4. $d \equiv R(d_1, \; ... \; , d_n)$*, where R is any other rule applies to the case, and* $d_1, \; ... \; , d_n$ *are the premises of rule R,*

   $wkn(d_1, d', \Gamma) = R(wkn(d_1, d', \Gamma), \; ... \; , wkn(d_n, d', \Gamma))$

Algorithm A.3. ***(presupposition algorithms 1)*** *There exist algorithms pre_1, pre_2, pre_3, from derivations of $T[C]^-$ to derivations of $T[C]^-$ that satisfy the following properties.*

1. *If d is a derivation of $\Gamma, \Gamma' \vdash J$, then $pre_1(d, \Gamma)$ is a derivation of $\Gamma \vdash$ **valid** (J is of form:* **valid**, K_0 **kind**, $k_0 : K_0$, $K_1 = K_2$, *or* $k_1 = k_2 : K_0$*);*

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(1.1),* $d \equiv \dfrac{}{<>\vdash \textbf{valid}}$.
   $pre_1(d, \Gamma_1) \equiv d$

   (b) *rule(1.2),* $d \equiv \dfrac{\Gamma_1 \overset{d_1}{\vdash} K \ \textbf{kind}}{\Gamma_1, x : K \vdash \textbf{valid}}(x \notin FV(\Gamma_1))$.
   It means that $\Gamma, \Gamma' \equiv \Gamma_1, x : K$

      i. $\Gamma' \equiv <>$, *hence* $\Gamma \equiv \Gamma_1, x : K$
      $pre_1(d, \Gamma) \equiv d$

      ii. $\Gamma' \not\equiv <>$
      $pre_1(d, \Gamma) \equiv pre_1(d_1, \Gamma)$

   (c) $d \equiv R(d_1, \ \dots \ , d_n)$*, where R is any other rule applies to the case (including rule CA1, CA2 and CD), and* $d_1, \ \dots \ , d_n$ *are the premises of rule R,*
   $pre_1(d, \Gamma) = pre_1(d_1, \Gamma)$

2. *If d is a derivation of* $\Gamma, x : K, \Gamma' \vdash J$*, then* $pre_2(d, \Gamma)$ *is a derivation of* $\Gamma \vdash K$ **kind** *(J is of form:* **valid**, K_0 **kind**, $k_0 : K_0$, $K_1 = K_2$, *or* $k_1 = k_2 : K_0$*)*

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(1.1),* $d \equiv \dfrac{}{<>\vdash \textbf{valid}}$. *It trivially doesn't apply the case.*

   (b) *rule(1.2),* $d \equiv \dfrac{\Gamma_1 \overset{d_1}{\vdash} K_0 \ \textbf{kind}}{\Gamma_1, y : K_0 \vdash \textbf{valid}}(x \notin FV(\Gamma_1))$.
   It means that $\Gamma, x : K, \Gamma' \equiv \Gamma_1, y : K_0$

      i. $\Gamma' \equiv <>$, *hence* $\Gamma \equiv (\Gamma_1, y : K_0, x : K)$
      $pre_2(d, \Gamma) \equiv d_1$

      ii. $\Gamma' \not\equiv <>$,
      $pre_2(d, \Gamma) \equiv pre_2(d_1, \Gamma)$

(c) $d \equiv R(d_1, \ ... \ , d_n)$, *where R is any other rule applies to the case (including rule CA1, CA1 and CD), and* $d_1, \ ... \ , d_n$ *are the premises of rule R,*
$pre_2(d, \Gamma) = pre_2(d_1, \Gamma)$

3. *If d is a derivation of* $\Gamma \vdash (x : K_1)K_2$ **kind**, *then* $pre_3(d)$ *is a derivation of* $\Gamma, x : K_1 \vdash K_2$ **kind**.

rule(6.1) is the only possible rule according to the syntax of the judgement.

$$d \equiv \frac{\Gamma \overset{d_1}{\vdash} K \ \textbf{kind} \quad \Gamma, x : K \overset{d_2}{\vdash} K' \ \textbf{kind}}{\Gamma \vdash (x : K)K' \ \textbf{kind}}$$

$pre_3(d) \equiv d_2$

Algorithm A.4. ***(substitutions)*** *There are following algorithms in* $T[\mathcal{C}]^-$, *constructed simultaneously:*

1. *If in d is a derivation of* $\Gamma, x : K, \Gamma' \vdash$ **valid**, d' *is a derivation of* $\Gamma \vdash k : K$, *then* $sub_1(d, d')$ *is a derivation of* $\Gamma, [k/x]\Gamma' \vdash$ **valid**.

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(1.1)*, $d \equiv \dfrac{}{<>\vdash \textbf{valid}}$,
   $sub_1(d, d') \equiv d$

   (b) *rule(1.2)*, $d \equiv \dfrac{\Gamma_1 \overset{d_1}{\vdash} K_1 \ \textbf{kind}}{\Gamma_1, y : K_1 \vdash \textbf{valid}}(y \notin FV(\Gamma_1))$
   It means that $(\Gamma, x : K, \Gamma') \equiv (\Gamma_1, y : K_1)$

      i. *if* $\Gamma' \equiv <>$, *hence* $\Gamma \equiv \Gamma_1$ *and* $x : K \equiv y : K_1$,
      $sub_1(d, d') \equiv pre_1(d, \Gamma)$

      ii. *otherwise,* $\Gamma' \equiv (\Gamma'', y : K_1)$ *for some* Γ'',
      $sub_1(d, d') \equiv R_{(1.2)}(sub_2(d_1, d'))$

2. *If d is a derivation of* $\Gamma, x : K, \Gamma' \vdash K'$ **kind**, d' *is a derivation of* $\Gamma \vdash k : K$, *then* $sub_2(d, d')$ *is a derivation of* $\Gamma, [k/x]\Gamma' \vdash [k/x]K'$ **kind**.

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(4.1)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} \textbf{valid}}{\Gamma, x : K, \Gamma' \vdash \textbf{Type kind}}$,
   $sub_2(d, d') \equiv R_{(4.1)}(sub_1(d_1, d'))$

(b) *rule(4.2)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} A : \mathbf{Type}}{\Gamma, x : K, \Gamma' \vdash El(A)\ \mathbf{kind}}$,
$sub_2(d, d') \equiv R_{(4.2)}(sub_3(d_1, d'))$

(c) *rule(6.1)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} K_1\ \mathbf{kind} \quad \Gamma, x : K, \Gamma', y : K_1 \overset{d_2}{\vdash} K_2\ \mathbf{kind}}{\Gamma, x : K, \Gamma' \vdash (y : K_1)K_2\ \mathbf{kind}}$,
$sub_2(d, d') \equiv R_{(6.1)}(sub_2(d_1, d'), sub_2(d_2, d'))$

3. *If* d *is a derivation of* $\Gamma, x : K, \Gamma' \vdash k' : K'$, d' *is a derivation of* $\Gamma \vdash k : K$, *then* $sub_3(d, d')$ *is a derivation of* $\Gamma, [k/x]\Gamma' \vdash [k/x]k' : [k/x]K'$.

The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

(a) *rule(1.3)*, $d \equiv \dfrac{\Gamma_1, y : K_1, \Gamma_2 \overset{d_1}{\vdash} \mathbf{valid}}{\Gamma_1, y : K_1, \Gamma_2 \vdash y : K_1}$
It means that $(\Gamma, x : K, \Gamma' \vdash k' : K') \equiv (\Gamma_1, y : K_1, \Gamma_2 \vdash y : K_1)$

i. $x : K$ *is in* Γ_2, *so there is a* Γ_3, *such that* $\Gamma \equiv (\Gamma_1, y : K_1, \Gamma_3)$, *and* x *is not free in* K_1,
$sub_3(d, d') \equiv R_{(1.3)}(sub_1(d_1, d'), \Gamma_1)$

ii. $x : K$ *is in* Γ_1, *so there is a* Γ_4, *such that* $(\Gamma, x : K, \Gamma_4) \equiv \Gamma_1$, *hence* $\Gamma' \equiv (\Gamma_4, y : K_1, \Gamma_2)$,
$sub_3(d, d') \equiv R_{(1.3)}(sub_1(d_1, d'), (\Gamma, [k/x]\Gamma_4))$

iii. $x : K$ *is exactly* $y : K_1$, *so* $\Gamma \equiv \Gamma_1$, $\Gamma' \equiv \Gamma_2$. *Hence* $sub_1(d_1, d')$ *is a derivation of* $\Gamma, [k/x]\Gamma' \vdash \mathbf{valid}$,
$sub_3(d, d') \equiv wkn(d', sub_1(d_1, d'), \Gamma)$

(b) *rule(3.1)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} k' : K'' \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} K'' = K'}{\Gamma, x : K, \Gamma' \vdash k' : K'}$,
$sub_3(d, d') \equiv R_{(3.1)}(sub_3(d_1, d'), sub_4(d_2, d'))$

(c) *rule(6.3)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma', y : K_1 \overset{d_1}{\vdash} k_1 : K_1'}{\Gamma, x : K, \Gamma' \vdash [y : K_1]k_1 : (y : K_1)K_1'}$
It means that $k' : K' \equiv [y : K_1]k_1 : (y : K_1)K_1'$,
$sub_3(d, d') \equiv R_{(6.3)}(sub_3(d_1, d'))$

(d) *rule(6.5)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} f : (y : K_1)K_1' \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} k_1 : K_1}{\Gamma, x : K, \Gamma' \vdash f(k_1) : [k_1/y]K_1'}$
It means that $k' : K' \equiv f(k_1) : [k_1/y]K'$,
$sub_3(d, d') \equiv R_{(6.5)}(sub_3(d_1, d'), sub_3(d_2, d'))$

(e) *rule(CA1)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} f : (y : K_1)K' \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} k_0 : K_0 \quad \Gamma, x : K, \Gamma' \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma, x : K, \Gamma' \vdash f(k_0) : [c(k_0)/y]K'}$,
$sub_3(d, d') \equiv R_{(CA1)}(sub_3(d_1, d'), sub_3(d_2, d'), sub_K(d_3, d'))$

4. *If d is a derivation of $\Gamma, x : K, \Gamma' \vdash K' = K''$, d' is a derivation of $\Gamma \vdash k : K$, then $sub_4(d, d')$ is a derivation of $\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k/x]K''$.*

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(2.1),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} K' \ \textbf{kind}}{\Gamma, x : K, \Gamma' \vdash K' = K'}$,
   $sub_4(d, d') \equiv R_{(2.1)}(sub_2(d_1, d'))$

   (b) *rule(2.2),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} K'' = K'}{\Gamma, x : K, \Gamma' \vdash K' = K''}$,
   $sub_4(d, d') \equiv R_{(2.2)}(sub_4(d_1, d'))$

   (c) *rule(2.3),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} K' = K''' \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} K''' = K'}{\Gamma, x : K, \Gamma' \vdash K' = K''}$,
   $sub_4(d, d') \equiv R_{(2.3)}(sub_3(d_1, d'), sub_4(d_2, d')$

   (d) *rule(4.3),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} A = B : \textbf{Type}}{\Gamma, x : K, \Gamma' \vdash El(A) = El(B)}$,
   $sub_5(d, d') \equiv R_{(4.3)}(sub_5(d_1, d'))$

   (e) *rule(6.2),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} K_1 = K_2 \quad \Gamma, x : K, \Gamma', y : K_1 \overset{d_2}{\vdash} K_1' = K_2'}{\Gamma, x : K, \Gamma' \vdash (y : K_1)K_1' = (y : K_2)K_2'}$,
   $sub_4(d, d') \equiv R_{(6.2)}(sub_4(d_1, d'), sub_4(d_2, d')$

5. *If d is a derivation of $\Gamma, x : K, \Gamma' \vdash k' = k'' : K'$, d' is a derivation of $\Gamma \vdash k : K$, then $sub_5(d, d')$ is a derivation of $\Gamma, [k/x]\Gamma' \vdash [k/x]k' = [k/x]k'' : [k/x]K'$.*

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(2.4),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} k' : K'}{\Gamma, x : K, \Gamma' \vdash k' = k' : K'}$,
   $sub_5(d, d') \equiv R_{(2.4)}(sub_3(d_1, d'))$

   (b) *rule(2.5),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} k'' = k' : K'}{\Gamma, x : K, \Gamma' \vdash k' = k'' : K'}$,
   $sub_5(d, d') \equiv R_{(2.5)}(sub_5(d_1, d'))$

   (c) *rule(2.6),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} k' = k''' : K' \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} k''' = k'' : K'}{\Gamma, x : K, \Gamma' \vdash k' = k'' : K'}$,
   $sub_5(d, d') \equiv R_{(2.6)}(sub_5(d_1, d'), sub_5(d_2, d'))$

   (d) *rule(3.2),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} k' = k'' : K'' \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} K'' = K'}{\Gamma, x : K, \Gamma' \vdash k' = k'' : K'}$,
   $sub_5(d, d') \equiv R_{(3.2)}(sub_5(d_1, d'), sub_4(d_2, d'))$

(e) *rule(6.4),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K_1 = K_2 \quad \Gamma, x:K, \Gamma', y:K_1 \overset{d_2}{\vdash} k_1 = k_2 : K_3}{\Gamma, x:K, \Gamma' \vdash [y:K_1]k_1 = [y:K_2]k_2 : (y:K_1)K_3}$,
$sub_5(d,d') \equiv R_{(6.4)}(sub_4(d_1,d'), sub_5(d_2,d'))$

(f) *rule(6.6),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} f_1 = f_2 : (y:K_1)K_2 \quad \Gamma, x:K, \Gamma' \overset{d_2}{\vdash} k_1 = k_2 : K_1}{\Gamma, x:K, \Gamma' \vdash f_1(k_1) = f_2(k_2) : [k_1/y]K_2}$,
$sub_5(d,d') \equiv R_{(6.6)}(sub_5(d_1,d'), sub_5(d_2,d'))$

(g) *rule(6.7),* $d \equiv \dfrac{\Gamma, x:K, \Gamma', y:K_1 \overset{d_1}{\vdash} k_2 : K_2 \quad \Gamma, x:K, \Gamma' \overset{d_2}{\vdash} k_1 : K_1}{\Gamma, x:K, \Gamma' \vdash ([y:K_1]k_2)k_1 = [k_1/y]k_2 : [k_1/y]K_2}$,
$sub_5(d,d') \equiv R_{(6.7)}(sub_3(d_1,d'), sub_3(d_2,d')$

(h) *rule(6.8),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} f : (y:K_1)K_2}{\Gamma, x:K, \Gamma' \vdash [y:K_1]f(y) = f : (y:K_1):K_2}$,
$(y \notin FV(\Gamma, x:K, \Gamma'))$
$sub_5(d,d') \equiv R_{(6.8)}(sub_3(d_1,d'))$

(i) *rule(CA2),*

$$d \equiv \frac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} f = f' : (y:K_1)K' \quad \Gamma, x:K, \Gamma' \overset{d_2}{\vdash} k_0 = k_0' : K_0 \quad \Gamma, x:K, \Gamma' \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma, x:K, \Gamma' \vdash f(k_0) = f'(k_0') : [c(k_0)/y]K'}$$

$sub_5(d,d') \equiv R_{(CA2)}(sub_5(d_1,d'), sub_5(d_2,d'), sub_K(d_3,d'))$

(j) *rule(CD),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} f : (y:K_1)K' \quad \Gamma, x:K, \Gamma' \overset{d_2}{\vdash} k_0 : K_0 \quad \Gamma, x:K, \Gamma' \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma, x:K, \Gamma' \vdash f(k_0) = f(c(k_0)) : [c(k_0)/y]K'}$
$sub_5(d,d') \equiv R_{(CD)}(sub_3(d_1,d'), sub_3(d_2,d'), sub_K(d_3,d'))$

6. *If d is a derivation of* $\Gamma, x:K, \Gamma' \vdash K_1 <_c K_2$, d' *is a derivation of* $\Gamma \vdash k : K$, *then* $sub_K(d,d')$ *is a derivation of* $\Gamma, [k/x]\Gamma' \vdash [k/x]K_1 <_{[k/x]c} [k/x]K_2$.

The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

(a) *rule(SK1),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} A <_c B : \textbf{Type}}{\Gamma, x:K, \Gamma' \vdash El(A) <_c El(B)}$,
$sub_K(d,d') \equiv R_{(SK1)}(R_{(ST6)}(d_1,d'))$

(b) *rule(SK2),*

$$d \equiv \frac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K_1' <_{c_1} K_1 \quad \Gamma, x:K, \Gamma', x':K_1' \overset{d_2}{\vdash} [c_1(x')/x]K_2 = K_2' \quad \Gamma, x:K, \Gamma', y:K_1 \overset{d_3}{\vdash} K_2 \ \textbf{kind}}{\Gamma, x:K, \Gamma' \vdash (y:K_1)K_2 <_c (x':K_1')K_2'},$$

where $c \equiv [f:(y:K_1)K_2][x':K_1']f(c_1(x'))$
$sub_K(d,d') \equiv R_{(SK2)}(sub_K(d_1,d'), sub_4(d_2), sub_2(d_3))$

(c) *rule(SK3),*

$$d \equiv \frac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K_1' = K_1 \quad \Gamma, x:K, \Gamma', x':K_1' \overset{d_2}{\vdash} K_2 <_{c_2} K_2' \quad \Gamma, x:K, \Gamma', y:K_1 \overset{d_3}{\vdash} K_2 \ \textbf{kind}}{\Gamma, x:K, \Gamma' \vdash (y:K_1)K_2 <_c (x':K_1')K_2'},$$

where $c \equiv [f:(y:K_1)K_2][x':K_1']c_2 f(x')$

$sub_K(d,d') \equiv R_{(SK3)} sub_4(d_1,d'), sub_K(d_2,d'), sub_2(d_3,d')$

(d) *rule(SK4),*

$$d \equiv \frac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K_1' <_{c_1} K_1 \quad \Gamma, x:K, \Gamma', x':K_1' \overset{d_2}{\vdash} [c_1(x')/x]K_2 <_{c_2} K_2' \quad \Gamma, x:K, \Gamma', y:K_1 \overset{d_3}{\vdash} K_2 \ \mathbf{kind}}{\Gamma, x:K, \Gamma' \vdash (y:K_1)K_2 <_c (x':K_1')K_2'},$$

where $c \equiv [f:(y:K_1)K_2][x':K_1']c_2(f(c_1(x')))$

$sub_K(d,d') \equiv R_{(SK4)}(sub_K(d_1,d'), sub_K(d_2,d'), sub_2(d_3,d'))$

(e) *rule(SK5),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K <_c K' \quad \Gamma, x:K, \Gamma' \overset{d_2}{\vdash} c = c' : (K)K'}{\Gamma, x:K, \Gamma' \vdash K <_{c'} K''},$

$sub_K(d,d') \equiv R_{(SK5)}(sub_K(d_1,d'), sub_5(d_2,d'))$

(f) *rule(SK6),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K_1 <_c K_2 \quad \Gamma, x:K, \Gamma' \overset{d_2}{\vdash} K_1 = K_1'}{\Gamma, x:K, \Gamma' \vdash K_1' <_c K_2},$

$sub_K(d,d') \equiv R_{(SK6)}(sub_K(d_1,d'), sub_4(d_2,d'))$

(g) *rule(SK7),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K_1 <_c K_2 \quad \Gamma, x:K, \Gamma' \overset{d_2}{\vdash} K_2 = K_2'}{\Gamma, x:K, \Gamma' \vdash K_1 <_c K_2'},$

$sub_K(d,d') \equiv R_{(SK7)}(sub_K(d_1,d'), sub_4(d_2,d'))$

(h) *rule(SK8),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K_1 <_{c_1} K_2 \quad \Gamma, x:K, \Gamma' \overset{d_2}{\vdash} K_2 <_{c_2} K_3}{\Gamma, x:K, \Gamma' \vdash K_1 <_{c_2 \circ c_1} K_3'},$

$sub_K(d,d') \equiv R_{(SK8)}(sub_K(d_1,d'), sub_K(d_2,d'))$

Algorithm A.5. ***(weak equality substitutions)*** *The following two algorithms are held in* $T[\mathcal{C}]^-$*:*

1. *If* d *is a derivation of* $\Gamma, x:K, \Gamma' \vdash K'$ **kind**, d' *is a derivation of* $\Gamma \vdash k_1 = k_2 : K$, d'' *is a derivation of* $\Gamma \vdash k_1 : K$, d''' *is a derivation of* $\Gamma \vdash k_2 : K$, *then* $sub_6'(d,d',d'',d''')$ *is a derivation of* $\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]K' = [k_2/x]K'$.

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(4.1),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} \mathbf{valid}}{\Gamma, x:K, \Gamma' \vdash \mathbf{Type\ kind}}$

   $sub_6'(d,d',d'',d''') \equiv R_{(2.1)}(R_{(4.1)}(sub_1(d_1,d'')))$

   (b) *rule(4.2),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} A : \mathbf{Type}}{\Gamma, x:K, \Gamma' \vdash El(A)\ \mathbf{kind}}$

   $sub_6'(d,d',d'',d''') \equiv R_{(4.3)}(sub_7'(d_1,d',d'',d'''))$

   (c) *rule(6.1),* $d \equiv \dfrac{\Gamma, x:K, \Gamma' \overset{d_1}{\vdash} K_1\ \mathbf{kind} \quad \Gamma, x:K, \Gamma', y:K_1 \overset{d_2}{\vdash} K_2\ \mathbf{kind}}{\Gamma, x:K, \Gamma' \vdash (y:K_1)K_2\ \mathbf{kind}}$

   $sub_6'(d,d',d'',d''') \equiv R_{(6.2)}(sub_6'(d_1,d',d'',d'''), sub_6'(d_2,d',d'',d'''))$

2. *If d is a derivation of $\Gamma, x : K, \Gamma' \vdash k' : K'$, d' is a derivation of $\Gamma \vdash k_1 = k_2 : K$, then $sub'_7(d, d', d'', d''')$ is a derivation of $\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]k' = [k_2/x]k' : [k_1/x]K'$.*

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(1.3),* $d \equiv \dfrac{\Gamma_1, y : K_1, \Gamma_2 \overset{d_1}{\vdash} \textbf{valid}}{\Gamma_1, y : K_1, \Gamma_2 \vdash y : K_1}$.
   It means that $\Gamma, x : K, \Gamma' \equiv \Gamma_1, y : K_1, \Gamma'$.

      i. *$x : K$ is in Γ_2, which means that, there's Γ_3 such that $\Gamma \equiv (\Gamma_1, y : K_1, \Gamma_3)$. Hence x is not free in K_1,*
      $sub'_7(d, d', d'', d''') \equiv R_{(2.4)}(sub_3(d_1, d''))$

      ii. *$x : K$ is $y : K_1$, which means that $\Gamma \equiv \Gamma_1$, $\Gamma' \equiv \Gamma_2$ and $x : K \equiv y : K_1$. So $([k_1/x]y = [k_2/x]y : [k_1/x]K_1) \equiv (k_1 = k_2 : K)$, hence,*
      $sub'_7(d, d', d'', d''') \equiv wkn(d', sub_1(d_1, d''), \Gamma)$

      iii. *$x : K$ is in Γ_1, which means that, there's Γ_4 such that $(\Gamma, x : K, \Gamma_4) \equiv \Gamma_1$,*
      $sub'_7(d, d', d'', d''') \equiv R_{(2.4)}(sub_3(d_1, d''))$

   (b) *rule(3.1),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} k' : K'' \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} K'' = K'}{\Gamma, x : K, \Gamma' \vdash k' : K'}$
   $sub'_7(d, d', d'', d''') \equiv R_{(3.1)}(sub'_7(d_1, d', d'', d'''), sub_4(d_2, d'))$

   (c) *rule(6.3),* $d \equiv \dfrac{\Gamma, x : K, \Gamma', y : K_1 \overset{d_1}{\vdash} k_0 : K_0}{\Gamma, x : K, \Gamma' \vdash [y : K_1]k_0 : (y : K_1)K_0}$
   $sub'_7(d, d', d'', d''') \equiv R_{(6.4)}(sub'_6(pre_2(d_1, (\Gamma, x : K, \Gamma')), d', d'', d'''), sub'_7(d_1, d', d'', d'''))$

   (d) *rule(6.5),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} f : (y : K_1)K_2 \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} m : K_1}{\Gamma, x : K, \Gamma' \vdash f(m) : [m/y]K_2}$
   $sub_7(d, d', d'', d''') \equiv R_{(6.6)}(sub'_7(d_1, d', d'', d'''), sub'_7(d_2, d', d'', d'''))$

   (e) *rule(CA1),* $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} f : (y : K_1)K' \quad \Gamma, x : K, \Gamma' \overset{d_2}{\vdash} k_0 : K_0 \quad \Gamma, x : K, \Gamma' \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma, x : K, \Gamma' \vdash f(k_0) : [c(k_0)/y]K'}$
   $sub_7(d, d', d'', d''') \equiv R_{(CA1)}(sub_7(d_1, d', d'', d'''), sub_3(d_2, d', d'', d'''), sub_K(d_3, d'))$

Algorithm A.6. ***(weak context retyping)*** *If d is a derivation of $\Gamma, x : K, \Gamma' \vdash J$, d' is a derivation of $\Gamma \vdash K = K'$, d'' is a derivation of $\Gamma \vdash K$* **kind***, d''' is a derivation of $\Gamma \vdash K'$* **kind***, then we have $ctx'(d, d', d'', d''')$ as a derivation of $\Gamma, x : K', \Gamma' \vdash J$ (J is of form:* **valid**, K_0 **kind**, $k_0 : K_0$, $K_1 = K_2$, $k_1 = k_2 : K_0$, *or* $K_1 <_c K_2$*).*

The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

1. *rule(1.2)*, $d \equiv \dfrac{\Gamma_1 \overset{d_1}{\vdash} K_1 \textbf{ kind}}{\Gamma_1, y : K_1 \vdash \textbf{valid}} (y \notin FV(\Gamma))$

   It means that $(\Gamma, x : K, \Gamma') \equiv (\Gamma_1, y : K_1)$

   (a) $\Gamma' \equiv <>$, *hence* $\Gamma \equiv \Gamma_1$ *and* $x : K \equiv y : K_1$,

   $ctx'(d, d', d'', d''') \equiv R_{(1.2)}(d'')$

   (b) $y : K_1$ *is the tail of* Γ', *there's* Γ'' *such that* $\Gamma' \equiv (\Gamma'', y : K_1)$ *and* $(\Gamma, x : K, \Gamma'') \equiv \Gamma_1$,

   $ctx'(d, d', d'', d''') \equiv R_{(1.2)}(ctx'(d_1, d', d'', d'''))$

2. *rule(SK1)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} A <_c B : \textbf{Type}}{\Gamma, x : K, \Gamma' \vdash El(A) <_c El(B)}$,

   $ctx'(d, d', d'', d''') \equiv R_{(SK1)}(R_{(ST8)}(d_1, d'))$

3. $d \equiv R(d_1, \ \dots \ , d_n)$, *where* R *is any other rule applies to the case, and* $d_1, \ \dots \ , d_n$ *are the premises of rule* R,

   $ctx'(d_1, d', d'', d''') = R(ctx'(d_1, d', d'', d'''), \ \dots \ , ctx'(d_n, d', d'', d'''))$

Algorithm A.7. ***(presupposition algorithms 2)***

1. *If* d *is a derivation of* $\Gamma \vdash K_1 = K_2$, *then* $pre^1_4(d)$ *is a derivation of* $\Gamma \vdash K_1 \textbf{ kind}$, *then we have* $pre^2_4(d)$ *as a derivation of* $\Gamma \vdash K_2 \textbf{ kind}$;

   The algorithm is constructed inductively on the derivation of d, *depending on the last rule of* d:

   (a) *rule(2.1)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K \textbf{ kind}}{\Gamma \vdash K = K}$,

   $pre^1_4(d) \equiv pre^2_4(d) \equiv d_1$

   (b) *rule(2.2)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K = K'}{\Gamma \vdash K' = K}$,

   $pre^1_4(d) \equiv pre^2_4(d_1)$

   $pre^2_4(d) \equiv pre^1_4(d_1)$

   (c) *rule(2.3)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K = K' \quad \Gamma \overset{d_2}{\vdash} K' = K''}{\Gamma \vdash K = K''}$,

   $pre^1_4(d) \equiv pre^1_4(d_1)$

   $pre^2_4(d) \equiv pre^2_4(d_2)$

   (d) *rule(4.3)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A = B : \textbf{Type}}{\Gamma \vdash El(A) = El(B)}$,

   $pre^1_4(d) \equiv R_{(4.3)}(pre^1_5(d_1))$

   $pre^2_4(d) \equiv R_{(4.3)}(pre^2_5(d_1))$

(e) *rule(6.2)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1 = K_2 \quad \Gamma, x : K_1 \overset{d_2}{\vdash} K_1' = K_2'}{\Gamma \vdash (x : K_1)K_1' = (x : K_2)K_2'}$,

$pre_4^1(d) \equiv R_{(6.1)}(pre_4^1(d_1), pre_4^1(d_2))$

$pre_4^2(d) \equiv R_{(6.1)}(pre_4^2(d_1), ctx'(pre_4^2(d_2), d_1, pre_4^1(d_1), pre_4^2(d_1)))$

2. *If d is a derivation of $\Gamma \vdash k_1 = k_2 : K$, then $pre_5^1(d)$ is a derivation of $\Gamma \vdash k_1 : K$, $pre_5^2(d)$ is a derivation of $\Gamma \vdash k_2 : K$.*

The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

(a) *rule(2.4)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k : K}{\Gamma \vdash k = k : K}$,

$pre_5^1(d) \equiv pre_5^2(d) \equiv d_1$

(b) *rule(2.5)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k = k' : K}{\Gamma \vdash k' = k : K}$,

$pre_5^1(d) \equiv pre_5^2(d_1)$

$pre_5^2(d) \equiv pre_5^1(d_1)$

(c) *rule(2.6)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k = k' : K \quad \Gamma \overset{d_2}{\vdash} k' = k''}{\Gamma \vdash k = k'' : K}$,

$pre_5^1(d) \equiv pre_5^1(d_1)$

$pre_5^2(d) \equiv pre_5^2(d_2)$

(d) *rule(3.2)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k = k' : K \quad \Gamma \overset{d_2}{\vdash} K = K'}{\Gamma \vdash k = k' : K'}$,

$pre_5^1(d) \equiv R_{(3.1)}(pre_5^1(d_1), d2)$

$pre_5^2(d) \equiv R_{(3.1)}(pre_5^2(d_1), d2)$

(e) *rule(6.4)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1 = K_2 \quad \Gamma, x : K_1 \overset{d_2}{\vdash} k_1 = k_2 : K}{\Gamma \vdash [x : K_1]k_1 = [x : K_2]k_2 : (x : K_1)K}$,

$pre_5^1(d) \equiv R_{(6.3)}(pre_5^1(d_2))$

$pre_5^2(d) \equiv R_{(3.1)}(R_{(6.3)}(h_1), h_2)$

where

$h_1 \equiv ctx'(pre_5^2(d_2), d_1, pre_4^1(d_1), pre_4^2(d_1))$

$h_2 \equiv R_{(2.2)}(R_{(6.2)}(d_1, R_{(2.1)}(pre_6(d_2))))$

(f) *rule(6.6)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f = f' : (x : K)K' \quad \Gamma \overset{d_2}{\vdash} k_1 = k_2 : K}{\Gamma \vdash f(k_1) = f'(k_2) : [k_1/x]K'}$,

$pre_5^1(d) \equiv R_{(6.5)}(pre_5^1(d_1), pre_5^1(d_2))$

$pre_5^2(d) \equiv R_{(3.2)}(R_{(6.5)}(pre_5^2(d_1), pre_5^2(d_2)), R_{(2.2)}(h))$

where

$h \equiv sub_6'(pre_3(pre_6(d_1)), d_2, pre_5^1(d_2), pre_5^2(d_2))$

(g) *rule(6.7)*, $d \equiv \dfrac{\Gamma, x : K \overset{d_1}{\vdash} k' : K' \quad \Gamma \overset{d_2}{\vdash} k : K}{\Gamma \vdash ([x : K]k')k = [k/x]k' : [k/x]K'}$,

$pre_5^1(d) \equiv R_{(6.5)}(R_{(6.3)}(d_1), d_2)$

$pre_5^2(d) \equiv sub_3(d, d')$

(h) *rule(6.8)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f : (x : K)K'}{\Gamma \vdash [x : K]f(x) = f : (x : K)K'}(x \notin FV(\Gamma))$,

$pre_5^1(d) \equiv R_{(6.3)}(R_{(6.5)}(h_1, h_2))$

where

$h_1 \equiv wkn(d_1, pre_1(pre_3(d_1), (\Gamma, x : K)), \Gamma)$

$h_2 \equiv R_{(1.3)}(pre_1(pre_3(d_1), (\Gamma, x : K)), \Gamma)$

$pre_5^2(d) \equiv d_1$

(i) *rule(CA2)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f = f' : (x : K_1)K' \quad \Gamma \overset{d_2}{\vdash} k_0 = k_0' : K_0 \quad \Gamma \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma \vdash f(k_0) = f'(k_0') : [c(k_0)/x]K'}$,

$pre_5^1(d) \equiv R_{(CA1)}(pre_5^1(d_1), pre_5^2(d_2), d_3)$

$pre_5^2(d) \equiv R_{(3.1)}(R_{(CA1)}(pre_5^1(d_1), pre_5^2(d_2), d_3), R_{(2.2)}(sub_6'(per_3(pre_6(d_1)), h_1, h_2, h_3)))$

where

$h_1 \equiv R_{(6.6)}(R_{(2.4)}(co(d_3), d_2))$

$h_2 \equiv R_{(6.5)}(co(d_3), pre_5^1(d_2))$

$h_3 \equiv R_{(6.5)}(co(d_3), pre_5^2(d_2))$

(j) *rule(CD)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f : (x : K_1)K' \quad \Gamma \overset{d_2}{\vdash} k_0 : K_0 \quad \Gamma \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma \vdash f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$,

$pre_5^1(d) \equiv R_{(CA1)}(d_1, d_2, d_3)$

$pre_5^2(d) \equiv R_{(6.5)}(d_1, R_{(6.5)}(d_2, co(d_3)))$

3. *If d is a derivation of $\Gamma \vdash \Sigma : K$, then $pre_6(d)$ is a derivation of $\Gamma \vdash K$ **kind** (Σ denotes term or term equality here).*

The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

(a) *rule(1.3)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} \textbf{valid}}{\Gamma, x : K, \Gamma' \vdash x : K}$,

$pre_6(d) \equiv wkn(pre_2(d_1, \Gamma), d_1, \Gamma)$

(b) *rule(2.4)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k : K}{\Gamma \vdash k = k : K}$,

$pre_6(d) \equiv pre_6(d_1)$

(c) *rule(2.5)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k = k' : K}{\Gamma \vdash k' = k : K}$,

$pre_6(d) \equiv pre_6(d_1)$

(d) *rule(2.6)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k = k' : K \quad \Gamma \overset{d_2}{\vdash} k' = k'' : K}{\Gamma \vdash k = k'' : K}$,
$pre_6(d) \equiv pre_6(d_1)$

(e) *rule(3.1)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k : K \quad \Gamma \overset{d_2}{\vdash} K = K'}{\Gamma \vdash k : K'}$,
$pre_6(d) \equiv pre_4^2(d_2)$

(f) *rule(3.2)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} k = k' : K \quad \Gamma \overset{d_2}{\vdash} K = K'}{\Gamma \vdash k = k' : K'}$,
$pre_6(d) \equiv pre_4^2(d_2)$

(g) *rule(6.3)*, $d \equiv \dfrac{\Gamma, x : K \overset{d_1}{\vdash} k : K'}{\Gamma \vdash [x : K]k : (x : K)K'}$,
$pre_6(d) \equiv R_{(6.1)}(pre_6(d_1))$

(h) *rule(6.4)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1 = K_2 \quad \Gamma, x : K_1 \overset{d_2}{\vdash} k_1 = k_2 : K}{\Gamma \vdash [x : K_1]k_1 = [x : K_2]k_2 : (x : K_1)K}$,
$pre_6(d) \equiv R_{(6.1)}(pre_6(d_2))$

(i) $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f : (x : K)K' \quad \Gamma \overset{d_2}{\vdash} k : K}{\Gamma \vdash f(k) : [k/x]K'}$,
$pre_6(d) \equiv sub_2(pre_3(pre_6(d_1)), d_2)$

(j) $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f = f' : (x : K)K' \quad \Gamma \overset{d_2}{\vdash} k_1 = k_2 : K}{\Gamma \vdash f(k_1) = f'(k_2) : [k_1/x]K'}$,
$pre_6(d) \equiv sub_2(pre_3(pre_6(d_1)), pre_5^1(d_2))$

(k) $d \equiv \dfrac{\Gamma, x : K \overset{d_1}{\vdash} k' : K' \quad \Gamma \overset{d_2}{\vdash} k : K}{\Gamma \vdash ([x : K]k')k = [k/x]k' : [k/x]K'}$,
$pre_6(d) \equiv sub_2(pre_6(d_1)), d_2)$

(l) $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f : (x : K)K'}{\Gamma \vdash [x : K]f(x) = f : (x : K)K'} (x \notin FV(\Gamma))$,
$pre_6(d) \equiv pre_6(d_1)$

(m) *rule(CA1)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f : (x : K_1)K' \quad \Gamma \overset{d_2}{\vdash} k_0 : K_0 \quad \Gamma \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma \vdash f(k_0) : [c(k_0)/x]K'}$,
$pre_6(d) \equiv sub_2(pre_3(pre_6(d_1)), R_{(6.5)}(co(d_3), d_2))$

(n) *rule(CA2)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f = f' : (x : K_1)K' \quad \Gamma \overset{d_2}{\vdash} k_0 = k_0' : K_0 \quad \Gamma \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma \vdash f(k_0) = f'(k_0') : [c(k_0)/x]K'}$,
$pre_6(d) \equiv sub_2(pre_3(pre_6(d_1)), R_{(6.5)}(co(d_3), pre_5^1(d_2))$

(o) *rule(CD)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} f : (x : K_1)K' \quad \Gamma \overset{d_2}{\vdash} k_0 : K_0 \quad \Gamma \overset{d_3}{\vdash} K_0 <_c K_1}{\Gamma \vdash f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$,
$pre_6(d) \equiv sub_2(pre_3(pre_6(d_1)), R_{(6.5)}(co(d_3), d_2))$

4. *If d is a derivation of $\Gamma \vdash A <_c B : \mathbf{Type}$, then $pre_7^1(d)$ is a derivation of $\Gamma \vdash A : \mathbf{Type}$, $pre_7^2(d)$ is a derivation of $\Gamma \vdash B : \mathbf{Type}$, $co_t(d)$ is a derivation of $\Gamma \vdash c : (El(A))El(B)$*

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

   (a) *rule(ST1)*, $d \equiv \dfrac{\Gamma \vdash A <_c B : \mathbf{Type} \in \mathcal{C}}{\Gamma \vdash A <_c B : \mathbf{Type}}$,
   the algorithm simply generated by the coherence.

   (b) *rule(ST2)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A <_c B : \mathbf{Type} \quad \Gamma \overset{d_2}{\vdash} c = c' : (El(A))El(B)}{\Gamma \vdash A <_{c'} B : \mathbf{Type}}$,
   $pre_7^1(d) \equiv pre_7^1(d_1)$
   $pre_7^2(d) \equiv pre_7^2(d_1)$
   $co_t(d) \equiv pre_5^2(d_2)$

   (c) *rule(ST3)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A <_c B : \mathbf{Type} \quad \Gamma \overset{d_2}{\vdash} A = A' : \mathbf{Type}}{\Gamma \vdash A' <_c B : \mathbf{Type}}$,
   $pre_7^1(d) \equiv pre_5^2(d_2)$
   $pre_7^2(d) \equiv pre_7^2(d_1)$
   $co_t(d) \equiv R_{(3.1)}(co_t(d_1), R_{(6.2)}(R_{(5.3)}(d_2), wkn(h_1, h_2, \Gamma)))$
   where
   $h_1 \equiv R_{(2.1)}(R_{(5.2)}(pre_7^2(d_1)))$
   $h_2 \equiv R_{(1.2)}(R_{(5.2)}(pre_7^1(d_1)))$

   (d) *rule(ST4)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A <_c B : \mathbf{Type} \quad \Gamma \overset{d_2}{\vdash} B = B' : \mathbf{Type}}{\Gamma \vdash A <_c B' : \mathbf{Type}}$,
   $pre_7^1(d) \equiv pre_7^1(d_1)$
   $pre_7^2(d) \equiv pre_5^2(d_2)$
   $co_t(d) \equiv R_{(3.1)}(co_t(d_1), R_{(6.2)}(h_1, wkn(R_{(5.2)}(d_2), h_2, \Gamma)))$
   where
   $h_1 \equiv R_{(5.2)}(pre_7^1(d_1))$
   $h_2 \equiv R_{(3.2)}(pre_7^1(d_1))$

   (e) *rule(ST5)*, $d \equiv \dfrac{\Gamma \vdash A <_{c_1} B : \mathbf{Type} \quad \Gamma \vdash B <_{c_2} C : \mathbf{Type}}{\Gamma \vdash A <_{c_2 \circ c_1} C : \mathbf{Type}}$,
   $pre_7^1(d) \equiv pre_7^1(d_1)$
   $pre_7^2(d) \equiv pre_7^2(d_2)$
   $co_t(d) \equiv R_{(6.3)}(R_{(6.5)}(wkn(co_t(d_2), h_1, \Gamma), R_{(6.5)}(wkn(co_t(d_1), h_1, \Gamma), R_{(1.3)}(h_1, \Gamma)))$,
   where $h_1 \equiv R_{(1.2)}(R_{(5.2)}(pre_7^1(d_1)))$

(f) *rule(ST6)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} A <_c B : \mathbf{Type} \quad \Gamma \overset{d_2}{\vdash} k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]A <_{[k/x]c} [k/x]B : \mathbf{Type}}$,

$pre^1_7(d) \equiv sub_3(pre^1_7(d_1), d_2)$

$pre^2_7(d) \equiv sub_3(pre^2_7(d_1), d_2)$

$co_t(d) \equiv sub_3(co_t(d_1), d_2)$

(g) *rule(ST7)*, $d \equiv \dfrac{\Gamma, \Gamma' \overset{d_1}{\vdash} A <_c B : \mathbf{Type} \quad \Gamma, \Gamma'' \overset{d_2}{\vdash} \mathbf{valid}}{\Gamma, \Gamma'', \Gamma' \vdash A <_c B : \mathbf{Type}}$,

$pre^1_7(d) \equiv wkn(pre^1_7(d_1), d_2, \Gamma)$

$pre^2_7(d) \equiv wkn(pre^2_7(d_1), d_2, \Gamma)$

$co_t(d) \equiv wkn(co_t(d_1), d_2, \Gamma)$

(h) *rule(ST8)*, $d \equiv \dfrac{\Gamma, x : K, \Gamma' \overset{d_1}{\vdash} A <_c B : \mathbf{Type} \quad \Gamma \overset{d_2}{\vdash} K = K'}{\Gamma, x : K', \Gamma' \vdash A <_c B : \mathbf{Type}}$,

$pre^1_7(d) \equiv ctx'(pre^1_7(d_1), d_2, pre^1_5(d_2), pre^2_5(d_2))$

$pre^2_7(d) \equiv ctx'(pre^2_7(d_1), d_2, pre^1_5(d_2), pre^2_5(d_2))$

$co_t(d) \equiv ctx'(co_t(d_1), d_2, pre^1_5(d_2), pre^2_5(d_2))$

5. *If d is a derivation of* $\Gamma \vdash K_1 <_c K_2$, *then* $pre^1_8(d)$ *is a derivation of* $\Gamma \vdash K_1$ **kind**, $pre^2_8(d)$ *is a derivation of* $\Gamma \vdash K_2$ **kind**, $co(d)$ *is a derivation of* $\Gamma \vdash c : (K_1)K_2$

   The algorithm is constructed inductively on the derivation of d, depending on the last rule of d:

(a) *rule(SK1)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A <_c B : \mathbf{Type}}{\Gamma \vdash El(A) <_c El(B)}$,

$pre^1_8(d) \equiv R_{(5.2)}(pre^1_7(d_1))$

$pre^2_8(d) \equiv R_{(5.2)}(pre^2_7(d_1))$

$co(d) \equiv co_t(d_1)$

(b) *rule(SK2)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1' <_{c_1} K_1 \quad \Gamma, x' : K_1' \overset{d_2}{\vdash} [c_1(x')/x]K_2 = K_2' \quad \Gamma, x : K_1 \overset{d_3}{\vdash} K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$,

where $c \equiv [f : (x : K_1)K_2][x' : K_1']f(c_1(x'))$,

$pre^1_8(d) \equiv R_{(6.1)}(d_3)$

$pre^2_8(d) \equiv R_{(6.1)}(pre^2_4(d_2))$

$co(d) \equiv R_{(6.3)}(R_{(6.3)}(R_{(3.1)}(h_3, wkn(d_2, h_1, \Gamma))))$

where

$h_0 \equiv pre_1(d_2, (\Gamma, x' : K_1'))$

$h_1 \equiv R_{(1.2)}(R_{(6.1)}(d_3))$

$h_2 \equiv R_{(6.5)}(wkn(co(d_1), h_0, \Gamma), R_{(1.3)}(h_0, \Gamma))$

$h_3 \equiv R_{(6.5)}(R_{(1.3)}(wkn(h_0, h_1, \Gamma), \Gamma), wkn(h_2, h_1, \Gamma))$

(c) *rule(SK3)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1' = K_1 \quad \Gamma, x' : K_1' \overset{d_2}{\vdash} K_2 <_{c_2} K_2' \quad \Gamma, x : K_1 \overset{d_3}{\vdash} K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$,

where $c \equiv [f : (x : K_1)K_2][x' : K_1']c_2 f(x')$,

$pre_8^1(d) \equiv R_{(6.1)}(d_3)$

$pre_8^2(d) \equiv R_{(6.1)}(pre_8^2(d_2))$

$co(d) \equiv R_{(6.3)}(R_{(6.3)}(R_{(6.5)}(wkn(co(d_2), h_1, \Gamma), h_3)))$

where

$h_0 \equiv pre_1(d_2, (\Gamma, x' : K_1'))$

$h_1 \equiv R_{(1.2)}(R_{(6.1)}(d_3))$

$h_2 \equiv R_{(3.1)}(R_{(1.3)}(h_0, \Gamma), wkn(d_1, h_0, \Gamma)$

$h_3 \equiv R_{(6.5)}(R_{(1.3)}(h_4, \Gamma), wkn(h_2, h_1, \Gamma))$

$h_4 \equiv wkn(h_0, h_1, \Gamma)$

(d) *rule(SK4)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1' <_{c_1} K_1 \quad \Gamma, x' : K_1' \overset{d_2}{\vdash} [c_1(x')/x]K_2 <_{c_2} K_2' \quad \Gamma, x : K_1 \overset{d_3}{\vdash} K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 <_c (x' : K_1')K_2'}$

where $c \equiv [f : (x : K_1)K_2][x' : K_1']c_2(f(c_1(x')))$,

$pre_8^1(d) \equiv R_{(6.1)}(d_3)$

$pre_8^2(d) \equiv R_{(6.1)}(pre_8^2(d_2))$

$co(d) \equiv R_{(6.3)}(R_{(6.3)}(R_{(6.5)}(wkn(co(d_2), h_1, \Gamma), h_3)))$

where

$h_0 \equiv pre_1(d_2, (\Gamma, x' : K_1'))$

$h_1 \equiv R_{(1.2)}(R_{(6.1)}(d_3))$

$h_2 \equiv R_{(6.5)}(wkn(co(d_1), h_0, \Gamma), R_{(1.3)}(h_0, \Gamma))$

$h_3 \equiv R_{(6.5)}(R_{(1.3)}(wkn(h_0, h_1, \Gamma), \Gamma), wkn(h_2, h_1, \Gamma))$

(e) *rule(SK5)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K <_c K' \quad \Gamma \overset{d_2}{\vdash} c = c' : (K)K'}{\Gamma \vdash K <_{c'} K'}$,

$pre_8^1(d) \equiv pre_8^1(d_1)$

$pre_8^2(d) \equiv pre_8^2(d_1)$

$co(d) \equiv pre_5^2(d_2)$

(f) *rule(SK6)*, $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1 <_c K_2 \quad \Gamma \overset{d_2}{\vdash} K_1 = K_1'}{\Gamma \vdash K_1' <_c K_2}$,

$pre_8^1(d) \equiv pre_5^2(d_2)$

$pre_8^2(d) \equiv pre_8^2(d_2)$

$co(d) \equiv R_{(3.1)}(co(d_1), R_{(6.2)}(d_2, wkn(h_1, h_2, \Gamma)))$,

where

$h_1 \equiv R_{(2.1)}(pre_8^2(d_1))$

$h_2 \equiv R_{(1.2)}(pre_8^1(d_1))$

(g) *rule(SK7)*, $d \equiv \dfrac{\overset{d_1}{\Gamma \vdash K_1 <_c K_2} \quad \overset{d_2}{\Gamma \vdash K_2 = K_2'}}{\Gamma \vdash K_1 <_c K_2'}$,

$pre_8^1(d) \equiv pre_8^1(d_1)$

$pre_8^2(d) \equiv pre_5^2(d_2)$

$co(d) \equiv R_{(3.1)}(co(d_1), R_{(6.2)}(h_1, wkn(d_2, h_2, \Gamma)))$

where

$h_1 \equiv R_{(2.1)}(pre_8^1(d_1))$

$h_2 \equiv R_{(1.2)}(pre_8^1(d_1))$

(h) *rule(SK8)*, $d \equiv \dfrac{\overset{d_1}{\Gamma \vdash K_1 <_{c_1} K_2} \quad \overset{d_2}{\Gamma \vdash K_2 <_{c_2} K_3}}{\Gamma \vdash K_1 <_{c_2 \circ c_1} K_3}$,

$pre_8^1(d) \equiv pre_8^1(d_1)$

$pre_8^2(d) \equiv pre_8^2(d_2)$

$co_t(d) \equiv R_{(6.3)}(R_{(6.5)}(wkn(co(d_2), h_1, \Gamma), R_{(6.5)}(wkn(co(d_1), h_1, \Gamma), R_{(1.3)}(h_1, \Gamma)))$,

where $h_1 \equiv R_{(1.2)}(pre_8^1(d_1))$

Algorithm A.8. ***(substitutions 2 – equality substitutions)***

1. *If* d *is a derivation of* $\Gamma, x : K, \Gamma' \vdash K'$ **kind**, d' *is a derivation of* $\Gamma \vdash k_1 = k_2 : K$, *then* $sub_6(d, d')$ *is a derivation of* $\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]K' = [k_2/x]K'$.

   The algorithm is constructed as follows:

   $sub_6(d, d') \equiv sub_6'(d, d', pre_5^1(d'), pre_5^2(d'))$

2. *If* d *is a derivation of* $\Gamma, x : K, \Gamma' \vdash k' : K'$, d' *is a derivation of* $\Gamma \vdash k_1 = k_2 : K$, *then* $sub_7(d, d')$ *is a derivation of* $\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]k' = [k_2/x]k' : [k_1/x]K'$.

   The algorithm is constructed as follows:

   $sub_7(d, d') \equiv sub_7'(d, d', pre_5^1(d'), pre_5^2(d'))$

Algorithm A.9. ***(context retyping)*** *If* d *is a derivation of* $\Gamma, x : K, \Gamma' \vdash J$, d' *is a derivation of* $\Gamma \vdash K = K'$, *then we have* $ctx(d, d')$ *as a derivation of* $\Gamma, x : K', \Gamma' \vdash J$.

The algorithm is constructed as follows:

$ctx(d, d') \equiv ctx'(d, d', pre_4^1(d'), pre_4^2(d'))$

Lemma A.10. *In system* $T[\mathcal{C}]_{0K}$, *the following hold:*

1. *If* $\Gamma \vdash El(A) = K$ *is derivable, then there is a term* B, *such that* $K \equiv El(B)$.
2. *If* $\Gamma \vdash K = El(A)$ *is derivable, then there is a term* B, *such that* $K \equiv El(B)$.

Proof. Proof the two properties simultaneously.

1. Induction on the derivation d of $\Gamma \vdash El(A) = K$, depending on the last rule of d, rule(6.2) doesn't satisfy the syntax.

   (a) rule(2.1), $K \equiv El(A)$

   (b) rule(2.2), proved by IH with 2

   (c) rule(2.3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} El(A) = K_0 \quad \Gamma \overset{d_2}{\vdash} K_0 = K}{\Gamma \vdash El(A) = K}$

   We first use IH on d_1, there's a term C, such that $K_0 \equiv El(C)$. Then use IH on d_2, lemma is proved.

   (d) rule(5.3), lemma is proved trivially.

2. Induction on the derivation d of $\Gamma \vdash K = El(A)$, depending on the last rule of d, rule(6.2) doesn't satisfy the syntax.

   (a) rule(2.1), $K \equiv El(A)$

   (b) rule(2.2), proved by IH with 1

   (c) rule(2.3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K = K_0 \quad \Gamma \overset{d_2}{\vdash} K_0 = El(A)}{\Gamma \vdash K = El(A)}$

   We first use IH on d_2, there's a term C, such that $K_0 \equiv El(C)$. Then use IH on d_1, lemma is proved.

   (d) rule(5.3), lemma is proved trivially.

□

Algorithm A.11. *tp is a transformation from $T[\mathcal{C}]^-_{0K}$ to $T[\mathcal{C}]^-_{0K}$. For any derivation d of judgement $\Gamma \vdash El(A) = El(B)$ in $T[\mathcal{C}]^-_{0K}$, $tp(d)$ is a derivation of $\Gamma \vdash A = B :$ **Type**.*

The algorithm is constructed inductively on the derivation of d, according to the last rule of d:

1. rule(2.1), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} El(A)\ \mathbf{kind}}{\Gamma \vdash El(A) = El(A)}$.

   The derivation d_1 could only be $d_1 \equiv \dfrac{\Gamma \overset{d_1'}{\vdash} A : \mathbf{Type}}{\Gamma \vdash El(A)\ \mathbf{kind}}$, hence

   $tp(d) \equiv R_{(2.4)}(d_1')$

2. rule(2.2), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} El(B) = El(A)}{\Gamma \vdash El(A) = El(B)}$.

   $tp(d) \equiv R_{(2.2)}(tp(d_1))$

3. rule(2.3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} El(A) = K \quad \Gamma \overset{d_2}{\vdash} K = El(B)}{\Gamma \vdash El(A) = El(B)}$.

   Using lemma A.10, there's term C such that $K \equiv El(C)$,

   $tp(d) \equiv R_{(2.3)}(tp(d_1), tp(d_2))$

4. rule(5.3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A = B : \mathbf{Type}}{\Gamma \vdash El(A) = El(B)}$.

   $tp(d) \equiv d_1$

5. rule(6.2), the syntax doesn't apply to the case.

Lemma A.12. *In system $T[\mathcal{C}]_{0K}$, the following hold:*

1. *If $\Gamma \vdash (x : K_1)K_2 = M$, then there are terms N_1, N_2, such that $M \equiv (x : N_1)N_2$.*

2. *If $\Gamma \vdash M = (x : K_1)K_2$, then there are terms N_1, N_2, such that $M \equiv (x : N_1)N_2$.*

Proof. Proof the two properties simultaneously.

1. Induction on the derivation d of $\Gamma \vdash (x : K_1)K_2 = M$, according to the last rule of d, rule(5.3) doesn't satisfy the syntax.

   (a) rule(2.1), $M \equiv (x : K_1)K_2$

   (b) rule(2.2), proved by IH with 2

   (c) rule(2.3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} (x : K_1)K_2 = M_0 \quad \Gamma \overset{d_2}{\vdash} M_0 = M}{\Gamma \vdash (x : K_1)K_2 = M}$.

   We first use IH on d_1, there're terms N_1', N_2', such that $M_0 \equiv (x : N_1')N_2'$. Then use IH on d_2, lemma is proved.

   (d) rule(6.2), lemma is proved trivially.

2. Induction on the derivation d of $\Gamma \vdash M = (x : K_1)K_2$, according to the last rule of d, rule(5.3) doesn't satisfy the syntax.

   (a) rule(2.1), $M \equiv (x : K_1)K_2$

   (b) rule(2.2), proved by IH with 1

   (c) rule(2.3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M = M_0 \quad \Gamma \overset{d_2}{\vdash} M_0 = (x : K_1)K_2}{\Gamma \vdash M = (x : K_1)K_2}$.

   We first use IH on d_2, there're terms N_1', N_2', such that $M_0 \equiv (x : N_1')N_2'$. Then use IH on d_1, lemma is proved.

   (d) rule(6.2), lemma is proved trivially.

□

Algorithm A.13. *$prod_1$ and $prod_2$ are transformations from $T[\mathcal{C}]^-_{0K}$ to $T[\mathcal{C}]^-_{0K}$. For any derivation d of judgement $\Gamma \vdash (x : K_1)K_2 = (x : N_1)N_2$ in $T[\mathcal{C}]^-_{0K}$, $prod_1(d)$ is a derivation of $\Gamma \vdash K_1 = N_1$ and $prod_2(d)$ is a derivation of $\Gamma, x : K_1 \vdash K_2 = N_2$.*

The algorithm is constructed inductively on the derivation of d, according to the last rule of d:

1. rule(2.1), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} (x : K_1)K_2 \ \mathbf{kind}}{\Gamma \vdash (x : K_1)K_2 = (x : K_1)K_2}$.

   $prod_1(d) \equiv R_{(2.1)}(pre_2(pre_3(d_1)), \Gamma)$

   $prod_2(d) \equiv R_{(2.1)}(pre_3(d_1))$

2. rule(2.2), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} (x : N_1)N_2 = (x : K_1)K_2}{\Gamma \vdash (x : K_1)K_2 = (x : N_1)N_2}$.

   $prod_1(d) \equiv R_{(2.2)}(prod_1(d_1))$

   $prod_2(d) \equiv R_{(2.2)}(ctx(prod_2(d_1), prod_1(d_1)))$

3. rule(2.3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} (x : K_1)K_2 = M \quad \Gamma \overset{d_2}{\vdash} M = (x : N_1)N_2}{\Gamma \vdash (x : K_1)K_2 = (x : N_1)N_2}$.

   Using lemma A.12, there are terms M_1, M_2 such that $M \equiv (x : M_1)M_2$,

   $prod_1(d) \equiv R_{(2.3)}(prod_1(d_1), prod_1(d_2))$

   $prod_2(d) \equiv R_{(2.3)}(prod_2(d_1), ctx(prod_2(d_2), R_{(2.2)}(prod_1(d_1))))$

4. rule(5.3), the syntax doesn't apply to the case.

5. rule(6.2), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1 = N_1 \quad \Gamma, x : K_1 \overset{d_2}{\vdash} K_2 = N_2}{\Gamma \vdash (x : K_1)K_2 = (x : N_1)N_2}$.

   $prod_1(d) \equiv d_1$

   $prod_2(d) \equiv d_2$

Algorithm A.14. *If d is a derivation of $\Gamma \vdash K_1 <_c K_2$, d' is a derivation of $\Gamma \vdash K_1 = K_1'$, $trans_1(d, d')$ is a derivation of $\Gamma \vdash K_1' <_c K_2$.*

The algorithm is constructed inductively on the derivation of d, according to the last rule of d.

1. rule(SK1), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A <_c B : \mathbf{Type}}{\Gamma \vdash El(A) <_c El(B)}$.

   So d' is a derivation of $\Gamma \vdash El(A) = K_1'$, using lemma A.10, there's term A', such that $K_1' \equiv El(A')$.

   $trans_1(d, d') \equiv R_{(SK1)}(R_{(ST3)}(d_1, tp(d')))$

2. rule(SK2), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 = M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$,
   where $c \equiv [f : (x : M_1)M_2][x' : M_1']f(c_1(x'))$.

   So d' is a derivation of $\Gamma \vdash (x : M_1)M_2 = K_1'$, using lemma A.12, there are terms N_1 and N_2, such that $K_1' \equiv (x : N_1)N_2$.

   $\Gamma \overset{d'}{\vdash} (x : M_1)M_2 = (x : N_1)N_2$

   $prod_1(d')$ is a derivation of $\Gamma \vdash M_1 = N_1$, $prod_2(d')$ is a derivation of $\Gamma, x : M_1 \vdash M_2 = N_2$

   $trans_1(d, d') \equiv R_{(SK5)}(h_3, h_7)$

   where

   $h_0 \equiv pre_1(d_2, (\Gamma, x' : M_1'))$

   $h_1 \equiv R_{(6.5)}(wkn(co(d_1), h_0), R_{(1.3)}(h_0, \Gamma))$

   $h_2 \equiv R_{(2.3)}(R_{(2.2)}(sub_4(wkn(prod_2(d'), h_0), h_1)), d_2)$

   $h_3 \equiv R_{(SK2)}(trans_2(d_1, prod_1(d')), h_2, pre_3(pre_4^2(d')))$

   $h_4 \equiv R_{(1.2)}(pre_4^2(d'))$

   $h_5 \equiv R_{(3.2)}(h_1, wkn(prod_1(d'), h_0))$

   $h_6 \equiv R_{(3.2)}(R_{(6.5)}(R_{(1.3)}(wkn(h_0, h_4), \Gamma), wkn(h_5, h_4)), wkn(h_2, h_4))$

   $h_7 \equiv R_{(6.4)}(R_{(2.2)}(d'), R_{(2.5)}(R_{(6.3)}(h_6)))$

3. rule(SK3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M_1' = M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [x'/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$,
   where $c \equiv [f : (x : M_1)M_2][x' : M_1']c_2 f(x')$.

   So d' is a derivation of $\Gamma \vdash (x : M_1)M_2 = K_1'$, using lemma A.12, there are terms N_1 and N_2, such that $K_1' \equiv (x : N_1)N_2$.

   $\Gamma \overset{d'}{\vdash} (x : M_1)M_2 = (x : N_1)N_2$

   $prod_1(d')$ is a derivation of $\Gamma \vdash M_1 = N_1$, $prod_2(d')$ is a derivation of $\Gamma, x : M_1 \vdash M_2 = N_2$

   $trans_1(d, d') \equiv R_{(SK5)}(h_3, h_8)$

   where

   $h_0 \equiv pre_1(d_2, (\Gamma, x' : M_1'))$

   $h_1 \equiv R_{(3.1)}(R_{(1.3)}(h_0, \Gamma), wkn(d_1, h_0))$

   $h_2 \equiv sub_4(wkn(prod_2(d'), h_0), h_1)$

   $h_3 \equiv R_{(SK2)}(R_{(2.3)}(d_1, prod_1(d')), trans_1(d_2, h_2), pre_3(pre_4^2(d')))$

   $h_4 \equiv R_{(1.2)}(pre_4^2)$

   $h_5 \equiv R_{(3.1)}(h_1, wkn(prod_1(d'), h_0)$

$h_6 \equiv R_{(3.1)}(R_{(6.5)}(R_{(1.3)}(wkn(h_0, h_4), \Gamma), h_5), R_{(2.2)}(wkn(h_2, h_4)))$

$h_7 \equiv R_{(6.5}(wkn(co(d_2), h_4), h_6)$

$h_8 \equiv R_{(6.4)}(R_{(2.2)}(d'), R_{(2.5)}(R_{(6.3)}(h_7)))$

4. rule(SK4), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$
   where $c \equiv [f : (x : M_1)M_2][x' : M_1']c_2(f(c_1(x')))$.

   So d' is a derivation of $\Gamma \vdash (x : M_1)M_2 = K_1'$, using lemma A.12, there are terms N_1 and N_2, such that $K_1' \equiv (x : N_1)N_2$.

   $\Gamma \overset{d'}{\vdash} (x : M_1)M_2 = (x : N_1)N_2$

   $prod_1(d')$ is a derivation of $\Gamma \vdash M_1 = N_1$, $prod_2(d')$ is a derivation of $\Gamma, x : M_1 \vdash M_2 = N_2$

   $trans_1(d, d') \equiv R_{(SK5)}(h_3, h_8)$ where,

   $h_0 \equiv pre_1(d_2, (\Gamma, x' : M_1'))$

   $h_1 \equiv R_{(6.5)}(wkn(co(d_1), h_0), R_{(1.3)}(h_0, \Gamma))$

   $h_2 \equiv R_{(3.1)}(R_{(2.2)}(sub_4(wkn(prod_2(d'), h_0), h_1)), d_2)$

   $h_3 \equiv R_{(SK4)}(R_{(2.2)}(trans_2(d_1, prod_1(d')), trans_1(d_2, h_2), pre_3(pre_4^2(d')))$

   $h_4 \equiv R_{(1.2)}(pre_4^2)$

   $h_5 \equiv R_{(3.1)}(h_1, wkn(prod_1(d'), h_0)$

   $h_6 \equiv R_{(3.1)}(R_{(6.5)}(R_{(1.3)}(wkn(h_0, h_4), \Gamma), wkn(h_5, h_4)), R_{(2.2)}(wkn(h_2, h_4)))$

   $h_7 \equiv R_{(6.5}(wkn(co(d_2), h_4), h_6)$

   $h_8 \equiv R_{(6.4)}(R_{(2.2)}(d'), R_{(2.5)}(R_{(6.3)}(h_7)))$

5. rule(SK5), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1 <_c K_2 \quad \Gamma \overset{d_2}{\vdash} c = c' : (K_1)K_2}{\Gamma \vdash K_1 <_{c'} K_2}$,

   $trans_1(d, d') \equiv R_{(SK5)}(trans_1(d_1, d'), R_{(3.2)}(d_2, h))$

   where $h \equiv R_{(6.2)}(d', R_{(2.1)}(pre_3(pre_6(d_2))))$

Algorithm A.15. *If d is a derivation of $\Gamma \vdash K_1 <_c K_2$, d' is a derivation of $\Gamma \vdash K_2 = K_2'$, $trans_2(d, d')$ is a derivation of $\Gamma \vdash K_1 <_c K_2'$.*

The algorithm is constructed inductively on the derivation of d, according to the last rule of d.

1. rule(SK1), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A <_c B : \mathbf{Type}}{\Gamma \vdash El(A) <_c El(B)}$.

   So d' is a derivation of $\Gamma \vdash El(B) = K_2'$, using lemma A.10, there's term B', such that $K_2' \equiv El(B')$.

   $trans_2(d, d') \equiv R_{(SK1)}(R_{(ST4)}(d_1, tp(d')))$

2. rule(SK2), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 = M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \textbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$, where $c \equiv [f : (x : M_1)M_2][x' : M_1']f(c_1(x'))$.

   So d' is a derivation of $\Gamma \vdash (x' : M_1')M_2' = K_2'$, using lemma A.12, there are terms N_1 and N_2, such that $K_2' \equiv (x' : N_1)N_2$.

   $\Gamma \overset{d'}{\vdash} (x' : M_1')M_2' = (x' : N_1)N_2$

   $prod_1(d')$ is a derivation of $\Gamma \vdash M_1' = N_1$, $prod_2(d')$ is a derivation of $\Gamma, x' : M_1' \vdash M_2' = N_2$

   $trans_2(d, d') \equiv R_{(SK5)}(h_1, h_6)$

   $h_0 \equiv pre_1(d_2, (\Gamma, x' : M_1'))$

   $h_1 \equiv R_{(SK2)}(trans_1(d_1, prod_(d')), R_{(2.3)}(d_2, prod_2(d')), d_3)$

   $h_2 \equiv R_{(1.2)}(pre_7^1(d))$

   $h_3 \equiv R_{(6.5)}(wkn(co(d_1), h_0), R_{(1.3)}(h_0, \Gamma))$

   $h_4 \equiv R_{(3.1)}(R_{(6.5)}(R_{(1.3)}(wkn(h_0, h_2), \Gamma)), wkn(R_{(2.3)}(d_2, prod_2(d')), h_2))$

   $h_5 \equiv R_{(6.4)}(R_{(2.2)}(wkn(prod_1(d'), h_2)), R_{(2.5)}(ctx(h_4, wkn(prod_1(d'), h_2))))$

   $h_6 \equiv R_{(6.4)}(R_{(2.1)}(pre_7^1(d)), h_5)$

3. rule(SK3), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M_1' = M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [x'/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \textbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$, where $c \equiv [f : (x : M_1)M_2][x' : M_1']c_2(f(x'))$.

   So d' is a derivation of $\Gamma \vdash (x' : M_1')M_2' = K_2'$, using lemma A.12, there are terms N_1 and N_2, such that $K_2' \equiv (x' : N_1)N_2$.

   $\Gamma \overset{d'}{\vdash} (x' : M_1')M_2' = (x' : N_1)N_2$

   $prod_1(d')$ is a derivation of $\Gamma \vdash M_1' = N_1$, $prod_2(d')$ is a derivation of $\Gamma, x' : M_1' \vdash M_2' = N_2$

   $trans_2(d, d') \equiv R_{(SK5)}(h_1, h_7)$

   $h_0 \equiv pre_1(d_2, (\Gamma, x' : M_1'))$

   $h_1 \equiv R_{(SK3)}(R_(2.3)(d_1, prod_1(d')), trans_2(d_2, prod_2(d')), d_3)$

   $h_2 \equiv R_{(1.2)}(pre_7^1(d))$

   $h_3 \equiv R_{(3.1)}(R_{(1.3)}(h_0, \Gamma), wkn(d_1, h_0))$

   $h_4 \equiv R_{(6.5)}(R_{(1.3)}(wkn(h_0, h_2), \Gamma), wkn(h_3, h_2))$

   $h_5 \equiv R_{(3.1)}(R_{(6.5)}(wkn(co(d_2), h_2), h_4), wkn(prod_2(d'), h_2)$

   $h_6 \equiv R_{(6.4)}(R_{(2.2)}(wkn(prod_1(d'), h_2)), R_{(2.5)}(ctx(h_5, wkn(prod_1(d'), h_2))))$

   $h_7 \equiv R_{(6.4)}(R_{(2.1)}(pre_7^1(d)), h_6)$

4. rule(SK4), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$
   where $c \equiv [f : (x : M_1)M_2][x' : M_1']c_2(f(c_1(x')))$.

   So d' is a derivation of $\Gamma \vdash (x' : M_1')M_2' = K_2'$, using lemma A.12, there are terms N_1 and N_2, such that $K_2' \equiv (x' : N_1)N_2$.

   $\Gamma \overset{d'}{\vdash} (x' : M_1')M_2' = (x' : N_1)N_2$

   $prod_1(d')$ is a derivation of $\Gamma \vdash M_1' = N_1$, $prod_2(d')$ is a derivation of $\Gamma, x' : M_1' \vdash M_2' = N_2$

   $trans_2(d, d') \equiv R_{(SK5)}(h_1, h_7)$ where,

   $h_0 \equiv pre_1(d_2, (\Gamma, x' : M_1'))$

   $h_1 \equiv R_{(SK4)}(trans_1(d_1, prod_1(d')), trans_2(d_2, prod_2(d')), d_3)$

   $h_2 \equiv R_{(1.2)}(pre_7^1(d))$

   $h_3 \equiv R_{(6.5)}(R_{(1.3)}(wkn(co(d_1), h_0), R_{(1.3)}(h_0, \Gamma))$

   $h_4 \equiv R_{(6.5)}(R_{(1.3)}(wkn(h_0, h_2), \Gamma), wkn(h_3, h_2))$

   $h_5 \equiv R_{(3.1)}(R_{(6.5)}(wkn(co(d_2), h_2), h_4), wkn(prod_2(d'), h_2)$

   $h_6 \equiv R_{(6.4)}(R_{(2.2)}(wkn(prod_1(d'), h_2)), R_{(2.5)}(ctx(h_5, wkn(prod_1(d'), h_2))))$

   $h_7 \equiv R_{(6.4)}(R_{(2.1)}(pre_7^1(d)), h_6)$

5. rule(SK5), $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1 <_c K_2 \quad \Gamma \overset{d_2}{\vdash} c = c' : (K_1)K_2}{\Gamma \vdash K_1 <_{c'} K_2}$,

   $trans_2(d, d') \equiv R_{(SK5)}(trans_1(d_1, d'), R_{(3.2)}(d_2, h))$ where,

   $h \equiv R_{(6.2)}(R_{(2.1)}(pre_7^1(d_1)), wkn(d', R_{(1.2)}(pre_7^1(d_1))))$

Algorithm A.16. *If d is a derivation of $\Gamma \vdash K_1 <_c K_2$, d' is a derivation of $\Gamma \vdash K_2 <_{c'} K_3$, $trans_3(d, d')$ is a derivation of $\Gamma \vdash K_1 <_{c' \circ c} K_3$.*

The algorithm is constructed inductively on the derivation of d and d', according to the last rule of the derivations.

1. If one of the derivations ends with rule(SK5),

   (a) $d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} K_1 <_{c_1} K_2 \quad \Gamma \overset{d_2}{\vdash} c_1 = c : (K_1)K_2}{\Gamma \vdash K_1 <_c K_2}$,
   $trans_3(d, d') \equiv R_{(SK5)}(trans_3(d_1, d'), h_1)$ where,
   $h_0 \equiv R_{(1.2)}(pre_7^1(d))$
   $h_1 \equiv R_{(6.4)}(R_{(2.1)}(pre_7^1(d)), R_{(6.6)}(R_{(2.4)}(wkn(co(d'), h_0)), R_{(6.6)}(wkn(d_2, h_0), R_{(1.3)}(h_0, \Gamma))))$

(b) $d' \equiv \dfrac{\Gamma \overset{d_1'}{\vdash} K_2 <_{c_1'} K_3 \quad \Gamma \overset{d_2'}{\vdash} c_1' = c' : (K_2)K_3}{\Gamma \vdash K_2 <_{c'} K_3}$

$trans_3(d, d') \equiv R_{(SK5)}(trans_3(d_1, d'), h_1)$ where,

$h_0 \equiv R_{(1.2)}(pre_7^1(d))$

$h_1 \equiv R_{(6.4)}(R_{(2.1)}(pre_7^1(d)), R_{(6.6)}(wkn(d_2', h_0), R_{(2.4)}(R_{(6.5)}(wkn(co(d), h_0), R_{(1.3)}(h_0, \Gamma)))))$

2. Both d and d' end with rule(SK1),

$d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} A <_c B : \mathbf{Type}}{\Gamma \vdash El(A) <_c El(B)}$ and $d' \equiv \dfrac{\Gamma \overset{d_1'}{\vdash} B <_c' C : \mathbf{Type}}{\Gamma \vdash El(B) <_{c'} El(C)}$.

$K_1 \equiv El(A)$, $K_2 \equiv El(B)$ and $K_3 \equiv El(C)$

$trans_3(d, d') \equiv R_{(SK_1)}(R_{(ST5)}(d_1, d_1'))$

3. Both d and d' end with rule(SK2-4).

(a) d rule(SK2), d' rule(SK2)

$d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 = M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$,

where $c \equiv [f : (x : M_1)M_2][x' : M_1']f(c_1(x'))$.

$d' \equiv \dfrac{\Gamma \overset{d_1'}{\vdash} M_1'' <_{c_1'} M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [c_1'(x'')/x']M_2' = M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \mathbf{kind}}{\Gamma \vdash (x' : M_1')M_2' <_{c'} (x'' : M_1'')M_2''}$,

where $c' \equiv [f' : (x' : M_1')M_2'][x'' : M_1'']f'(c_1'(x''))$.

$trans_3(d, d') \equiv R_{(SK2)}(trans_3(d_1', d_1), h_2, d_3)$

where

$h_0 \equiv pre_1(d_2, (\Gamma, x'' : M_1''))$

$h_1 \equiv R_{(6.5)}(wkn(d_1', h_0), R_{(1.3)}(h_0, \Gamma))$

$h_2 \equiv R_{(2.3)}(sub_4(wkn(d_2, h_0), h_1), d_2')$

(b) d rule(SK2), d' rule(SK3)

$d \equiv \dfrac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 = M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$,

where $c \equiv [f : (x : M_1)M_2][x' : M_1']f(c_1(x'))$.

$d' \equiv \dfrac{\Gamma \overset{d_1'}{\vdash} M_1'' = M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [x''/x']M_2' <_{c_2'} M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \mathbf{kind}}{\Gamma \vdash (x' : M_1')M_2' <_{c'} (x'' : M_1'')M_2''}$,

where $c' \equiv [f' : (x : M_1')M_2'][x'' : M_1'']c_2'(f'(x''))$.

$trans_3(d, d') \equiv R_{(SK4)}(trans_1(d_1, R_{(2.2)}(d_1')), h_2, d_3)$

where

$h_0 \equiv pre_1(d_2, (\Gamma, x'' : M_1''))$

$h_1 \equiv R_{(3.1)}(R_{(1.3)}(h_0, \Gamma), wkn(d_1', h_0))$

$h_2 \equiv sub_4(wkn(d_2, h_0), h_1)$

(c) d rule(SK2), d' rule(SK4)

$$d \equiv \frac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 = M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'},$$

where $c \equiv [f : (x : M_1)M_2][x' : M_1']f(c_1(x'))$.

$$d' \equiv \frac{\Gamma \overset{d_1'}{\vdash} M_1'' <_{c_1'} M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [c_1'(x'')/x']M_2' <_{c_2'} M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \mathbf{kind}}{\Gamma \vdash (x' : M_1')M_2' <_{c'} (x'' : M_1'')M_2''}$$

where $c' \equiv [f' : (x' : M_1')M_2'][x'' : M_1'']c_2'(f'(c_1'(x'')))$.

$trans_3(d, d') \equiv R_{(SK4)}(trans_3(d_1', d_1), h_2, d_3)$ where,

$h_0 \equiv pre_1(d_2, (\Gamma, x'' : M_1''))$

$h_1 \equiv R_{(6.5)}(wkn(d_1', h_0), R_{(1.3)}(h_0, \Gamma))$

$h_2 \equiv trans_1(d_2', R_{(2.2)}(sub_4(wkn(d_2, h_0), h_1)))$

(d) d rule(SK3), d' rule(SK2)

$$d \ \equiv \ \frac{\Gamma \overset{d_1}{\vdash} M_1' = M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [x'/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'},$$

where $c \equiv [f : (x : M_1)M_2][x' : M_1']c_2(f(x'))$.

$$d' \equiv \frac{\Gamma \overset{d_1'}{\vdash} M_1'' <_{c_1'} M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [c_1'(x'')/x']M_2' = M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \mathbf{kind}}{\Gamma \vdash (x' : M_1')M_2' <_{c'} (x'' : M_1'')M_2''},$$

where $c' \equiv [f' : (x' : M_1')M_2'][x'' : M_1'']f'(c_1'(x''))$.

$trans_3(d, d') \equiv R_{(SK4)}(trans_2(d_1', d_1), h_2, d_3)$

where

$h_0 \equiv pre_1(d_2, (\Gamma, x'' : M_1''))$

$h_1 \equiv R_{(6.5)}(wkn(d_1', h_0), R_{(1.3)}(h_0, \Gamma))$

$h_2 \equiv (trans_2(sub_K(wkn(d_2, h_0), h_1), d_2')$

(e) d rule(SK3), d' rule(SK3)

$$d \ \equiv \ \frac{\Gamma \overset{d_1}{\vdash} M_1' = M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [x'/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'},$$

where $c \equiv [f : (x : M_1)M_2][x' : M_1']c_2(f(x'))$.

$$d' \equiv \frac{\Gamma \overset{d_1'}{\vdash} M_1'' = M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [x''/x']M_2' <_{c_2'} M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \mathbf{kind}}{\Gamma \vdash (x' : M_1')M_2' <_{c'} (x'' : M_1'')M_2''},$$

where $c' \equiv [f' : (x : M_1')M_2'][x'' : M_1'']c_2'(f'(x''))$.

$trans_3(d, d') \equiv R_{(SK_3}(R_{(2.3)}(d_1', d_1), h_2, d_3)$

where

$h_0 \equiv pre_1(d_2', (\Gamma, x'' : M_1''))$

$h_1 \equiv R_{(3.1)}(R_{(1.3)}(h_0,\Gamma), wkn(d_1', h_0))$

$h_2 \equiv trans_3(sub_K(wkn(d_2,h_0),h_1),d_2')$

(f) d rule(SK3), d' rule(SK4)

$$d \;\equiv\; \frac{\Gamma \overset{d_1}{\vdash} M_1' = M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [x'/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x:M_1)M_2 <_c (x':M_1')M_2'},$$
where $c \equiv [f:(x:M_1)M_2][x':M_1']c_2(f(x'))$.

$$d' \equiv \frac{\Gamma \overset{d_1'}{\vdash} M_1'' <_{c_1'} M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [c_1'(x'')/x']M_2' <_{c_2'} M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \mathbf{kind}}{\Gamma \vdash (x':M_1')M_2' <_{c'} (x'':M_1'')M_2''}$$
where $c' \equiv [f':(x':M_1')M_2'][x'':M_1'']c_2'(f'(c_1'(x'')))$.

$trans_3(d,d') \equiv R_{(SK4)}(trans_2(d_1',d_1),h_2,d_3)$

where

$h_0 \equiv pre_1(d_2,(\Gamma,x'':M_1''))$

$h_1 \equiv R_{(6.5)}(wkn(d_1',h_0),R_{(1.3)}(h_0,\Gamma))$

$h_2 \equiv trans_3(sub_K(wkn(d_2,h_0),h_1),d_2')$

(g) d rule(SK4), d' rule(SK2)

$$d \equiv \frac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x:M_1)M_2 <_c (x':M_1')M_2'}$$
where $c \equiv [f:(x:M_1)M_2][x':M_1']c_2(f(c_1(x')))$.

$$d' \equiv \frac{\Gamma \overset{d_1'}{\vdash} M_1'' <_{c_1'} M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [c_1'(x'')/x']M_2' = M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \mathbf{kind}}{\Gamma \vdash (x':M_1')M_2' <_{c'} (x'':M_1'')M_2''},$$
where $c' \equiv [f':(x':M_1')M_2'][x'':M_1'']f'(c_1'(x''))$.

$trans_3(d,d') \equiv R_{(SK4)}(trans_3(d_1',d_1),h_2,d_3)$

where

$h_0 \equiv pre_1(d_2,(\Gamma,x'':M_1''))$

$h_1 \equiv R_{(6.5)}(wkn(d_1',h_0),R_{(1.3)}(h_0,\Gamma))$

$h_2 \equiv trans_2(sub_K(wkn(d_2,h_0),h_1),d_2')$

(h) d rule(SK4), d' rule(SK3)

$$d \equiv \frac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \mathbf{kind}}{\Gamma \vdash (x:M_1)M_2 <_c (x':M_1')M_2'}$$
where $c \equiv [f:(x:M_1)M_2][x':M_1']c_2(f(c_1(x')))$.

$$d' \equiv \frac{\Gamma \overset{d_1'}{\vdash} M_1'' = M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [x''/x']M_2' <_{c_2'} M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \mathbf{kind}}{\Gamma \vdash (x':M_1')M_2' <_{c'} (x'':M_1'')M_2''},$$
where $c' \equiv [f':(x:M_1')M_2'][x'':M_1'']c_2'(f'(x''))$.

$trans_3(d,d') \equiv R_{(SK4)}(trans_1(d_1,R_{(2.2)}d_1'),h_2,d_3)$

where

$h_0 \equiv pre_1(d_2, (\Gamma, x'' : M_1''))$

$h_1 \equiv R_{(3.1)}(R_{(1.3)}(h_0, \Gamma), wkn(d_1', d_0))$

$h_2 \equiv trans_3(sub_K(wkn(d_2, h_0), h_1), d_2')$

(i) d rule(SK4), d' rule(SK4)

$$d \equiv \frac{\Gamma \overset{d_1}{\vdash} M_1' <_{c_1} M_1 \quad \Gamma, x' : M_1' \overset{d_2}{\vdash} [c_1(x')/x]M_2 <_{c_2} M_2' \quad \Gamma, x : M_1 \overset{d_3}{\vdash} M_2 \ \textbf{kind}}{\Gamma \vdash (x : M_1)M_2 <_c (x' : M_1')M_2'}$$

where $c \equiv [f : (x : M_1)M_2][x' : M_1']c_2(f(c_1(x')))$.

$$d' \equiv \frac{\Gamma \overset{d_1'}{\vdash} M_1'' <_{c_1'} M_1' \quad \Gamma, x'' : M_1'' \overset{d_2'}{\vdash} [c_1'(x'')/x']M_2' <_{c_2'} M_2'' \quad \Gamma, x' : M_1' \overset{d_3'}{\vdash} M_2' \ \textbf{kind}}{\Gamma \vdash (x' : M_1')M_2' <_{c'} (x'' : M_1'')M_2''}$$

where $c' \equiv [f' : (x' : M_1')M_2'][x'' : M_1'']c_2'(f'(c_1'(x'')))$.

$trans_3(d, d') \equiv R_{(SK4)}(trans_3(d_1', d_1), h_2, d_3)$

where

$h_0 \equiv pre_1(d_2, (\Gamma, x'' : M_1''))$

$h_1 \equiv R_{(6.5)}(wkn(d_1', h_0), R_{(1.3)}(h_0, \Gamma))$

$h_2 \equiv trans_3(sub_K(wkn(d_2, h_0), h_1), d_2')$