Durham E-Theses

Coherence and transitivity in coercive subtyping

Luo, Yong

How to cite:

Luo, Yong (2004) *Coherence and transitivity in coercive subtyping*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/3025/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, Durham University, University Office, Old Elvet, Durham DH1 3HP
e-mail: e-theses.admin@dur.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

Coherence and Transitivity in Coercive Subtyping

A copyright of this thesis rests with the author. No quotation from it should be published without his prior written consent and information derived from it should be acknowledged.

Yong Luo

A Thesis presented for the degree of

Doctor of Philosophy

Department of Computer Science

University of Durham

September 2004

2 8 FEB 2005

Coherence and Transitivity in Coercive Subtyping

Yong Luo

Submitted for the degree of Doctor of Philosophy
September 2004

Abstract

The aim of this thesis is to study coherence and transitivity in coercive subtyping. Among other things, coherence and transitivity are key aspects for a coercive subtyping system to be consistent and for it to be implemented in a correct way. The thesis consists of three major parts.

First, I prove that, for the subtyping rules of some parameterised inductive data types, coherence holds and the normal transitivity rule is admissible.

Second, the notion of weak transitivity is introduced. The subtyping rules of a large class of parameterised inductive data types are suitable for weak transitivity, but not compatible with the normal transitivity rule.

Third, I present a new formulation of coercive subtyping in order to combine incoherent coercions for the type of dependent pairs. There are two subtyping relations in the system and hence a further understanding of coherence and transitivity is needed. This thesis has the first case study of combining incoherent coercions in a single system.

The thesis provides a clearer understanding of the subtyping rules for parameterised inductive data types and explains why the normal transitivity rule is not admissible for some natural subtyping rules. It also demonstrates that coherence and transitivity at type level can sometimes be very difficult issues in coercive subtyping. Besides providing theoretical understanding, the thesis also gives algorithms for implementing the subtyping rules for parameterised inductive data types.

Acknowledgements

I learned the idea of coercive subtyping mainly from Professor Zhaohui Luo, my supervisor, and the work I have done in the thesis is a success. I would like to express my deep appreciation to him, not only for his supervision and guidance in my research, but also for his support in applying for a studentship and his help with many personal matters.

Thanks also to all members of the Computer Assisted Reasoning Group at Durham. They kindly answered many of my questions in the early stage of my research.

I would like to thank Professor Sergei Soloviev who gave me helpful suggestions and comments on my study. Thanks also to Professor Simon Thompson, my colleagues Paul Callaghan and Edwin Brady, and my friend Tamzin Lafford. They carefully read the thesis and gave me many suggestions.

A special word of thanks goes to my wife and my family in China. They love me and encourage me always.

I am deeply grateful for my friends at Durham. They love me and help me like brothers and sisters. I have a valuable time of my life because of them.

Declaration

I declare that this thesis was composed by myself, and the work reported herein is my own unless explicitly declared otherwise. Some parts of the work have already been published in [LL01, LLS02, LL04a, LL04b].

Copyright © 2004 by Yong Luo.
"The copyright of this thesis rests with the author. No quotations from it should be published without the author's prior written consent and information derived from it should be acknowledged".

Contents

Abstract 2

Acknowledgements 3

Declaration 4

Chapter 1. Introduction 8
- 1.1. Formal Verification 8
- 1.2. Type Theory for Computer Science 9
- 1.3. Coercive Subtyping for Abbreviation 10
- 1.4. Coherence in Coercive Subtyping 11
- 1.5. Transitivity and Substitution in Coercive Subtyping 12
- 1.6. Major Contributions 13
- 1.7. Structure of the thesis 15
- 1.8. Related work 16
  - 1.8.1. Subtyping in programming languages 16
  - 1.8.2. Coercions in type theory 19

Chapter 2. UTT 22
- 2.1. Logical Framework 22
  - 2.1.1. The inference rules of LF 22
  - 2.1.2. Specifying type theories in LF 25
  - 2.1.3. Computational equality 26
- 2.2. SOL: the internal logical mechanism 27
- 2.3. Inductive data types 29
- 2.4. ST-form: a subset of inductive data types 35
- 2.5. Related work and Extensional type theory 41
  - 2.5.1. Related work on UTT 41
  - 2.5.2. Extensional type theory 42

Chapter 3. Coercive Subtyping 44
- 3.1. Basic idea 44

3.2. A formal presentation 45
3.2.1. The system $T[\mathcal{R}]_0$ 46
3.2.2. Coherence of the subtyping rules 47
3.2.3. The system of $T[\mathcal{R}]$ 47
3.3. The problems 49
3.4. Well-defined coercions 50
3.5. Subtyping rules 51

Chapter 4. Coherence and Transitivity 57
4.1. Coherence of $T[\mathcal{R}]_0$ 57
4.2. Admissibility of Substitution and Transitivity 61
4.3. Algorithm for the coercion search 67
4.3.1. Algorithm $Alg(\Gamma, M_1, M_2)$ for $T[\mathcal{R}]_0$ 67
4.3.2. Soundness and Completeness 69
4.3.3. Decidability of the Coercion Search in $T[\mathcal{R}]_0$ 69
4.4. Subtyping rules for ST-form 69

Chapter 5. Weak Transitivity 73
5.1. A problem with transitivity 73
5.2. Weak transitivity 75
5.2.1. Meta-level equality requirement 75
5.2.2. Coercion dependency 77
5.3. Weak transitivity schemata 79
5.4. General subtyping rules for WT-schemata 82
5.5. Coherence 96
5.6. Admissibility of Substitution and Weak Transitivity 99
5.7. Extension of WT-schemata 101
5.8. Discussion: new computation rules 102
5.8.1. New computation rule for lists 104
5.8.2. New computation rules in general 105

Chapter 6. Combining Incoherent Coercions for Σ-types 106
6.1. The Coherence Problem 106
6.1.1. Subtyping rules for Σ-types 106
6.1.2. A counter example 107
6.1.3. Informal explanation of the solution 108
6.2. A formal presentation 108
6.2.1. A new subtyping relation 109

6.2.2. The systems $T[\mathcal{R}\pi_1]_0$ and $T[\mathcal{R}\pi_1]$ 109
6.3. New subtyping rules for inductive types 112
6.4. Coherence of $T[\mathcal{R}\pi_1]_0$ 114
6.5. Admissibility of substitution and transitivity 118
6.6. Algorithm for the coercion search in $T[\mathcal{R}\pi_1]_0$ 125
6.6.1. Algorithm $ALG(\Gamma, M_1, M_2)$ for $T[\mathcal{R}\pi_1]_0$ 125
6.6.2. Soundness and Completeness 127
6.6.3. Decidability of the Coercion Search in $T[\mathcal{R}\pi_1]_0$ 129
6.7. Discussions 129
6.7.1. Side conditions 129
6.7.2. New computation rules 130
6.7.3. Combining incoherent coercions in general 130

Chapter 7. Conclusion 131
7.1. Summary 131
7.2. Implementation 132
7.3. Future work 133

Bibliography 135

Index 141

CHAPTER 1

Introduction

This chapter introduces the area of interest and informally explains the significance of the work and major contributions. It also includes the structure of the thesis, summarising the material that the other chapters will cover. Other work related to the thesis is at the end of the chapter.

1.1. Formal Verification

Computers have become indispensable in our life and itself changes everyday. We use it to perform fast computation, to communicate, to conduct sophisticated control, and so on. Thousands of programs or software are developed and produced everyday. However, how do we know whether a program will behave as intended? Or, how do we check the correctness of a program? Testing is a common method used by every computer programmer, but has its obvious limitations because test data can only be finite. As complexity increases, the reliability of testing very much depends on a careful choice of input data, and it becomes difficult to carry out the test by hand, case after case.

A complementary and more rigorous method is *formal verification.* Computer scientists want to use computers to formally (mathematically) verify the correctness of programs. Formal verification can often help to detect logical problems, missed cases because of carelessness, and other bugs. Therefore, it can significantly increase the confidence in the correctness of programs. *Model-checking* is one of the formal verification techniques. It can automatically verify finite-state concurrent systems [CGL94]. There also are many verification tools often called *interactive theorem provers* in which not only finite-state systems can be verified but also infinite-state ones. Some interactive theorem provers are based on *simply-typed λ-calculus* [Chu40], such as *HOL* [GM93], *Isabelle* [Pau93] and *PVS* [ORS92]. Some are based on type theories and more details will be given in the next section.

1.2. Type Theory for Computer Science

Why is (constructive) type theory a good foundation for computer science? We can at least give three reasons here. First, type theory has dependent types and hence it has more expressive power than simply-typed systems. For example, the type of vectors is a dependent type and can be easily defined in a type theory but not in a simply-typed system. Second, type theory is a high level (functional) programming language. Its computation and operational semantics are simple and clear – reducing *well-typed* terms to normal (or canonical) form. Third, type theory has its internal logic and reasoning can be carried out. The activity of proving a theorem in type theory coincides with that of writing a program that satisfies a given typing specification in the well-known principle of *propositions-as-types.* Therefore, for computer scientists, type theory provides a framework in which both programming and reasoning can proceed [Tho91, NPS90, Luo94].

There are various type theories with various logics such as *Pure Type Systems* [Bar92], *Martin-Löf's Intuitionistic Type Theory* [ML84], *Calculus of Inductive Constructions* [PM93], *Extended Calculus of Constructions (ECC)* [Luo90] and a unifying theory for dependent types *(UTT)* [Luo94]. There also are various *Logical Frameworks* to specify type theory, such as *Martin-Löf's Logical Framework* [NPS90], *Edinburgh Logical Framework* [HHP92] and *PAL+* - a λ-free logical framework [Luo03]. Many proof systems based on type theories have been developed and widely used by formal reasoning communities. *NuPRL* [C^+86] is based on *Martin-Löf's Intuitionistic Type Theory. Coq* [B^+00] is an implementation of the *Calculus of Inductive Constructions. Lego* [LP92] is based on the *Extended Calculus of Constructions (ECC). Plastic* [CL01] is based on *Logical Framework* and *UTT* can be specified in it. In the libraries of these proof systems, thousands of mathematical theorems and computer programs have been proved and verified.

However, in many cases, proofs are very tedious and users have to fill in every tiny detail carefully. Especially, when formal proofs become very large, too much detail will cause proofs to be unreadable for human beings and will cost a lot of time. More seriously, no one would like to use any too-costly proof system in practice. So, a very important task is to make proofs more readable and omit unnecessary

details. Towards this direction, subtyping has been studied as an inheritance or abbreviation mechanism in type theory [BF99, Luo99]. In the next section, I will give a brief survey of the study of subtyping in literature and explain why *Coercive Subtyping* is a powerful abbreviation mechanism in type theory.

1.3. Coercive Subtyping for Abbreviation

Intuitively, a type in type theory can be understood as a set consisting of its canonical objects. For example, the type N of natural numbers consists of all the natural numbers as its canonical objects. Some inductive types have parameters. For example, $List(A)$ (the type of lists of objects of type A) is parameterised by type A. $\Sigma(A, B)$ (type of dependent pairs) is parameterised by type A and a family of types B, and if a is an object of A and b is an object of type $B(a)$ then a pair of a and b is an object of $\Sigma(A, B)$.

Some of the subtyping systems are based on the intuition of "subtypes as subsets". A subtype is a collection of canonical objects from its supertype. For example, one can create a supertype by adding new constructors in an existing inductive type [Pol97]. A similar study on constructor subtyping is in [BF99] and [BvR00], and the basic idea is that A is a subtype of B if the constructors in A form a subset of those in B. However, both of the approaches would exclude very simple examples such as $List(A)$ is a subtype of $List(B)$ if A is a subtype of B.

Coercive subtyping is based on a different concept of subtyping, in which a coercion is regarded as evidence that one type is a subtype of another. It offers a nice formulation so that subtyping can be understood naturally and uniformly in a single framework. In particular, *coercive subtyping* is a simple and powerful framework to handle subtyping and inheritance relations between inductive data types. For example, one can simply give subtyping rules to express that $List(A)$ is a subtype of $List(B)$ if A is a subtype of B. Another example often mentioned in literature is the component-wise subtyping rules for the type of dependent pairs, that is, $\Sigma(A, B)$ is a subtype of $\Sigma(A', B')$ if A is a subtype of A' and B is a sub-family of B'.

Coercive subtyping is also regarded as an abbreviation mechanism in type theory. With implicit coercions, terms will become more readable and their meaning clearer. Here is a sample to give a flavour of such an abbreviation mechanism; there is no need for detailed understanding for now.

Example 1.3.1 *Suppose that we have two inductive types in type theory, Even (the type of even numbers) and N (the type of natural numbers). Since Even is a subtype of N, List(Even) is a subtype of List(N). For any function operator f with domain List(N) and any object x of List(Even), f(x) is well-typed in the framework of coercive subtyping and it is an abbreviation of a very long term*[1].

This abbreviation mechanism not only make terms significantly shorter and more readable but also captures the natural understanding of subtyping.

A significant use of coercions as an abbreviation mechanism is in Anthony Bailey's thesis [Bai98]. In the formalisation of the constructive version of the fundamental theorem of Galois Theory, he employed three kinds of coercions and extended the system Lego with coercion synthesis (called *LEGOwcs*).

1.4. Coherence in Coercive Subtyping

The meaning of a term in any logical system must be clear and precise. Ambiguity is not allowed. It must be completely determined and be understood in the same way by all human beings at any time in the same logical system. Coercive subtyping is an abbreviation mechanism in type theory, so we must have a coherent understanding for an abbreviated expression. In other words, there is a vital requirement that any abbreviated term in coercive subtyping represents a unique expanded term at any time. The notion of *coherence* in coercive subtyping guarantees this requirement, which essentially says that coercions between

[1]

$$
\begin{aligned}
f(x) \quad = \quad & f(\mathcal{E}_{List}(Even, [l : List(Even)]List(N), nil(N), \\
& [a : Even][l : List(Even)][l' : List(N)] \\
& cons(N, \mathcal{E}_{Even}([n : Even]N, 0, \\
& [n : Even][m : N]S(S(m)), a), l'), x))
\end{aligned}
$$

any two types must be (computationally) unique. If there are two coercions c_1 and c_2 from type A to B (*i.e.* $A <_{c_1} B$ and $A <_{c_2} B$), then c_1 and c_2 must be computationally equal. For any object x of type A and function operator f with domain B, $f(x)$ is an abbreviation of $f(c_1(x))$ and $f(c_2(x))$. Since c_1 and c_2 are computationally equal, $f(c_1(x))$ and $f(c_2(x))$ are computationally equal and regarded as the same in type theory.

In general, coherence is not decidable, especially when there are infinitely many coercions as introduced by parameterisation. It is impossible to check coherence in many cases unless we can prove it. One of the major contributions in this thesis is to study proving coherence at type level when infinite coercions are generated by the natural subtyping rules of parameterised inductive data types.

Some very useful coercions cannot be put together directly because they are incoherent. This prevents them from being used together in a uniform framework although they are coherent separately. Another major contribution regarding coherence in this thesis is to study how to combine incoherent coercions for the type of dependent pairs.

1.5. Transitivity and Substitution in Coercive Subtyping

For any subtyping system, we naturally have *transitivity* and *substitution*. The meaning of transitivity is that, if A is a subtype of B and B is a subtype of C then A must be a subtype of C. The meaning of substitution is that, if type $B(x)$ is a subtype of $C(x)$ for any x of type A, then for any concrete object a of A, $B(a)$ is a subtype of $C(a)$. Because of the difficulties of implementing the transitivity rule and substitution rule, an important issue with any subtyping system is that of admissibility or elimination of transitivity and substitution.

For coercive subtyping, proving the admissibility of substitution is straightforward for most of the subtyping rules considered in this thesis. So, I will concentrate on the issue of the admissibility of transitivity. The meaning of *transitivity* in coercive subtyping is that, if there is a coercion c_1 from type A to B (*i.e.* $A <_{c_1} B$) and a coercion c_2 from B to C (*i.e.* $B <_{c_2} C$) , then there is a coercion c_3 from A to C (*i.e.* $A <_{c_3} C$). The normal transitivity rule also requires that $c_3 = c_2 \circ c_1$ (computational equality). For many subtyping rules, for

example, the component-wise subtyping rules for the type of dependent pairs, the transitivity rule is admissible when one uses the projection operators to define coercions. However, the requirement of $c_3 = c_2 \circ c_1$ (computational equality) is sometimes too strong in intensional type theories. For some parameterised inductive data types together with their natural subtyping rules, the transitivity rule fails to be admissible or eliminatable. So, we introduce a new concept – *Weak Transitivity* that only requires that c_3 and $c_2 \circ c_1$ are extensionally equal, without compromising coherence (computational uniqueness). Many natural subtyping rules, for example, the subtyping rule for lists, are suitable for weak transitivity.

Through our investigation, we also found out that neither the normal transitivity rule nor the weak transitivity rule (*i.e.* no matter which equality is chosen) can be admissible when we combine some natural subtyping rules, for example the subtyping rules for the types of dependent pairs and lists. This leads us to more fundamental research that is important for coercive subtyping as well as for type theory itself. If we introduce new computation rules for parameterised inductive types and add them to the original type theory, then the normal transitivity rule is admissible for the extended type theory in which some important meta-properties such as Strong Normalisation and Church-Rosser are assumed and believed to be true.

In the case that there is more than one subtyping relation, new transitivity rules are introduced in order to capture the meaning of transitivity, that is, if there are coercions from type A to B and from B to C then there must be a coercion from A to C.

1.6. Major Contributions

After briefly introducing the two important issues in coercive subtyping, coherence and transitivity, I summarise the major contributions of the thesis in this section. The thesis focuses on the coercions between parameterised inductive data types and shows the serious problems with these coercions concerning coherence and transitivity. New techniques are developed to solve these problems. The main work in this thesis can be divided into three parts.

1. In the first part, we consider the normal transitivity rule which basically says that, if $A <_{c_1} B$ and $B <_{c_2} C$ where A, B, C are types, then $A <_{c_3} C$ for some c_3 and $c_3 = c_2 \circ c_1$ (*i.e.* c_3 and $c_2 \circ c_1$ are computationally equal). In general, the coercions between parameterised inductive data types are inductively defined by means of case analysis. However, the coercions defined in this way will cause the normal transitivity rule not to be admissible and, if adding it into the system, coherence fails to be satisfied. Fortunately, for some parameterised inductive data types, coercions can be defined in a nice way where some special function operators are used. Coherence holds and the normal transitivity rule is admissible for these coercions. To make this clear, we choose two typical and representative data types to demonstrate how the coercions are defined and how the coherence and admissibility of the normal transitivity rule are proved. One example is the type of dependent pairs and the other is the type of dependent functions. A common factor of these two data types is that they have only one constructor and some special function operators over them can be defined. One doesn't have to define the coercions inductively and instead, can define them by using the special function operators. In the end, we discuss the results more generally and demonstrate how coercions are defined for those parameterised inductive data types that have only one constructor.
2. The second part starts from examples to make the problems clear, that is, for certain inductive data types such as lists, coercions have to be defined inductively and the normal transitivity rule is not admissible. Through a close look at key examples, we shall get a better understanding of the coercions between parameterised inductive data types in general. We introduce a new notion, *Weak Transitivity,* which basically says that, if $A <_{c_1} B$ and $B <_{c_2} C$ where A, B, C are types, then $A <_{c_3} C$ for some c_3. The meta-level equality requirement is that c_3 is extensionally equal to $c_2 \circ c_1$. This part will give a clear characterisation of different combinations of subtyping rules by means of inductive schemata. We prove that, for a large class of inductive data types with their subtyping rules, coherence and weak transitivity hold.

3. In the third part, we study how to combine incoherent coercions for the type of dependent pairs. There are at least two sets of subtyping rules for the type of dependent pairs; one is the component-wise subtyping rules (*i.e.* $\Sigma(A, B)$ is a subtype of $\Sigma(A', B')$ if A is a subtype of A' and B is a sub-family of B') and the other is the subtyping rule of its first projection (*i.e.* $\Sigma(A, B)$ is a subtype of A). A counter example is given to show that these two sets of subtyping rules are incoherent if they are put together directly. Our solution to this coherence problem is basically, by introducing a new subtyping relation and giving a new formulation of coercive subtyping, to ensure that there is only one coercion (with respect to computational equality) between any two types (if there is a coercion at all). This new formulation not only satisfies coherence requirements but also enjoys other properties, particularly, the admissibility of substitution and transitivity.

To summarise, the thesis provides not only the proofs concerning coherence and transitivity but also clearer understanding of the problems with the subtyping rules for parameterised inductive data types. The problems identified here have not been realised before in the literature except in some of my publications in collaboration with Zhaohui Luo and Sergei Soloviev. The discovery of these problems also leads us to fundamental future work on the extension of type theory by adding new computation rules for parameterised inductive types so that the natural subtyping rules for all the parameterised inductive types can be uniformly used together.

1.7. Structure of the thesis

In Chapter 2, I give a formal and detailed presentation of Zhaohui Luo's UTT. UTT is an intensional type theory specified by a typed version of Martin-Löf's logical framework. It includes an internal logic (*i.e.* second order logic, SOL) and a large class of inductive data types generated by inductive schemata. We also consider a subset of inductive data types that have only one constructor and give a general definition of function operators. These operators play important roles in the definitions of coercions later.

Chapter 3 is a formal presentation of coercive subtyping. Some important issues in the system, such as the coherence and transitivity, are discussed and made precise. It also carries a important concept of *Well-defined coercions* (WDC). The subtyping rules for the type of dependent pairs and dependent functions are presented. The different choices of defining coercions and their consequences are remarked.

In Chapter 4, I study how to prove the coherence and the admissibility of transitivity for the subtyping rules. An algorithm for the coercion search is also given. We also discuss the results more generally and demonstrate how coercions are defined for those parameterised inductive data types that have only one constructor.

In Chapter 5, I present the notion of *weak transitivity* and give a general form of subtyping rules for a large class of parameterised inductive types. Coherence and weak transitivity will be proved for these subtyping rules. At the end of this chapter, we discuss new computation rules for parameterised inductive types.

Chapter 6 studies how to combine the incoherent subtyping rules for the type of dependent pairs: the component-wise rules and the rule of its first projection. Coherence and the admissibility of transitivity for the new formulation of coercive subtyping are proved.

Finally, conclusions are presented and some related issues such as implementation of coercive subtyping and future work are discussed in Chapter 7.

1.8. Related work

In this section I briefly review some of the other pieces of work that are related to this thesis. Subtyping in computer science is not a new concept and it is traditionally understood as the notion of subsets in mathematics. However, it is fair to say that the notion of subtyping is one the most important concepts in programming languages.

1.8.1. Subtyping in programming languages

Subtyping is characteristically found in object-oriented languages and is often considered as an essential feature of the object-oriented style. Object-oriented languages take the view that all types are systematically related in a type hierarchy. Types lower in the hierarchy are somehow compatible with more general types higher in the hierarchy.

For example, a integer can be "converted" or "cast" to a floating-point number.

Besides simple subtypes, there are subtyping rules used in object-oriented languages.

- Record subtyping and Product subtyping:

$$\frac{S_1 <: T_1, ..., S_k <: T_k}{\{a_1 : S_1, ..., a_n : S_n\} <: \{a_1 : T_1, ..., a_k : T_k\}} \quad for\ 1 \leq k \leq n$$

$$\frac{S_1 <: T_1 \quad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2}$$

  where $<:$ means "is a subtype of".

- Function subtyping:

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

  One method is a valid replacement for another if it obeys the function subtyping rule. In particular, the arguments of the subtype method must be of more general types. Very few languages obey both the covariant and contravariant parts of the rule. Languages such as Java and C++ are less flexible partly due to interactions with other rules for resolving name overloading.

Subtyping is also suggested to obtain the implicit polymorphism in functional programming language. In [Mit91], a general framework based on untyped λ-calculus provides a simple semantic model of subtyping and the algorithms may be extended to allow polymorphic function declarations as in ML. Most traditional λ-calculi with subtyping include the function subtyping rule (as above), subsumption rule and transitivity rule as follows.

$$(Subsumption\ rule) \qquad \frac{t : U \quad U <: T}{t : T}$$

$$(Transitivity\ rule) \qquad \frac{S <: U \quad U <: T}{S <: T}$$

The name and form of these rules may be variant, for example, the coerce rule in [Mit91] is another version of the subsumption rule and the cut rule in [LMS00] is the transitivity rule. The subsumption and transitivity rules are not immediately suitable for implementation. Their premises mention the type U which does not appear in the conclusion. We have to find a type U in a type checking algorithm. If there is only

a finite number of Us, that is fine. However, in many cases there is an infinite number of Us, so it is unlikely to give an algorithm to find a type U. This is one of the reasons that we often need to prove the admissibility (or elimination) of the transitivity rule.

The subsumption rule has another problem when we want to reason over inductive data types. The standard reasoning principle is that if we can prove that a proposition P holds for every ***canonical*** object of an inductive data type A then P holds for every object of A. If we have the subsumption rule, the canonical objects in a subtype are also canonical objects of its supertype and how to formulate the reasoning principles may become very difficult when various inductive data types are considered.

Some systems include the *top type* (Top) and *bottom type*. Every type is a subtype of the top type and is a supertype of the bottom type. In [Reh96], the property of strong normalisation has been proven in a very simple subtyping system with the top type and bottom type. The system $F_{<:}$ [CMMS91], an extension of the system F with subtyping, also includes the top type that is a convenient technical device to recover ordinary unbounded quantification from bounded quantification. A unbounded quantification $\forall X.P$ is just an abbreviation of bounded quantification $\forall X <: Top.P$.

Subtyping between record types has also been studied in [BT98, Tas97]. One can inherit from an existing record type by adding new labels associated with their types and get a sub-record type. The essence is the same as that of the record subtyping rule in object-oriented languages. In [BF99, BvR00], constructor subtyping has been introduced in simply typed λ-calculi and dependently typed systems. An inductive type A is viewed as another inductive type B if B has more constructors than A. This idea is in line with that of the subsumption rule and the system is not well-behaved with respect to canonical objects in inductive data types. For example, $nil(Even)$ and $nil(Nat)$ are both closed normal objects of $List(Nat)$ although they represent the same thing, the empty list of $List(Nat)$.

In [LMS95, LMS00], a logic of subtyping has been studied. The idea is that one can give a logical understanding of "σ is a subtype of τ" as "σ implies τ", or more precisely as "σ entails τ" ($\sigma \vdash \tau$). The

function subtyping rule in the system is in a different form

$$\frac{\sigma' \vdash \sigma \quad \tau \vdash \tau'}{\sigma \to \tau \vdash \sigma' \to \tau'}$$

and the notion of subtyping in the system is a special case of intuitionistic implication: a proof of $\sigma \vdash \tau$.

1.8.2. Coercions in type theory

The early development of the framework of coercive subtyping is closely related to Aczel's idea in type-checking overloading methods for classes and the work on giving coercion semantics to λ-calculi with subtyping by Breazu-Tannen et al [BCGS91]. In [Luo99], Z. Luo formalised coercive subtyping, a formal extension with subtyping of dependent type theories such as Martin-Löf's type theory [NPS90] and the type theory UTT [Luo94].

The implementation of coercions

Coercion mechanisms of non-dependent coercions with certain restrictions have been implemented in both the proof development systems Lego [LPT89] and Coq [B$^+$00], by Bailey [Bai98] and Saïbi [Sai97], respectively.

Bailey has extended the Lego system with coercion synthesis (*i.e.* LEGOwcs) [Bai98] and introduced three kinds of implicit coercions; Standard coercions, which coerce an object a of type A into an object $c(a)$ of type B; Kind coercions, which coerce an object a of type A into a kind $c(a)$; and Π-coercions, which coerce an object a of type A into a function $c(a)$, where c is a coercion. Coercions in LEGOwcs are represented by a finite graph with parameters, so it is fairly easy to guarantee coherence and transitivity.

Saïbi has also introduced an inheritance mechanism and implements coercions in Coq. The use of this mechanism, with some other facilities such as the implicit argument synthesis and infix notions, makes mathematical statements more readable. He has introduced two abstract classes; SORTCLASS, which allows us to write $x : A$ when A is not a type, but can be seen in a certain sense as a type such as set, group and category; and FUNCLASS, which 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.

Callaghan of the Computer Assisted Reasoning Group at Durham has implemented Plastic [Cal99, CLP01, CL01], a proof assistant that supports logical framework and coercive subtyping with a mixture of simple coercions, parameterised coercions and dependent coercions.

Theoretical study on coercive subtyping

Some important meta-theoretical aspects of coercive subtyping (for non-dependent coercions) have been studied. In particular, the results on conservativity and on transitivity elimination for kinds have been proved in [JLS98, SL02]. The conservativity result says, intuitively, that every judgement that is derivable in the theory with coercive subtyping and that does not contain coercive applications is derivable in the original type theory. Furthermore, for every derivation in the theory with coercive subtyping, one can always insert coercions correctly to obtain a derivation in the original type theory.

The main result of [SL02] is that coherence of basic subtyping rules does imply conservativity, under certain conditions. (These conditions are satisfied, for example, for the type theory UTT). The proof of the conservativity theorem consists of the following three major parts:

1. Lemmas about general meta-theoretical properties of the theory with coercive subtyping;
2. Transitivity elimination in the calculus with subtyping and subkinding but without coercive application and definition rules;
3. The proof of the well-definedness (totality) of a coercion completion which maps derivations of the full theory into the calculus without coercive application and definition rules.

These results not only justify the adequacy of the theory from the proof-theoretical consideration, but also provide the proof-theoretical basis for implementation of coercive subtyping.

An important study on coercive subtyping is Dependent Coercions [LS99]. A dependent coercion is a function from a type to a family of types; informally, the supertype is the union of the types in the family. It is different from parameterised coercions. The dependent coercions

and non-dependent coercions have the same meta-theoretical results, that is, the conservativity theorem holds.

CHAPTER 2

UTT

In this chapter, we give a formal and detailed presentation of Zhaohui Luo's UTT. UTT is an intensional type theory specified by a typed version of Martin-Löf's logical framework. It includes an internal logic (*i.e.* second order logic, SOL) and a large class of inductive data types generated by inductive schemata. Related work on UTT and *Extensional Type Theory* will be discussed at the end of this chapter.

2.1. Logical Framework

Logical frameworks arise because one wants to create a single framework, which is a kind of meta-logic or universal logic, which is itself implementable and in which the logics can be represented. The Edinburgh Logical Framework [HHP87] is intended to provide such a means of presentation. It comprises a formal system yielding a formal means of presentation of logical systems, and an informal method of finding such presentations. An important part in presenting logics is played by a *judgements-as-types* principle, which can be regarded as the meta-theoretical analogue of the well-known *propositions-as-types* principles [CF58, dB80, How80]. Martin-Löf's logical framework [NPS90] has been developed by Martin-Löf to present his intensional type theory. In UTT [Luo94], Luo proposed a typed version of Martin-Löf's logical framework (LF), in which untyped functional operations of the form $(x)k$ are replaced by typed $[x : K]k$.

In this section, we consider the typed version of Martin-Löf's logical framework, and how to use it as a meta-language to specify type theories.

2.1.1. The inference rules of LF

First, there are five forms of judgements in LF, as follows:

- Γ *valid*, which asserts that Γ is a valid context;
- $\Gamma \vdash K$ *kind*, which asserts that K is a 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; and
- $\Gamma \vdash K = K'$, which asserts that K and K' are equal kinds.

The inference rules of LF are given in Figure 2.1.1. There is a special kind *Type* in LF, each of whose objects A generates a kind $El(A)$. When specifying a type theory in LF, *Type* corresponds to the conceptual universe of types of the type theory to be specified, and for any type A, an object of kind *Type*, kind $El(A)$ corresponds to the collection of objects of type A.

Definition 2.1.1 ***(types, kinds*** and ***small kinds)*** *A is called a Γ-type if $\Gamma \vdash A : Type$, K is called a Γ-kind if $\Gamma \vdash K$ 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.*

Notation 2.1.2 *We shall use the following notational conventions:*

- When no confusion may occur, we shall often omit the *El*-operator in LF to write, for example, A for $El(A)$, $(x : A)B$ for $(x : El(A))El(B)$, $\Gamma \vdash a = b : A$ for $\Gamma \vdash a = b : El(A)$ etc.
- $FV(M)$ is the set of free variables in M. For a context Γ, if Γ is $x_1 : K_1, ..., x_n : K_n$ then $FV(\Gamma) = \{x_1, ..., x_n\}$.
- We shall write $(K)K'$ for $(x : K)K'$ when x does not occur free in K'. For application of a functional operator, we shall write $f(k_1, ..., k_n)$ for $f(k_1)...(k_2)$.
- Functional composition: for $f : (K_1)K_2$ and $g : (y : K_2)K_3[y]$, define $g \circ f =_{df} [x : K_1]g(f(x)) : (x : K_1)K_3[f(x)]$, where x does not occur free in f or g.
- Substitution: as usual, $[N/x]M$ stands for the expression obtained from M and substituting N for the free occurrences of variables x in M, defined as usual with possible changes of bound variables; informally, we sometimes use $M[x]$ to indicate that variable x may occur free in M and subsequently write $M[N]$ for $[N/x]M$, when no confusion may occur.

  If M is a sequence $< M_1, ..., M_n >$, we often write $[N/x]M$ for the sequence $< [N/x]M_1, ..., [N/x]M_n >$

  We also write $[\overline{N}/\overline{x}]M$ for $[N_1/x_1, ..., N_n/x_n]M$ where $\overline{N}$ and $\overline{x}$ are

Contexts and assumptions:

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

General equality rules:

$$\frac{\Gamma \vdash K\ kind}{\Gamma \vdash K = K} \qquad \frac{\Gamma \vdash K = K'}{\Gamma \vdash K' = K} \qquad \frac{\Gamma \vdash K = K' \quad \Gamma \vdash K' = K''}{\Gamma \vdash K = K''}$$

$$\frac{\Gamma \vdash k : K}{\Gamma \vdash k = k : K} \qquad \frac{\Gamma \vdash k = k' : K}{\Gamma \vdash k' = k : K} \qquad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash k' = k'' : K}{\Gamma \vdash k = k'' : K}$$

Equality typing rules:

$$\frac{\Gamma \vdash k : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k : K'} \qquad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k = k' : K'}$$

Substitution rules:

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

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

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

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

The kind *Type*:

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

Dependent product kinds:

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

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

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

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

FIGURE 2.1.1. The inference rules of LF

sequences $< N_1, ..., N_n >$ and $< x_1, ..., x_n >$ which have the same length n ($n \in \omega$).

- Context equality: for $\Gamma \equiv x_1 : K_1, ..., x_n : K_n$ and $\Gamma' \equiv x_1 : K_1', ..., x_n : K_n'$, we shall write $\vdash \Gamma = \Gamma'$ for the sequence of judgements $\vdash K_1 = K_1'$, ..., $x_1 : K_1, ..., x_{n-1} : K_{n-1} \vdash K_n = K_n'$.

2.1.2. Specifying type theories in LF

In general, a specification of a type theory in LF consists of a collection of declarations of new constants and a collection of computation rules. Formally, we declare a new constant k of kind K by writing

$$k : K$$

which represents that we add a new inference rule

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

into the type theory (specified by means of LF). For a kind K which is either *Type* or of the form $El(A)$, we assert a computation rule by writing

$$k = k' : K$$

which represents that we add a new inference rule which is of the form

$$\frac{\mathit{premises}}{\Gamma \vdash k = k' : K}$$

into the type theory.

Example 2.1.3 *We can introduce the type of natural numbers by declaring the following constants:*

$$\begin{array}{rcl} N & : & Type \\ O & : & N \\ S & : & (N)N \\ \mathcal{E}_N & : & (C : (N)Type)(c : C(O)) \\ & & (f : (n : N)(C(n))C(S(n))) \\ & & (n : N)(C(n)) \end{array}$$

and asserting the following two computation rules:

$$\begin{aligned}\mathcal{E}_N(C,c,f,O) &= c : C(O)\\ \mathcal{E}_N(C,c,f,S(n)) &= f(n,\mathcal{E}_N(C,c,f,n)) : C(S(n))\end{aligned}$$

which represent that we add the following six new inference rule (Figure 2.1.2) into type theory.

Introduction rules:

$$\frac{\Gamma\ valid}{\Gamma \vdash N : Type} \qquad \frac{\Gamma\ valid}{\Gamma \vdash O : N} \qquad \frac{\Gamma\ valid}{\Gamma \vdash S : (N)N}$$

Elimination rule:

$$\frac{\Gamma\ valid}{\Gamma \vdash \mathcal{E}_N : (C : (N)Type)(c : C(O))(f : (n : N)(C(n))C(S(n)))(n : N)(C(n))}$$

Computation rules:

$$\frac{\Gamma \vdash C : (N)Type \quad \Gamma \vdash c : C(O) \quad \Gamma \vdash f : (n : N)(C(n))C(S(n))}{\Gamma \vdash \mathcal{E}_N(C,c,f,O) = c : C(O)}$$

$$\frac{\Gamma \vdash C : (N)Type \quad \Gamma \vdash c : C(O) \quad \Gamma \vdash f : (n : N)(C(n))C(S(n)) \quad \Gamma \vdash n : N}{\Gamma \vdash \mathcal{E}_N(C,c,f,S(n)) = f(n,\mathcal{E}_N(C,c,f,n)) : C(S(n))}$$

FIGURE 2.1.2. The inference rules for natural numbers

2.1.3. Computational equality

We shall say that two objects k and k' of the same kind K in the type theory UTT are *computationally equal* if for some valid context Γ, the judgement $\Gamma \vdash k = k' : K$ is derivable in UTT.

Note 2.1.4 *In the intensional type theory UTT, every well-typed term has a unique normal form. If* $\Gamma \vdash k = k' : K$ *is derivable,* k *and* k' *can be computed to a same normal form (or weak head normal form). Therefore, we can say that* k *and* k' *are* computationally equal *if they are well-typed and have the same normal form.*

If two objects of same kind are $\beta\eta$-convertible, we say that they are *definitionally equal.* Since computation consists of not only $\beta\eta$-reduction but also reduction rules introduced by asserting computation rules for inductive data types, if two objects are definitionally equal, they are of course computationally equal.

Notation 2.1.5 *We shall often write $M \equiv N$ to indicate that M and N are syntactically equal with respect to α-conversion.*

We shall say that k is *not computationally equal* to k' if for any Γ and K, the judgement $\Gamma \vdash k = k' : K$ is not derivable in the type theory UTT, in other words, k and k' cannot be computed to the same normal form.

2.2. SOL: the internal logical mechanism

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

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

and asserting the following computation rule:

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

The logical universe *Prop* is impredicative since universal quantification $\forall(A, P)$ can be formed for any type A and predicate P over A. In particular, A can be *Prop* itself or more complex.

The usual application operator can be defined as

$$\begin{array}{rcl} App & =_{df} & [A : Type][P : (A)Prop][F : Prf(\forall(A, P))][a : A] \\ & & \mathcal{E}_\forall(A, P, [G : Prf(\forall(A, P))]P(a), \\ & & [g : (x : A)Prf(P(x))]g(a), F) \end{array}$$

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

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

Notation 2.2.1 *For universal quantification, when no confusion may occur, we shall often write* $\forall x : A.P$ *for* $\forall(A, [x : A]P)$ *and* $App(F, a)$ *for* $App(A, P, F, a)$.

The usual logical operators can also be defined as follows (P_1 and P_2 are propositions, A is a type and P_3 is of kind $(A)Prop$):

$$\begin{aligned}
P_1 \supset P_2 &=_{df} \forall x : Prf(P_1).P_2 \\
true &=_{df} \forall P : Prop.P \supset P \\
false &=_{df} \forall P : Prop.P \\
P_1 \& P_2 &=_{df} \forall P : Prop.(P_1 \supset P_2 \supset P) \supset P \\
P_1 \vee P_2 &=_{df} \forall P : Prop.(P_1 \supset P) \supset (P_2 \supset P) \supset P \\
\neg P_1 &=_{df} P_1 \supset false \\
\exists x : A.P_3(x) &=_{df} \forall P : Prop.(\forall x : A.(P_3(x) \supset P)) \supset P
\end{aligned}$$

Propositional equality

Now, we introduce a new equality relation Eq of kind $(A : Type)(x : A)(y : A)Prop$ by declaring the following new constants in SOL.

$$\begin{aligned}
Eq &: (A : Type)(x : A)(y : A)Prop \\
eq &: (A : Type)(x : A)Prf(Eq(A, x, x)) \\
\mathcal{E}_{Eq} &: (A : Type)(x : A)(y : A)(P : (A)Prop) \\
&\quad (Prf(Eq(A, x, y)))(Prf(P(x)))Prf(P(y))
\end{aligned}$$

Remark 2.2.2 *We have the following remarks:*

- *There are two ways to introduce the equality* Eq. *One way is to declare new constants in SOL as above. The other is to use the definable Leibniz equality. Detailed discussion on different choices of introducing the equality* Eq *and the elimination operator* $\mathcal{E}_{Eq}$ *can be found in* [Luo94].
- *In Martin-Löf's type theory, the equality* $I(A, a, b)$ *(where* A *is a type and,* a *and* b *are objects of* A*) is introduced as a type rather than a proposition. Any two objects* p *and* q *of the type* $I(A, a, b)$

*are equal (*i.e. $p = q : I(A,a,b)$*); this is called proof irrelevancy. The introduction and elimination rules can be found in* [ML84].

Definition 2.2.3

- *We say that a proposition P is **provable** in a context Γ if $\Gamma \vdash p : Prf(P)$ for some p, and such a p is a **proof** of P.*
- *We say that the objects a and b of type A are **propositionally equal** if the proposition $Eq(A,a,b)$ is provable in the empty context in the intensional type theory UTT.*

2.3. Inductive data types

In this section, we shall introduce inductive data types based on the notion of inductive schemata, which is very similar to [Luo94], with the difference that we give recursive definitions of elimination rules and computation rules in this thesis. Inductive data types have been studied by, for example, Gentzen [Gen35] and Prawitz [Pra73, Pra74], for traditional logical systems, and by Martin-Löf [ML84], Backhouse [Bac88], Dybjer [Dyb91], and Coquand and Mohring [CPM90] for type theories. To understand an inductive type, we must understand both its introduction rules and its elimination/computation rules.

Definition 2.3.1 ***(Inductive schemata)*** *Let Γ be a valid context, $S_1, ..., S_k$ ($k \in \omega$) be kinds in Γ,* i.e. *judgement $\Gamma \vdash S_i$ Kind is derivable ($i = 1, ..., k$), X be a placeholder of kind $(s_1 : S_1)...(s_k : S_k)Type$ such that $X \notin FV(\Gamma)$.*

- ***A strictly positive operator** in Γ with respect to X is of one of the following forms:*
  1. *$\Phi \equiv X(s_1, ..., s_k)$, where $\Gamma \vdash s_i : S_i$ ($i = 1, ..., k$), or*
  2. *$\Phi \equiv (x : K)\Phi_0$, where K is a small Γ-kind, and Φ_0 is a strictly positive operator in $\Gamma, x : K$ with respect to X.*
- ***An inductive schema** Θ with respect to X is of one of the following forms:*
  1. *$\Theta \equiv X(s_1, ..., s_k)$, where $\Gamma \vdash s_i : S_i$ ($i = 1, ..., k$),or*
  2. *$\Theta \equiv (x : K)\Theta_0$, where K is a small Γ-kind, and Θ_0 is an inductive schema in $\Gamma, x : K$ with respect to X, or*

3. $\Theta \equiv (x : \Phi)\Theta_0$, *where* Φ *is a strictly positive operator in* Γ *with respect to* X, Θ_0 *is an inductive schema in* Γ *with respect to* X , *and* $x \notin FV(\Theta_0)$.

□

A strictly positive operator Φ with respect to X is of the form

$$(x_1 : K_1)...(x_t : K_t)X(s_1, ..., s_k)$$

where every K_j is a small kind. An inductive schema Θ with respect to X is of the form $(x_1 : M_1)...(x_t : M_t)X(s_1, ..., s_k)$, where every M_j is either a small kind or a strictly positive operator. When we introduce inductive data types into type theory, the smallness condition of kinds occurring in inductive schemata is important. For example, neither $(Type)X$ nor $((A)Type)X$ is an inductive schema because $Type$ is not a small kind. Otherwise such schema may lead to logical paradoxes.

Notation 2.3.2 *We often write* $\overline{s}$ *for* $s_1, ..., s_k$, $\overline{A}$ *for* $A_1, ..., A_n$, $\Phi[A]$ *for* $[A/X]\Phi$ *and* $\Theta[A]$ *for* $[A/X]\Theta$.

Definition 2.3.3 *Let* Θ *be an inductive schema. Then, for*

$$\begin{aligned} A &: (s_1 : S_1)...(s_k : S_k)Type \\ C &: (s_1 : S_1)...(s_k : S_k)(x : A(\overline{s}))Type \\ f &: (s_1 : S_1)...(s_k : S_k)(x : A(\overline{s}))C(\overline{s}, x) \\ z &: \Theta[A] \\ y &: \Phi[A] \end{aligned}$$

define kind $\Phi^*[C, y]$ *recursively as follows:*

$$\begin{aligned} (X(\overline{s}))^*[C, y] &= C(\overline{s}, y) \\ ((x : K)\Phi_0)^*[C, y] &= (x : K)\Phi_0^*[C, y(x)] \end{aligned}$$

define $\Phi^\natural[f, y]$ *of kind* $\Phi^*[C, y]$ *as follows:*

$$\begin{aligned} (X(\overline{s}))^\natural[f, y] &= f(\overline{s}, y) \\ ((x : K)\Phi_0)^\natural[f, y] &= [x : K]\Phi_0^\natural[f, y(x)] \end{aligned}$$

and define kind $\Theta^\circ[A, C, z]$ *recursively as follows:*

$$
\begin{aligned}
(X(\overline{s}))^\circ[A, C, z] &= C(\overline{s}, z) \\
((x : K)\Theta_0)^\circ[A, C, z] &= (x : K)\Theta_0^\circ[A, C, z(x)] \\
((x : \Phi)\Theta_0)^\circ[A, C, z] &= (x : \Phi[A])(x' : \Phi^*[C, x])\Theta_0^\circ[A, C, z(x)]
\end{aligned}
$$

With the above notations, we can now introduce the inductive data types.

Consider (parameterised) inductive data types generated by the following form:

$$\mathcal{T} =_{df} [A_1 : T_1]...[A_n : T_n]\mathcal{M}[\overline{\Theta}]$$

where every T_i $(i = 1, ..., n)$ is a kind in $\Gamma, A_1 : T_1, ..., A_{i-1} : T_{i-1}$, $\overline{\Theta} \equiv< \Theta_1, ..., \Theta_m >$ $(m \in \omega)$ is a finite sequence of inductive schemata in $\Gamma, A_1 : T_1, ..., A_n : T_n$.

Note 2.3.4 *None of the parameters occur free in* Γ *(*i.e. $\overline{A} \notin FV(\Gamma)$*) and there might be no parameter (*i.e. $n = 0$*).*

Then we declare the following constant expressions:

$$
\begin{aligned}
\mathcal{T} &: (A_1 : T_1)...(A_n : T_n)(s_1 : S_1)...(s_k : S_k)Type \\
l_j &: (A_1 : T_1)...(A_n : T_n)\Theta_j[\mathcal{T}(\overline{A})] \quad (j = 1, ..., m) \\
\mathcal{E}_{\mathcal{T}} &: (A_1 : T_1)...(A_n : T_n) \\
&\quad (C : (s_1 : S_1)...(s_k : S_k)(\mathcal{T}(\overline{A}, \overline{s}))Type) \\
&\quad (f_1 : \Theta_1^\circ[\mathcal{T}(\overline{A}), C, l_1(\overline{A})])... \\
&\quad (f_m : \Theta_m^\circ[\mathcal{T}(\overline{A}), C, l_m(\overline{A})]) \\
&\quad (s_1 : S_1)...(s_k : S_k)(z : \mathcal{T}(\overline{A}, \overline{s}))C(\overline{s}, z)
\end{aligned}
$$

In order to assert computation rules, we introduce the following notational definitions.

Definition 2.3.5 *Assume that* Θ *be of the form*

$$(x_1 : M_1)...(x_t : M_t)X(\overline{s})$$

and $x_1, ..., x_t$ *are fresh variables. Then*

- $\Theta^v =< x_1, ..., x_t >$,
- $\Theta^{last} = \overline{s}$,
- $\Theta^\sharp$ *as sequences of arguments:*

1. *if* $\Theta \equiv X$ *then* $\Theta^\sharp =<>$
2. *if* $\Theta \equiv (x_r : K)\Theta_0$ *then* $\Theta^\sharp =< x_r, \Theta_0^\sharp >$ *(*$r = 1, ..., t$*)*
3. *if* $\Theta \equiv (x_r : \Phi)\Theta_0$ *then* $\Theta^\sharp =< x_r, \Phi^\natural[\mathcal{E}_T(\overline{A}, C, \overline{f}), x_r], \Theta_0^\sharp >$ *(*$r = 1, ..., t$*)*

Remark 2.3.6 Θ^v *could be recursively defined as follows:*

1. *if* $\Theta \equiv X$ *then* $\Theta^v =<>$
2. *if* $\Theta \equiv (x_r : K)\Theta_0$ *then* $\Theta^v =< x_r, \Theta_0^v >$ *(*$r = 1, ..., t$*)*
3. *if* $\Theta \equiv (x_r : \Phi)\Theta_0$ *then* $\Theta^v =< x_r, \Theta_0^v >$ *(*$r = 1, ..., t$*)*

Finally, with above notational definition, we assert the following computation rules ($j = 1, ..., m$):

$$\mathcal{E}_T(\overline{A}, C, \overline{f}, \Theta_j^{last}, l_j(\overline{A}, \Theta_j^v)) = f_j(\Theta_j^\sharp) : C(\Theta_j^{last}, l_j(\overline{A}, \Theta_j^v))$$

Example 2.3.7 *We give five examples of inductive data types which will be used later.*

1. *The type of natural numbers:* $N =_{df} \mathcal{M}[X, (X)X]$
   There is no parameter for the type N *and the placeholder* X *is of kind* $Type$*. The declaration of constants and computation rule has already been given in Example 2.1.3. The functions for predecessor, addition, subtraction and multiplication can be defined as:*

$$\begin{aligned} pred &=_{df} \mathcal{E}_N([n : N]N, O, [x : N][y : N]x) \\ plus &=_{df} [m : N]\mathcal{E}_N([n : N]N, m, [x : N][y : N]S(y)) \\ minus &=_{df} [m : N]\mathcal{E}_N([n : N]N, m, [x : N][y : N]pred(y)) \\ times &=_{df} [m : N]\mathcal{E}_N([n : N]N, O, [x : N][y : N](m + y)) \end{aligned}$$

2. *Lists:* $List =_{df} [A : Type]\mathcal{M}[X, (A)(X)X]$
   There is one parameter $A : Type$ *and the placeholder* X *is of kind* $Type$*.*

Declare the following constants:

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

and assert the following computation rules:

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

The function map_{List} is defined as

$$\begin{array}{rcl} map_{List} & =_{df} & [A : Type][B : Type][c : (A)B] \\ & & \mathcal{E}_{List}(A, [l : List(A)]List(B), nil(B), \\ & & [a : A][l : List(A)][l' : List(B)]cons(B, c(a), l')) \end{array}$$

such that

$$\begin{array}{rcl} map_{List}(A, B, c, nil(A)) & = & nil(B) \\ map_{List}(A, B, c, cons(A, a, l)) & = & cons(B, c(a), d_{List}(l)) \end{array}$$

3. *Function types:* $(\to) =_{df} [A : Type][B : Type]\mathcal{M}[((A)B)X]$
   There are two parameters $A : Type$ and $B : Type$, and the placeholder X is of kind $Type$.
   Declare the following constants:

$$\begin{array}{rcl} (\to) & : & (A : Type)(B : Type)Type \\ lam & : & (A : Type)(B : Type)((A)B)(A \to B) \\ \mathcal{E}_{(\to)} & : & (A : Type)(B : Type)(C : (A \to B)Type) \\ & & ((g : (A)B)C(lam(A, B, g)))(z : A \to B)C(z) \end{array}$$

and assert the following computation rule:

$$\mathcal{E}_{(\to)}(A, B, C, f, lam(A, B, g)) = f(g) : C(lam(A, B, g))$$

4. *Binary trees:* $BTree =_{df} [A : Type]\mathcal{M}[X, (A)(X)(X)X]$
   There is one parameter $A : Type$ *and the placeholder* X *is of kind* $Type$.
   Declare the following constants:

$$\begin{aligned} BTree &: (A : Type)Type \\ empty &: (A : Type)BTree(A) \\ mk &: (A : Type)(a : A) \\ & \quad (t_1 : BTree(A))(t_2 : BTree(A))BTree(A) \\ \mathcal{E}_{BTree} &: (A : Type)(C : (BTree(A))Type)(C(empty(A))) \\ & \quad ((a : A)(t_1 : BTree(A))(C(t_1)) \\ & \quad (t_2 : BTree(A))(C(t_2))C(mk(A, a, t_1, t_2))) \\ & \quad (t : BTree(A))C(t) \end{aligned}$$

   and assert the following computation rules:

$$\begin{aligned} \mathcal{E}_{BTree}(A, C, c, f, empty(A)) &= c : C(empty(A)) \\ \mathcal{E}_{BTree}(A, C, c, f, mk(A, a, t_1, t_2)) &= f(a, t_1, \mathcal{E}_{BTree}(A, C, c, f, t_1), \\ & \quad t_2, \mathcal{E}_{BTree}(A, C, c, f, t_2)) \\ & \quad : C(mk(A, a, t_1, t_2)) \end{aligned}$$

5. *Vectors:* $Vec =_{df} [A : Type]\mathcal{M}[X, (n : N)(A)(X(n))X(S(n))]$
   There is one parameter $A : Type$ *and the placeholder* X *is of kind* $(N)Type$.
   Declare the following constants:

$$\begin{aligned} Vec &: (A : Type)(n : N)Type \\ vnil &: (A : Type)Vec(A, O) \\ vcons &: (A : Type)(n : N)(a : A)(l : Vec(A, n))Vec(A, S(n)) \\ \mathcal{E}_{Vec} &: (A : Type)(C : (n : N)(Vec(A, n))Type) \\ & \quad (C(O, vnil(A))) \\ & \quad ((n : N)(a : A)(l : Vec(A, n)) \\ & \quad (C(n, l))C(S(n), vcons(A, n, a, l))) \\ & \quad (n : N)(l : Vec(A, n))C(n, l) \end{aligned}$$

and assert the following computation rules:

$$\begin{aligned} \mathcal{E}_{Vec}(A, C, c, f, O, vnil(A)) &= c : C(O, vnil(A)) \\ \mathcal{E}_{Vec}(A, C, c, f, S(n), vcons(A, n, a, l)) &= f(n, a, l, \mathcal{E}_{Vec}(A, C, c, f, n, l)) \\ &\quad : C(S(n), vcons(A, n, a, l)) \end{aligned}$$

Remark 2.3.8 *Traditionally, the declaration of the elimination operator for binary trees is the following:*

$$\begin{aligned} \mathcal{E}_{BTree} \; : \; & (A : Type)(C : (BTree(A))Type)(C(empty(A))) \\ & ((a : A)(t_1 : BTree(A))(t_2 : BTree(A)) \\ & (C(t_1))(C(t_2))C(mk(A, a, t_1, t_2))) \\ & (t : BTree(A))C(t) \end{aligned}$$

During the time of my study of coercive subtyping rules for inductive data types, I discovered that the elimination operators and computation rules for inductive data types can be declared in a different way. The meaning of these new declarations is the same as before but the order of the arguements is different. The new order is generated by recursive functions over inductive schemata and it makes the implementation of inductive data types easier, especially, if one uses functional programming languages such as Haskell and ML. With these new declarations, it is also easier to give a general definition of coercions for the subtyping rules of parameterised inductive data types.

2.4. ST-form: a subset of inductive data types

In this section, we consider a subset of inductive data types that have only one constructor. We shall define some important function operators which will be used in later chapters.

Consider (parameterised) inductive data types generated by the following form (under a valid context Γ)[1] :

$$(ST - form) \qquad \mathcal{T} =_{df} [A_1 : T_1]...[A_n : T_n]\mathcal{M}[\Theta]$$

where Θ is an inductive schema in $\Gamma, A_1 : T_1, ..., A_n : T_n$ and has the form $(x_1 : K_1)...(x_t : K_t)X$, every K_j $(j = 1, ..., t)$ is a small kind, $X \notin K_j$, and T_i $(i = 1, ..., n)$ is a kind.

[1]ST stands for Strong Transitivity and is in contrast to WT that stands for Weak Transitivity.

Note *Θs that have the form $(x_1 : K_1)...(x_t : K_t)X$ is just a subset of inductive schemata in $\Gamma, A_1 : T_1, ..., A_n : T_n$ with respect to the placeholder X of Type, and type $\mathcal{T}(\overline{A})$ generated by ST-form has only one constructor. However, W-type*[2] *is not included in ST-form although it has only one constructor.*

One can also recursively define this subset of inductive schemata, which is called *ST-schema* later. An ST-schema Θ in Γ with respect to a placeholder X of *Type* is of one of the following forms:

1. $\Theta \equiv (x : K)X$, where K is a small kind in Γ, or
2. $\Theta \equiv (x : K)\Theta_0$, where K is a small kind in Γ and, Θ_0 is a ST-schema in $\Gamma, x : K$.

Then, we declare the following constant expressions:

$$\begin{aligned} \mathcal{T} &: (A_1 : T_1)...(A_n : T_n)Type \\ l &: (A_1 : T_1)...(A_n : T_n)\Theta[\mathcal{T}(\overline{A})] \\ \mathcal{E}_{\mathcal{T}} &: (A_1 : T_1)...(A_n : T_n) \\ &\quad (C : (\mathcal{T}(\overline{A}))Type) \\ &\quad (f : \Theta^{\circ}[\mathcal{T}(\overline{A}), C, l(\overline{A})]) \\ &\quad (z : \mathcal{T}(\overline{A}))C(z) \end{aligned}$$

and assert the following computation rule:

$$\mathcal{E}_{\mathcal{T}}(\overline{A}, C, f, l(\overline{A}, \Theta^{v})) = f(\Theta^{\sharp}) : C(l(\overline{A}, \Theta^{v}))$$

where the definitions of Θ°, Θ^{v} and $\Theta^{\sharp}$ are the same as in Section 2.3.

In order to define the function operators, we first introduce some notational definitions.

Definition 2.4.1 *Assume that small kind K has the form $(x_1 : K_1)...(x_t : K_t)El(A)$ and $x_1, ..., x_t$ are fresh variables. Then*

- $K^r = A$,
- $K^v = < x_1, ..., x_t >$,
- *Let z be a fresh variable of any kind K'. Define $K^p[z]$ of kind $(x_1 : K_1)...(x_t : K_t)K'$ as follows.*

$$K^p[z] = [x_1 : K_1]...[x_t : K_t]z$$

[2] $W =_{df} [A : Type][B : (A)Type]\mathcal{M}[(x : A)((B(x))X)X]$

Note *For any* $f : K$, *we have that* $f(K^v) : El(K^r)$.

Definition 2.4.2 *Assume that* Θ *is a ST-schema of the form* $(x_1 : K_1)...(x_t : K_t)X$ *and* $x_1, ..., x_t$ *are fresh variables and, let* z *be an fresh variable of any kind* K'. *Then, define* $\Theta^p[z]$ *of kind* $(x_1 : K_1)...(x_t : K_t)K'$ *as follows.*

$$\Theta^p[z] = [x_1 : K_1]...[x_t : K_t]z$$

With the above notations, we can now define function operators over inductive data types generated by the ST-form:

$$\mathcal{T} =_{df} [A_1 : T_1]...[A_n : T_n]\mathcal{M}[\Theta]$$

where Θ is a ST-schema of form $(x_1 : K_1)...(x_t : K_t)X$, and every K_j ($j = 1, ..., t$) is a small kind.

$$\begin{aligned} op_j \;\; =_{df} \;\; & [A_1 : T_1]...[A_n : T_n][z : \mathcal{T}(\overline{A})] \\ & [op_1(\overline{A}, z)/x_1, ..., op_{j-1}(\overline{A}, z)/x_{j-1}] \\ & K_j^p[\mathcal{E}_{\mathcal{T}}(\overline{A}, [G : \mathcal{T}(\overline{A})]K_j^r, \Theta^p[x_j(K_j^v)], z)] \end{aligned}$$

Now, we give some examples to demonstrate how to define function operators[3] over inductive data types with only one constructor.

Example 2.4.3 *The first example is the type of dependent function spaces; the second is the type of dependent pairs; the third is the type of non-dependent trio; the fourth example is the type of pairs in which the first component is functions and the second is objects of a type. These four types will be used in the later chapters.*

1. *The type of dependent function spaces:*

$$\Pi =_{df} [A : Type][B : (A)Type]\mathcal{M}[((x : A)B(x))X]$$

Declare the following constants:

$$\begin{aligned} \Pi \; &: \; (A : Type)(B : (A)Type)Type \\ \lambda \; &: \; (A : Type)(B : (A)Type)((x : A)B(x))\Pi(A, B) \\ \mathcal{E}_\Pi \; &: \; (A : Type)(B : (A)Type)(C : (\Pi(A, B)Type) \\ & \quad (f : (g : (x : A)B(x))C(\lambda(A, B, g))) \\ & \quad (z : \Pi(A, B))C(z) \end{aligned}$$

[3] One may regard these function operators as generalised projections.

and assert the following computation rule:

$$\mathcal{E}_\Pi(A, B, C, f, \lambda(A, B, g)) = f(g) : C(\lambda(A, B, g))$$

Then, the usual application operator can be defined as

$$\begin{aligned} app \ =_{df} \ & [A : Type][B : (A)Type] \\ & [F : \Pi(A, B)][a : A] \\ & \mathcal{E}_\Pi(A, B, [G : \Pi(A, B)]B(a), \\ & [g : (x : A)B(x)]g(a), F) \end{aligned}$$

which satisfies the equality (the β-rule for λ and app):

$$app(A, B, \lambda(A, B, g), a) = g(a) : B(a)$$

However, the η-rule does not hold:

$$\lambda(A, B, app(A, B, F)) \neq F$$

when $F : \Pi(A, B)$ is a variable.

2. *The type of dependent pairs:*

$$\Sigma =_{df} [A : Type][B : (A)Type]\mathcal{M}[(x : A)(B(x))X]$$

Declare the following constants:

$$\begin{aligned} \Sigma \ &: \ (A : Type)(B : (A)Type)Type \\ pair \ &: \ (A : Type)(B : (A)Type)(x : A)(B(x))\Sigma(A, B) \\ \mathcal{E}_\Sigma \ &: \ (A : Type)(B : (A)Type)(C : (\Sigma(A, B))Type) \\ & \quad (f : (x : A)(y : B(x))C(pair(A, B, x, y))) \\ & \quad (z : \Sigma(A, B))C(z) \end{aligned}$$

and assert the following computation rule:

$$\mathcal{E}_\Sigma(A, B, C, f, pair(A, B, x, y)) = f(x, y) : C(pair(A, B, x, y))$$

Then the projection operators can be defined as:

$$\begin{aligned}\pi_1 \;&=_{df}\; [A : Type][B : (A)Type][z : \Sigma(A,B)]\\ &\mathcal{E}_\Sigma(A,B,[z:\Sigma(A,B)]A,[x:A][y:B(x)]x,z)\\ \pi_2 \;&=_{df}\; [A : Type][B : (A)Type][z : \Sigma(A,B)]\\ &\mathcal{E}_\Sigma(A,B,[z:\Sigma(A,B)]B(\pi_1(A,B,z)),\\ &[x:A][y:B(x)]y,z)\end{aligned}$$

which satisfy the equalities:

$$\begin{aligned}\pi_1(A,B,pair(A,B,x.y)) &= x : A\\ \pi_2(A,B,pair(A,B,x,y)) &= y : B(x)\end{aligned}$$

3. *Non-dependent Trio:*

$$Trio =_{df} [A : Type][B : Type][C : Type]\mathcal{M}[(A)(B)(C)X]$$

Declare the following constants:

$$\begin{aligned}Trio \;&:\; (A:Type)(B:Type)(C:Type)Type\\ trio \;&:\; (A:Type)(B:Type)(C:Type)\\ &\quad (A)(B)(C)Trio(A,B,C)\\ \mathcal{E}_{Trio} \;&:\; (A:Type)(B:Type)(C:Type)\\ &\quad (D:(Trio(A,B,C))Type)\\ &\quad (f:(a:A)(b:B)(c:C)D(trio(A,B,C,a,b,c)))\\ &\quad (z:Trio(A,B,C))D(z)\end{aligned}$$

and assert the following computation rule:

$$\begin{aligned}&\mathcal{E}_{Trio}(A,B,C,D,f,trio(A,B,C,a,b,c))\\ &= f(a,b,c) : D(trio(A,B,C,a,b,c))\end{aligned}$$

Then the projection operators can be defined as

$$\begin{aligned}\pi_{Trio1} \;&=_{df}\; [A:Type][B:Type][C:Type]\\ &[z:Trio(A,B,C)]\\ &\mathcal{E}_{Trio}(A,B,C,[G:Trio(A,B,C)]A,\\ &[a:A][b:B][c:C]a,z)\end{aligned}$$

$$\begin{aligned}\pi_{Trio2} &=_{df} [A : Type][B : Type][C : Type]\\ &[z : Trio(A, B, C)]\\ &\mathcal{E}_{Trio}(A, B, C, [G : Trio(A, B, C)]B,\\ &[a : A][b : B][c : C]b, z)\end{aligned}$$

$$\begin{aligned}\pi_{Trio3} &=_{df} [A : Type][B : Type][C : Type]\\ &[z : Trio(A, B, C)]\\ &\mathcal{E}_{Trio}(A, B, C, [G : Trio(A, B, C)]C,\\ &[a : A][b : B][c : C]c, z)\end{aligned}$$

which satisfy the following equations:

$$\begin{aligned}\pi_{Trio1}(A, B, C, trio(A, B, C, a, b, c)) &= a : A\\ \pi_{Trio2}(A, B, C, trio(A, B, C, a, b, c)) &= b : B\\ \pi_{Trio3}(A, B, C, trio(A, B, C, a, b, c)) &= c : C\end{aligned}$$

4. $SPL =_{df} [A : Type][B : Type][C : Type]\mathcal{M}[((A)B)(C)X]$
Declare the following constants:

$$\begin{aligned}SPL &: (A : Type)(B : Type)(C : Type)Type\\ spl &: (A : Type)(B : Type)(C : Type)\\ &\quad ((A)B)(C)SPL(A, B, C)\\ \mathcal{E}_{SPL} &: (A : Type)(B : Type)(C : Type)\\ &\quad (D : (SPL(A, B, C))Type)\\ &\quad (f : (g : (A)B)(c : C)D(spl(A, B, C, g, c)))\\ &\quad (z : SPL(A, B, C))D(z)\end{aligned}$$

and assert the following computation rule:

$$\begin{aligned}&\mathcal{E}_{SPL}(A, B, C, D, f, spl(A, B, C, g, c))\\ &\quad = f(g, c) : D(spl(A, B, C, g, c))\end{aligned}$$

Then the function operators can be defined as

$$\begin{aligned}\pi_{SPL1} =_{df} & [A : Type][B : Type][C : Type] \\ & [z : SPL(A, B, C)][a : A] \\ & \mathcal{E}_{SPL}(A, B, C, [G : SPL(A, B, C)]B, \\ & [g : (A)B][c : C]g(a), z)\end{aligned}$$

$$\begin{aligned}\pi_{SPL2} =_{df} & [A : Type][B : Type][C : Type] \\ & [z : SPL(A, B, C)] \\ & \mathcal{E}_{SPL}(A, B, C, [G : SPL(A, B, C)]C, \\ & [g : (A)B][c : C]c, z)\end{aligned}$$

which satisfy the equalities:

$$\begin{aligned}\pi_{SPL1}(A, B, C, spl(A, B, C, g, c)) &= g : (A)B \\ \pi_{SPL2}(A, B, C, spr(A, B, C, g, c)) &= c : C\end{aligned}$$

2.5. Related work and Extensional type theory

2.5.1. Related work on UTT

It has been proved that, in Goguen's thesis [Gog94], UTT has nice meta-theoretical properties such as Church-Rosser, Subject Reduction, Strong Normalisation and the property of context replacement by equal kinds. We only give the following three properties in detail because they will be used later.

The theorem of Church-Rosser: If the judgement $\Gamma \vdash k_1 = k_2 : K$ is derivable in UTT then there is a term k_3 such that both k_1 and k_2 can be reduced to it.

The theorem of strong normalisation: Every well-typed term in UTT is strongly normalisable. That is, every computation sequence starting from a well-typed term in UTT is finite.

The property of context replacement by equal kinds: For any derivable judgement $\Gamma \vdash J$ in UTT, if $\vdash \Gamma = \Gamma'$ then $\Gamma' \vdash J$ is also derivable in UTT.

Implemented in the Lego proof development system, UTT has been applied to verification of functional programs [BM92, Bur93], imperative programs [Sch97], and concurrent programs [YL97], specification

and data refinement [Luo93] and formalisation of mathematics [Pol94]. UTT has also been implemented in Plastic, a proof development system, which contains the implementation of Martin-Löf's logical framework, inductive types, universes, and coercive subtyping [CL01, CL99]. I also implemented the logical framework and UTT can be specified in it. I used mutually recursive types to represent the terms and kinds in the logical framework so that as many as possible ill-typed terms are not representable. Another major difference is that I use recursive definitions of elimination rules and computation rules to implement inductive data types (see Section 2.3 for more details).

UTT also includes the predicative universes $Type_i$ ($i \in \omega$), which are types whose objects are names of types. Universes in UTT are specified in the Tarski style, using the explicit lifting operators to represent cumulativity in universes. We omit the details here because universes are irrelevant in the sense that the results in the thesis fit well into a type theory with or without universes.

2.5.2. Extensional type theory

In the intensional type theory UTT, if we add the following rule, the type theory then becomes an extensional type theory.

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash q : Prf(Eq(A, a, b))}{\Gamma \vdash a = b : A}$$

where Eq is the propositional equality, defined in Section 2.2, and $=$ is the judgemental equality. Note that the above rule makes the resulting type theory undecidable and it loses the property of strong normalisation.

Remark 2.5.1 *One may change the last premise of the above rule to* $\Gamma \vdash q : I(A, a, b)$ *where* $I(A, a, b)$ *is a type as introduced in Martin-Löf's type theory* [ML84] *because the* informal semantics *of* $Eq(A, a, b)$ *and* $I(A, a, b)$ *are the same for extensional type theories.*

Definition 2.5.2 ***(Extensional equality)*** *We say that* k_1 *and* k_2 *of kind* K *(under context* Γ*) are extensionally equal if the judgement* $\Gamma \vdash k_1 = k_2 : K$ *is derivable in the extensional type theory.*

Consistency

The internal logic in any type theory must be consistent, namely there is at least one formula in the system which cannot be proved. The consistency of a type theory cannot be established in itself; if the type theory is inconsistent, it proves everything, even its own consistency. So, in order to avoid circularity, model theory attempts to give semantics to explain a type theory using the notions outside the theory itself. In the literature, there are many models for Martin-Löf's intuitionistic type theory. For example, such models can be found in [Bee85, Smi84, Set93, Set04]. The existence of these (non-trivial) models implies the consistency of the extensional type theory.

CHAPTER 3

Coercive Subtyping

In order to make large scale formal reasoning easier, we need subtyping technology for abbreviation, reuse and inheritance. In this chapter, we first give a brief introduction to coercive subtyping, and summarise some results related to coercive subtyping. Then, we lay down the necessary formal details, and explain the notion of coherence and its importance.

3.1. Basic idea

An inductive type in type theory can be understood as a set consisting of its canonical objects. If we say type A is a subtype of type B, we mean that every object of type A is (regarded as) an object of type B.

The traditional approaches based on direct overloading do not generalise to inductive types. A natural consideration might be to form a subtype A of type B by selecting some (canonical) objects from B, which are regarded as the (canonical) objects of A. However, in such a setting, type-checking is difficult (and in general undecidable). It is not clear how one may introduce suitable restrictions on subtype formation to ensure decidable type-checking. One suggestion that has been made in the literature is to specify a subtype by declaring its constructors to be a subset of the constructors of an existing supertype [Coq92], but this would exclude some interesting applications of subtyping such as inheritance between mathematical theories represented as Σ-types.

As studied in [Luo99], coercive subtyping represents an approach to subtyping and inheritance in type theory. The basic idea of coercive subtyping is that A is a subtype of B if there is a (unique) coercion c from A to B, and therefore, any object of type A may be regarded as object of type B via c, where c is a functional operation from A to B in the type theory. In the theoretical framework of coercive subtyping, the role of c is represented by the coercive definition rule which says

that, if f is a functional operation with domain K, k_0 is an object of K_0, and c is a coercion from K_0 to K, then $f(k_0)$ is (well-typed and) definitionally equal to $f(c(k_0))$. The following rule is the basic coercive definition rule which shows the idea.

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

The above simple idea, when formulated in a typed logical framework [Luo94], becomes very powerful. Z. Luo has developed the framework that covers subtyping relations represented by the following kinds of coercions:

- *Simple coercions*: representing subtyping between two types. For example, coercions between basic inductive types: *Even* is a subtype of *Nat*.
- *Parameterised coercions:* representing (point-wise) subtyping (or subfamily relation) between two families of types indexed by objects of the same type. A coercion can be parameterised over free variables occurring in it and (possibly) its domain or range types. As a special of case, for example, each vector type $Vec(A, n)$ can be taken as a subtype of that of lists $List(A)$, parameterised by the index n, where the coercion would map the vector $< a_1, ..., a_n >$ to the list $[a_1, ..., a_n]$.
- *Coercions between parameterised inductive types:* we have general schematic rules that represent natural propagation of the basic coercions to other structured (or parameterised) inductive types. For example, $\Sigma(A, B)$ is a subtype of $\Sigma(A', B')$ if A is a subtype of A' and B is a subfamily of B'.

Coercive subtyping has applications in many areas such as large proof development, inductive reasoning, representing implicit syntax (*e.g.* overloading), etc.

3.2. A formal presentation

In this section, we give a formal presentation of the framework of coercive subtyping which is also the basis of our development latter.

A system with coercive subtyping, $T[\mathcal{R}]$, is an extension of any type theory T specified in LF, with two new judgement forms:

- $\Gamma \vdash A <_c B : Type$ asserts that type A is a subtyping of type B with c.
- $\Gamma \vdash K <_c K'$ asserts that kind K is subkind of kind K' with c.

The coercive subtyping system can be presented in two stages: first we consider the system $T[\mathcal{R}]_0$ with subtyping judgements of the form $\Gamma \vdash A <_c B : Type$, then the system $T[\mathcal{R}]$ with subkinding judgements of the form $\Gamma \vdash K <_c K'$.

Remark 3.2.1 *A type theory specified in LF, for example, Martin-Löf's intensional type theory or Luo's UTT, has nice meta-theoretical properties such as Church-Rosser, Subject Reduction and Strong Normalisation.*

3.2.1. The system $T[\mathcal{R}]_0$

$T[\mathcal{R}]_0$ is an extension of type theory T with the subtyping judgement form $\Gamma \vdash A <_c B : Type$, by adding the following rules:

- A set $\mathcal{R}$ of subtyping rules whose conclusions are subtyping judgements of the form $\Gamma \vdash A <_c B : Type$.
- The following congruence rule for subtyping judgements

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

In the presentation of coercive subtyping in [Luo99], $T[\mathcal{R}]_0$ also has the following substitution and transitivity rules:

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

$$(Trans)\quad \frac{\Gamma \vdash A <_c B : Type \quad \Gamma \vdash B <_{c'} C : Type}{\Gamma \vdash A <_{c' \circ c} C : Type}$$

Since we will prove that the substitution and transitivity rules are admissible, we do not include them as basic rules.

Remark 3.2.2 *We have the following remarks:*

- $T[\mathcal{R}]_0$ is obviously a conservative extension of the original type theory T, since the subtyping judgements do not contribute to any derivation of a judgement of any other form.
- The set of subtyping rules is supposed to be *coherent*; we shall give a definition and discussions of coherence in the next subsection.

- The substitution rule ($Subst$) and transitivity rule ($Trans$) cannot be directly implemented. For this reason, among others, proving the admissibility (or elimination) of such rules is always an important task for any subtyping system.

3.2.2. Coherence of the subtyping rules

The most basic requirement for such subtyping rules is that of coherence, given in the following definition, which essentially says that coercions between any two types must be unique.

Notation 3.2.3 *We often use the notation $\Gamma \not\vdash J$ which means the judgement $\Gamma \vdash J$ is not derivable in the current system.*

Definition 3.2.4 (Coherence condition of $T[\mathcal{R}]_0$) *We say that the subtyping rules are coherent if $T[\mathcal{R}]_0$ has the following coherence properties:*

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

Remark 3.2.5 *This notion of coherence is slightly different from the one given in* [Luo99], *since there the rules ($Subst$)($Trans$) are included in $T[\mathcal{R}]_0$. However, we will prove that these two rules are admissible in $T[\mathcal{R}]_0$. In general, when parameterised coercions and substitutions are present, coherence is undecidable. This is one of the reasons one needs to consider proofs of coherence in general.*

3.2.3. The system of $T[\mathcal{R}]$

Let $\mathcal{R}$ be a set of coherent subtyping rules. The system $T[\mathcal{R}]$, an extension of type theory T with coercive subtyping with respect to $\mathcal{R}$, is obtained from $T[\mathcal{R}]_0$ by adding the inference rules in Figure 3.2.1 and in Figure 3.2.2.

Remark 3.2.6 *The inference rules in Figure 3.2.1 and in Figure 3.2.2 are deliberately separated. In the system we are presenting at the moment, all the rules are included. In the system with weak transitivity,*

Basic subkinding rule:

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

Subkinding rule for dependent kinds:

$$\frac{\Gamma, x : K_1 \vdash K_2 <_c K_2'}{\Gamma \vdash (x : K_1)K_2 <_{[f:(x:K_1)K_2][x:K_1]c(f(x))} (x : K_1)K_2'}$$

Congruence rule for subkinding:

$$\frac{\begin{array}{c}\Gamma \vdash K_1 <_c K_2 \\ \Gamma \vdash K_1 = K_1' \quad \Gamma \vdash K_2 = K_2' \quad \Gamma \vdash c = c' : (K_1)K_2\end{array}}{\Gamma \vdash K_1' <_{c'} K_2'}$$

Substitution rule for subkinding:

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

Coercive application rules:

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

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

Coercive definition rule:

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

FIGURE 3.2.1. Inference rules in $T[\mathcal{R}]$

which we will present later, only the rules in Figure 3.2.1 will be included. (see Section 5.8 for more details).

The coherence of the subtyping rules is a necessary condition to preserve that the coercive subtyping system $T[\mathcal{R}]$ is a conservative extension of the original type theory T. In fact, as pointed out by Sergei Soloviev, we show that, by the coercive definition rule (CD) and $\beta\eta$-equality rules, if $\Gamma \vdash K <_c K'$ and $\Gamma \vdash K <_{c'} K'$,

Subkinding rule for dependent kinds:

$$\frac{\Gamma \vdash K_1' <_c K_1 \quad \Gamma, x : K_1 \vdash K_2\, kind \quad \Gamma, x' : K_1' \vdash [c(x')/x]K_2 = K_2'}{\Gamma \vdash (x : K_1)K_2 <_{[f:(x:K_1)K_2][x':K_1']f(c(x'))} (x : K_1')K_2'}$$

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

Transitivity rule for subkinding:

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

FIGURE 3.2.2. Inference rules in $T[\mathcal{R}]$

then $\Gamma \vdash c = c' : (K)K'$. The proof is the following:

$$\begin{aligned} c &=_{\beta\eta} [x : K]([y : K']y)(c(x)) \\ &=_{CD} [x : K]([y : K']y)(x) \\ &=_{CD} [x : K]([y : K']y)(c'(x)) \\ &=_{\beta\eta} c' \end{aligned}$$

3.3. The problems

As we mentioned above, a vital requirement for coercive subtyping system is that of coherence of the subtyping rules – computational uniqueness of coercions between any two types. When we implement coercive subtyping, a problem is how to decide whether subtyping rules are coherent. Unless coercions can be represented as a finite graph, this problem is in general undecidable with possibly infinitely many coercions (*e.g.* introduced by parameterised coercions). So, how to prove coherence of the subtyping rules which can probably generate infinite many coercions needs to be studied; this is one of the contributions of this thesis.

Another problem related to implementation of coercive subtyping is that substitution rules and transitivity rules cannot be directly implemented. For this reason, among others, proving the admissibility (or elimination) of such rules is always an important task for any subtyping system. Some results on transitivity elimination for subkinding

have been presented in [JLS98, SL02]. However, how to prove the admissibility of transitivity and substitution at type level (*Trans*, *Subst*) has not been studied; this is another subject of this thesis.

It is worth mentioning now that, for certain subtyping rules (*e.g.* rules in Figure 3.5.1 and 3.5.2), the transitivity rule (*Trans*) is admissible. However, for many very natural subtyping rules (*e.g.* the subtyping rule for lists), the transitivity rule (*Trans*) cannot be admissible. This problem inspires us to introduce in Chapter 5 a new notion called *'Weak Transitivity'*, and to prove that weak transitivity is admissible. The essence is that we are more concerned about the existence of coercions between two types, and this new notion has a wider application.

3.4. Well-defined coercions

In this section, we shall give a definition of well-defined coercions.

After new subtyping rules are added into $\mathcal{R}$, we need to prove that the system $T[\mathcal{R}]_0$ is still coherent and that the transitivity rule and substitution rule are admissible. A general strategy we adopt is to consider such proofs in a stepwise way. That is, we first suppose that some existing coercions (possibly generated by some existing rules) are coherent and have good admissibility properties; then prove that all the good properties are kept after new subtyping rules are added. This leads us to define the following concept of *well-defined coercions.*

Definition 3.4.1 (Well-defined coercions) *If $\mathcal{C}$ is a set of subtyping judgements of the form $\Gamma \vdash M <_d M' : Type$ which satisfies the following conditions, we say that $\mathcal{C}$ is a* well-defined set of judgements for coercions, *briefly called* Well-Defined Coercions (WDC*).*

1. *(Coherence)*
   (a) $\Gamma \vdash A <_c B : Type \in \mathcal{C}$ *implies* $\Gamma \vdash A : Type$, $\Gamma \vdash B : Type$ *and* $\Gamma \vdash c : (A)B$.
   (b) $\Gamma \vdash A <_c A : Type \notin \mathcal{C}$ *for any* Γ, A, *and* c.
   (c) $\Gamma \vdash A <_{c_1} B : Type \in \mathcal{C}$ *and* $\Gamma \vdash A <_{c_2} B : Type \in \mathcal{C}$ *imply* $\Gamma \vdash c_1 = c_2 : (A)B$.
2. *(Congruence)* $\Gamma \vdash A <_c B : Type \in \mathcal{C}$, $\Gamma \vdash A = A' : Type$, $\Gamma \vdash B = B' : Type$ *and* $\Gamma \vdash c = c' : (A)B$ *imply* $\Gamma \vdash A' <_{c'} B' \in \mathcal{C}$.

3. *(Transitivity)* $\Gamma \vdash A <_{c_1} B : Type \in \mathcal{C}$ *and* $\Gamma \vdash B <_{c_2} C : Type \in \mathcal{C}$ *imply* $\Gamma \vdash A <_{c_2 \circ c_1} C : Type \in \mathcal{C}$.
4. *(Substitution)* $\Gamma, x : K, \Gamma' \vdash A <_c B : Type \in \mathcal{C}$ *implies for any* k *such that* $\Gamma \vdash k : K$, $\Gamma, [k/x]\Gamma' \vdash [k/x]A <_{[k/x]c} [k/x]B : Type \in \mathcal{C}$.
5. *(Weakening)* $\Gamma \vdash A <_c B : Type \in \mathcal{C}$, $\Gamma \subseteq \Gamma'$ *and* Γ' *is valid imply* $\Gamma' \vdash A <_c B : Type \in \mathcal{C}$.

By the definition of WDC, we have the following properties.

Lemma 3.4.2 *Let* $\mathcal{C}$ *be a WDC.*

1. *If* $\Gamma \vdash A <_{c_1} B : Type \in \mathcal{C}$, $\Gamma \vdash B' <_{c_2} C : Type \in \mathcal{C}$ *and* $\Gamma \vdash B = B' : Type$ *then* $\Gamma \vdash A <_{c_2 \circ c_1} C : Type \in \mathcal{C}$.
2. *If* $\Gamma, x : K, \Gamma' \vdash A <_c B : Type \in \mathcal{C}$ *and* $\Gamma \vdash K = K'$ *then* $\Gamma, x : K', \Gamma' \vdash A <_c B : Type \in \mathcal{C}$.
3. *If* $\Gamma \vdash A <_c B : Type \in \mathcal{C}$ *and* $\vdash \Gamma = \Gamma'$ *then* $\Gamma' \vdash A <_c B : Type \in \mathcal{C}$.
4. *If* $\Gamma \vdash A <_c B : Type \in \mathcal{C}$, $\Gamma' \vdash A' <_{c'} B' : Type \in \mathcal{C}$, $\vdash \Gamma = \Gamma'$, $\Gamma \vdash A = A' : Type$ *and* $\Gamma \vdash B = B' : Type$ *then* $\Gamma \vdash c = c' : (A)B$.

We shall consider the system of coercive subtyping in which the set ($\mathcal{R}$) of the subtyping rules includes the following rule,

$$(WDCrule) \qquad \frac{\Gamma \vdash A <_c B : Type \quad \in \quad \mathcal{C}}{\Gamma \vdash A <_c B : Type}$$

where $\mathcal{C}$ is a WDC.

3.5. Subtyping rules

The set $\mathcal{R}$ of subtyping rules is open in the sense that we can always introduce new subtyping rules in $\mathcal{R}$, so long as the good properties such as coherence are kept. For example, at this moment, we introduce subtyping rules for Π-types and Σ-types into $\mathcal{R}$ (Figure 3.5.1 and Figure 3.5.2; details of Π-types and Σ-types are on Page 37 and Page 38). More subtyping rules will be introduced in later chapters.

Remark 3.5.1 *We have the following remarks:*

- *The basic understanding of the subtyping rules for* Π*-types is that* $\Pi(A, B)$ *is a subtype of* $\Pi(A', B')$ *if* A' *is a subtype of* A *and* B *is a sub-family of* B' *(we omit other cases such as:* $\Pi(A, B)$ *is a subtype of* $\Pi(A, B')$ *if* B *is a sub-family of* B'*).*

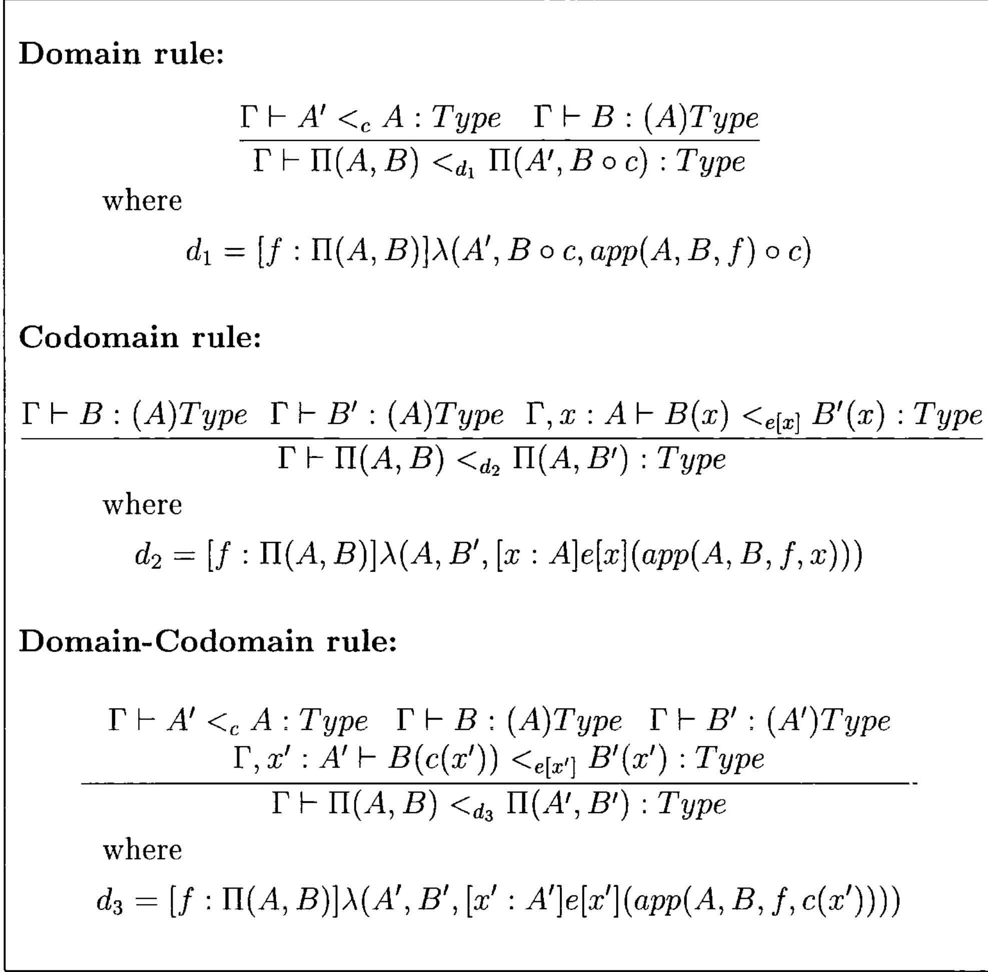


FIGURE 3.5.1. Subtyping rules for Π-types

- *We use the application operator app to define the coercions in Figure 3.5.1 and have the following equations:*

$$\begin{aligned} d_1(\lambda(A,B,g)) &= \lambda(A', B \circ c, g \circ c) \\ d_2(\lambda(A,B,g)) &= \lambda(A, B', [x : A]e[x](g(x))) \\ d_3(\lambda(A,B,g)) &= \lambda(A', B', [x' : A']e[x'](g(c(x')))) \end{aligned}$$

- *The basic understanding of the subtyping rules for Σ-types is that $\Sigma(A,B)$ is a subtype of $\Sigma(A',B')$ if A is a subtype of A' and B is a sub-family of B' (we omit other cases such as: $\Sigma(A,B)$ is a subtype of $\Sigma(A,B')$ if B is a sub-family of B').*

First Component rule:

$$\frac{\Gamma \vdash A <_c A' : Type \quad \Gamma \vdash B : (A')Type}{\Gamma \vdash \Sigma(A, B \circ c) <_{d_1} \Sigma(A', B) : Type}$$

where

$$d_1 = [z : \Sigma(A, B \circ c)]pair(A', B, c(\pi_1(A, B \circ c, z)), \pi_2(A, B \circ c, z))$$

Second Component rule:

$$\frac{\Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A)Type \quad \Gamma, x : A \vdash B(x) <_{e[x]} B'(x) : Type}{\Gamma \vdash \Sigma(A, B) <_{d_2} \Sigma(A, B') : Type}$$

where

$$d_2 = [z : \Sigma(A, B)]pair(A, B', \pi_1(A, B, z), e[\pi_1(A, B, z)](\pi_2(A, B, z)))$$

First-Second Component rule:

$$\frac{\Gamma \vdash A <_c A' : Type \quad \Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A')Type \quad \Gamma, x : A \vdash B(x) <_{e[x]} B'(c(x)) : Type}{\Gamma \vdash \Sigma(A, B) <_{d_3} \Sigma(A', B') : Type}$$

where

$$d_3 = [z : \Sigma(A, B)]pair(A', B', c(\pi_1(A, B, z)), e[\pi_1(A, B, z)](\pi_2(A, B, z)))$$

FIGURE 3.5.2. Subtyping rules for Σ-types

- *We use the projection operators π_1 and π_2 to define the coercions in Figure 3.5.2 and have the following equations:*

$$\begin{aligned} d_1(pair(A, B \circ c, x, y) &= pair(A', B, c(x), y) \\ d_2(pair(A, B, x, y) &= pair(A, B', x, e[x](y)) \\ d_3(pair(A, B, x, y) &= pair(A', B', c(x), e[x](y)) \end{aligned}$$

We now give two examples to show that the definitions of coercions in Figure 3.5.1 and in Figure 3.5.2 are suitable to the admissibility of the transitivity rule ($Trans$), but the inductively defined coercions are not.

Example 3.5.2 *Assume that* $\Gamma \vdash B : (A)Type$, $\Gamma \vdash B' : (A)Type$, $\Gamma \vdash B'' : (A)Type$ *and* $\Gamma, x : A \vdash B(x) <_{e_1[x]} B'(x)$, $\Gamma, x : A \vdash B'(x) <_{e_2[x]} B''(x)$ *and* $\Gamma, x : A \vdash B(x) <_{e_2[x] \circ e_1[x]} B''(x)$. *Then by the Codomain rule, we have*

$$\Gamma \vdash \Pi(A, B) <_{d_1} \Pi(A, B')$$

$$\Gamma \vdash \Pi(A, B') <_{d_2} \Pi(A, B'')$$

$$\Gamma \vdash \Pi(A, B) <_{d} \Pi(A, B'')$$

where

$$\begin{array}{rcl} d_1 & = & [f : \Pi(A,B)]\lambda(A, B', [x : A]e_1[y](app(A, B, f, x))) \\ d_2 & = & [g : \Pi(A,B')]\lambda(A, B'', [x : A]e_2[x](app(A, B', g, x))) \\ d & = & [f : \Pi(A,B)]\lambda(A, B'', [x : A]e_2[x](e_1[x](app(A, B, f, x)))) \end{array}$$

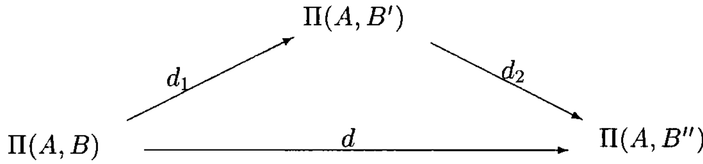


Figure for example 3.5.2

$$\begin{array}{rcl} d_2 \circ d_1 & =_{df} & [f : \Pi(A,B)]d_2(d_1(f)) \\ & = & [f : \Pi(A,B)] \\ & & \lambda(A, B'', [x : A]e_2[x](app(A, B', d_1(f), x))) \\ & = & [f : \Pi(A,B)]\lambda(A, B'', [x : A]e_2[x](app(A, B', \\ & & \lambda(A, B', [y : A]e_1[y](app(A, B, f, y))), x))) \\ & = & [f : \Pi(A,B)]\lambda(A, B'', \\ & & [x : A]e_2[x](([y : A]e_1[y](app(A, B, f, y)))(x))) \\ & = & [f : \Pi(A,B)] \\ & & \lambda(A, B'', [x : A]e_2[x](e_1[x](app(A, B, f, x)))) \\ & = & d \end{array}$$

However, if we choose the inductively defined coercions as the following:

$$\begin{aligned} d_1 &= \mathcal{E}_\Pi(A, B, [f : \Pi(A, B)]\Pi(A, B'), \\ &\quad [h : (x : A)B(x)]\lambda(A, B', [x : A]e_1[x](h(x)))) \\ d_2 &= \mathcal{E}_\Pi(A, B', [g : \Pi(A, B')]\Pi(A, B''), \\ &\quad [h : (x : A)B'(x)]\lambda(A, B'', [x : A]e_2[x](h(x)))) \\ d &= \mathcal{E}_\Pi(A, B, [f : \Pi(A, B)]\Pi(A, B''), \\ &\quad [h : (x : A)B(x)]\lambda(A, B'', [x : A]e_2[x](e_1[x](h(x))))) \end{aligned}$$

then d and $d_2 \circ d_1$ are not computationally equal in an intensional type theory. This causes the transitivity rule ($Trans$) not to be admissible, although d and $d_2 \circ d_1$ are extensionally equal. In fact, for any canonical object $\lambda(A, B, h)$ of type $\Pi(A, B)$, we have

$$\begin{aligned} d(\lambda(A, B, h)) &= \lambda(A, B'', [x : A]e_2[x](e_1[x](h(x)))) \\ &= d_2(d_1(\lambda(A, B, h))) \end{aligned}$$

Example 3.5.3 *Assume that* $\Gamma \vdash B : (A)Type$, $\Gamma \vdash B' : (A)Type$, $\Gamma \vdash B'' : (A)Type$ *and* $\Gamma, x : A \vdash B(x) <_{e_1[x]} B'(x)$, $\Gamma, x : A \vdash B'(x) <_{e_2[x]} B''(x)$ *and* $\Gamma, x : A \vdash B(x) <_{e_2[x] \circ e_1[x]} B''(x)$. *Then by the Second Component rule, we have*

$$\Gamma \vdash \Sigma(A, B) <_{d_1} \Sigma(A, B')$$

$$\Gamma \vdash \Sigma(A, B') <_{d_2} \Sigma(A, B'')$$

$$\Gamma \vdash \Sigma(A, B) <_{d} \Sigma(A, B'')$$

where

$$\begin{aligned} d_1 &= [z : \Sigma(A, B)]pair(A, B', \pi_1(A, B, z), \\ &\quad e_1[\pi_1(A, B, z)](\pi_2(A, B, z))) \\ d_2 &= [z' : \Sigma(A, B')]pair(A, B'', \pi_1(A, B', z'), \\ &\quad e_2[\pi_1(A, B', z')](\pi_2(A, B', z'))) \\ d &= [z : \Sigma(A, B)]pair(A, B'', \pi_1(A, B, z), \\ &\quad e_2[\pi_1(A, B, z)](e_1[\pi_1(A, B, z)](\pi_2(A, B, z)))) \end{aligned}$$

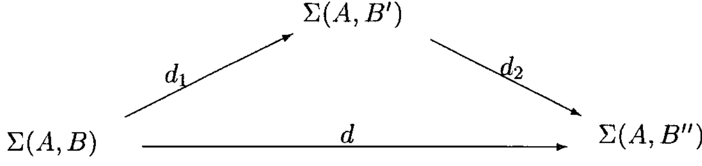


Figure for example 3.5.3

$$\begin{aligned} d_2 \circ d_1 &=_{df} [z : \Sigma(A,B)]d_2(d_1(z)) \\ &= [z : \Sigma(A,B)]pair(A, B'', \pi_1(A, B', d_1(z)), \\ &\quad e_2[\pi_1(A, B', d_1(z))](\pi_2(A, B', d_1(z)))) \\ &= [z : \Sigma(A,B)]pair(A, B'', \pi_1(A, B, z), \\ &\quad e_2[\pi_1(A, B, z)](e_1[\pi_1(A, B, z)](\pi_2(A, B, z)))) \\ &= d \end{aligned}$$

However, if we choose the inductively defined coercions as the following:

$$\begin{aligned} d_1 &= \mathcal{E}_\Sigma(A, B, [z : \Sigma(A,B)]\Sigma(A, B'), \\ &\quad [x : A][y : B(x)]pair(A, B', x, e_1[x](y))) \end{aligned}$$

$$\begin{aligned} d_2 &= \mathcal{E}_\Sigma(A, B', [z' : \Sigma(A,B')]\Sigma(A, B''), \\ &\quad [x : A][y : B'(x)]pair(A, B'', x, e_2[x](y))) \end{aligned}$$

$$\begin{aligned} d &= \mathcal{E}_\Sigma(A, B, [z : \Sigma(A,B)]\Sigma(A, B''), \\ &\quad [x : A][y : B(x)]pair(A, B'', x, e_2[x](e_1[x](y)))) \end{aligned}$$

then d and $d_2 \circ d_1$ are not computationally equal in an intensional type theory. This causes the transitivity rule $(Trans)$ not to be admissible, although d and $d_2 \circ d_1$ are extensionally equal. In fact, for any canonical object $pair(A, B, x, y)$ of type $\Sigma(A, B)$, we have

$$\begin{aligned} d(pair(A, B, x, y)) &= pair(A, B'', x, e_2[x](e_1[x](y))) \\ &= d_2(d_1(pair(A, B, x, y))) \end{aligned}$$

CHAPTER 4

Coherence and Transitivity

In this chapter, we shall use the subtyping rules for Π and Σ-types as examples to demonstrate how coherence can be proved. We shall also prove the admissibility of the substitution rule (*Subst*) and the transitivity rule (*Trans*). Let's make clear that the set $\mathcal{R}$ of subtyping rules now consists of the rule *WDCrule* where $\mathcal{C}$ in the rule is a set of well-defined coercions (WDC) and the subtyping rule for Π and Σ-types in Figure 3.5.1 and Figure 3.5.2 and, the system $T[\mathcal{R}]_0$ also includes the congruence rule (*Cong*). Furthermore, we assume that for any judgement $\Gamma \vdash A <_c B : Type \in \mathcal{C}$, neither A nor B is computationally equal to a Π-type or a Σ-type. We also assume that the original type theory T has good properties, in particular the Church-Rosser property and the property of context replacement by equal kinds.

We denote by $\mathcal{C}_{\mathcal{M}}$ the set of the derivable subtyping judgements of the form $\Gamma \vdash M <_d M' : Type$ in $T[\mathcal{R}]_0$; that is, $\Gamma \vdash M <_d M' : Type \in \mathcal{C}_{\mathcal{M}}$ if and only if $\Gamma \vdash M <_d M' : Type$ is derivable in $T[\mathcal{R}]_0$. In this chapter, we shall show that $\mathcal{C}_{\mathcal{M}}$ is a WDC.

It is obvious that $\mathcal{C}_{\mathcal{M}}$ is a superset of $\mathcal{C}$ (*i.e.* $\mathcal{C}_{\mathcal{M}} \supseteq \mathcal{C}$) because the rule *WDCrule* is included in the system $T[\mathcal{R}]_0$.

4.1. Coherence of $T[\mathcal{R}]_0$

We give a proof of coherence of the system $T[\mathcal{R}]_0$ in this section.

Notation 4.1.1 *Since we are not much concerned with the subkinding judgements and are mainly concerned with the subtyping judgements, we shall simply write $\Gamma \vdash A <_c B$ for $\Gamma \vdash A <_c B : Type$, where no confusion may occur. Sometimes, we shall also write $\Gamma \vdash k_1 = k_2$ for $\Gamma \vdash k_1 = k_2 : K$ when we have no concern for the kind K.*

Lemma 4.1.2 *If $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, then one of the following holds:*

- *$\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}$; or*

- *Both M_1 and M_2 are computationally equal to Π-types; or*
- *Both M_1 and M_2 are computationally equal to Σ-types.*

Proof. By induction on derivations.

If $\Gamma \vdash M_1 <_d M_2 \notin \mathcal{C}$, its derivation must end with a Π-subtyping rule, or a Σ-subtyping rule, or the congruence rule. If it is one of the Π or Σ-subtyping rules, then we know both M_1 and M_2 are Π-types or Σ-types. If the last rule is the congruence rule $(Cong)$,

$$\frac{\begin{array}{c}\Gamma \vdash M_1' <_{d'} M_2' \\ \Gamma \vdash M_1 = M_1' \quad \Gamma \vdash M_2 = M_2' \quad \Gamma \vdash d' = d : (M_1')M_2'\end{array}}{\Gamma \vdash M_1 <_d M_2}$$

then by the induction hypothesis, the lemma holds for $\Gamma \vdash M_1' <_{d'} M_2'$. If both M_1' and M_2' are computationally equal to Π-types or Σ-types, so are M_1 and M_2. If $\Gamma \vdash M_1' <_{d'} M_2' \in \mathcal{C}$, then $\Gamma \vdash M_1 <_d M_2 \in \mathcal{C}$ because $\mathcal{C}$ is a WDC, which is closed under congruence. □

Lemma 4.1.3 *We have the following lemmas.*

1. *If $\Gamma \vdash \Pi(A, B) <_d \Pi(A', B') : Type \in \mathcal{C}_\mathcal{M}$ then $\Gamma \vdash A = A' : Type$ or $\Gamma \vdash A' <_c A : Type \in \mathcal{C}_\mathcal{M}$ for some c.*
2. *If $\Gamma \vdash \Sigma(A, B) <_d \Sigma(A', B') : Type \in \mathcal{C}_\mathcal{M}$ then $\Gamma \vdash A = A' : Type$ or $\Gamma \vdash A <_c A' : Type \in \mathcal{C}_\mathcal{M}$ for some c.*
3. *If $\Gamma \vdash \Pi(A, B) <_d \Pi(A', B') : Type \in \mathcal{C}_\mathcal{M}$ and $\Gamma \vdash A = A' : Type$ then $\Gamma, x : A \vdash B(x) <_{e[x]} B'(x) : Type \in \mathcal{C}_\mathcal{M}$ for some e.*
4. *If $\Gamma \vdash \Sigma(A, B) <_d \Sigma(A', B') : Type \in \mathcal{C}_\mathcal{M}$ and $\Gamma \vdash A = A' : Type$ then $\Gamma, x : A \vdash B(x) <_{e[x]} B'(x) : Type \in \mathcal{C}_\mathcal{M}$ for some e.*
5. *If $\Gamma \vdash \Pi(A, B) <_d \Pi(A', B') : Type \in \mathcal{C}_\mathcal{M}$ and $\Gamma \vdash A' <_c A : Type \in \mathcal{C}_\mathcal{M}$ then $\Gamma, x : A' \vdash B(c(x)) = B'(x) : Type$ or $\Gamma, x : A' \vdash B(c(x)) <_{e[x]} B'(x) : Type \in \mathcal{C}_\mathcal{M}$ for some e.*
6. *If $\Gamma \vdash \Sigma(A, B) <_d \Sigma(A', B') : Type \in \mathcal{C}_\mathcal{M}$ and $\Gamma \vdash A <_c A' : Type \in \mathcal{C}_\mathcal{M}$ then $\Gamma, x : A \vdash B(x) = B'(c(x)) : Type$ or $\Gamma, x : A \vdash B(x) <_{e[x]} B'(c(x)) : Type \in \mathcal{C}_\mathcal{M}$ for some e.*

Proof. By induction on derivations.

We consider only the first statement here; the proofs of the others are similar. For the first, any derivation of the judgement $\Gamma \vdash \Pi(A, B) <_d \Pi(A', B')$ must contain a sub-derivation whose last rule is one of the subtyping rules for Π-types followed by a finite number of

applications of the congruence rule.

$$\frac{\vdots}{\Gamma \vdash \Pi(A_1, B_1) <_{d'} \Pi(A_2, B_2)}$$

$$\vdots$$

$$...(Congruence \quad rule)...$$

$$\frac{\vdots}{\Gamma \vdash \Pi(A, B) <_d \Pi(A', B')}$$

where $\Gamma \vdash \Pi(A_1, B_1) = \Pi(A, B)$, $\Gamma \vdash \Pi(A_2, B_2) = \Pi(A', B')$, and $\Gamma \vdash d' = d$ respectively. Hence, by the Church-Rosser theorem of the original type theory T and conservativity of $T[\mathcal{R}]_0$ over T, we have $\Gamma \vdash A_1 = A$, $\Gamma \vdash B_1 = B$, $\Gamma \vdash A_2 = A'$ and $\Gamma \vdash B_2 = B'$.

Since $\Gamma \vdash \Pi(A_1, B_1) <_{d'} \Pi(A_2, B_2)$ is derived by one of the three subtyping rules for Π-types, if it is the Codomain rule, we have $\Gamma \vdash A_1 = A_2$; if it is the Domain rule or the Domain-Codomain rule, we have $\Gamma \vdash A' <_c A$ for some c. So $\Gamma \vdash A = A'$ or $\Gamma \vdash A' <_c A$ for some c by the congruence rule. □

Lemma 4.1.4 *If* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$*, then* $\Gamma \nvdash M_1 = M_2 : Type$.

Proof. By induction on derivations and the definition of WDC, particularly, the coherence requirement 1(b) in the Definition 3.4.1. □

Theorem 4.1.5 ***(Coherence)*** *If* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, $\Gamma' \vdash M_1' <_{d'} M_2' : Type \in \mathcal{C}_{\mathcal{M}}$, $\vdash \Gamma = \Gamma'$, $\Gamma \vdash M_1 = M_1' : Type$, *and* $\Gamma \vdash M_2 = M_2' : Type$ *then* $\Gamma \vdash d = d' : (M_1)M_2$.

Proof. By induction on derivations.

By Lemma 4.1.2, we have to consider only the following three cases.

- $\Gamma \vdash M_1 <_d M_2 \in \mathcal{C}$. Then, none of M_1 and M_2 is computationally equal to a Π-type or Σ-type by the assumption; and nor is M_1' or M_2' because $\Gamma \vdash M_1 = M_1'$ and $\Gamma \vdash M_2 = M_2'$. So, by Lemma 4.1.2, we have $\Gamma \vdash M_1' <_{d'} M_2' \in \mathcal{C}$. Now, by Lemma 3.4.2(4), we have $\Gamma \vdash d = d' : (M_1)M_2$.
- Both M_1 and M_2 are computationally equal to Π-types. Then any derivation of $\Gamma \vdash M_1 <_d M_2$ contains a sub-derivation whose last

rule is one of the subtyping rules for Π-types followed by a finite number of applications of the congruence rule. We consider only the case where the Π-subtyping rule concerned is the third rule in Figure 3.5.1; *i.e*, the derivation is of the form

$$\dfrac{\dfrac{\Gamma \vdash A_2 <_c A_1 \qquad \Gamma, x : A_2 \vdash B_1(c(x)) <_{e[x]} B_2(x)}{\Gamma \vdash \Pi(A_1, B_1) <_{d_1} \Pi(A_2, B_2)} \quad ...(Congruence\ \ rule)...}{\Gamma \vdash M_1 <_d M_2}$$

where $\Gamma \vdash \Pi(A_1, B_1) = M_1$, $\Gamma \vdash \Pi(A_2, B_2) = M_2$, $\Gamma \vdash d_1 = d$ and

$$d_1 = [f : \Pi(A_1, B_1)]\lambda(A_2, B_2, [x : A_2]e[x](app(A_1, B_1, f, c(x))))$$

Now, it must be the case that any derivation of $\Gamma' \vdash M_1' <_{d'} M_2'$ must contain a sub-derivation whose last rule is also the same subtyping rule for Π-types as above, followed by a finite number of applications of the congruence rule; *i.e*, it must be of the form

$$\dfrac{\dfrac{\Gamma' \vdash A_2' <_{c'} A_1' \qquad \Gamma', x : A_2' \vdash B_1'(c'(x)) <_{e'[x]} B_2'(x)}{\Gamma' \vdash \Pi(A_1', B_1') <_{d_1'} \Pi(A_2', B_2')} \quad ...(Congruence\ \ rule)...}{\Gamma' \vdash M_1' <_{d'} M_2'}$$

where $\Gamma' \vdash \Pi(A_1', B_1') = M_1'$, $\Gamma' \vdash \Pi(A_2', B_2') = M_2'$, $\Gamma' \vdash d' = d_1'$ and

$$d_1' = [f : \Pi(A_1', B_1')]\lambda(A_2', B_2', [x : A_2']e'[x](app(A_1', B_1', f, c'(x))))$$

To see this is the case, by Lemma 4.1.3, we have to show only that

1. $\Gamma' \not\vdash A_2' = A_1'$, and

2. $\Gamma', x : A'_2 \not\vdash B'_1(c'(x)) = B'_2(x)$.

For the first case, since $\Gamma \vdash M_1 = M'_1$ and $\Gamma \vdash M_2 = M'_2$, we have $\Gamma \vdash \Pi(A_1, B_1) = \Pi(A'_1, B'_1)$ and $\Gamma \vdash \Pi(A_2, B_2) = \Pi(A'_2, B'_2)$. Hence, by Church-Rosser in T and conservativity of $T[\mathcal{R}]_0$ over T, we have $\Gamma \vdash A_1 = A'_1$, $\Gamma \vdash B_1 = B'_1$, $\Gamma \vdash A_2 = A'_2$ and $\Gamma \vdash B_2 = B'_2$. As $\Gamma \vdash A_2 <_c A_1$, we have by Lemma 4.1.4, $\Gamma \not\vdash A_2 = A_1$. So $\Gamma' \not\vdash A'_2 = A'_1$.

For the second case, we need to use the induction hypothesis first. Since we already know that the derivations of $\Gamma \vdash A_2 <_c A_1$ and $\Gamma' \vdash A'_2 <_{c'} A'_1$ are sub-derivations of $\Gamma \vdash M_1 <_d M_2$ and $\Gamma' \vdash M'_1 <_{d'} M'_2$, by the induction hypothesis we have $\Gamma \vdash c = c'$. Using this result, a similar argument as in the first case suffices to prove that $\Gamma', x : A'_2 \not\vdash B'_1(c'(x)) = B'_2(x)$.

Now, since the derivations must be of the above forms, by the induction hypothesis again, we have $\Gamma, x : A_2 \vdash e[x] = e'[x]$. Hence $\Gamma \vdash d = d' : (M_1)M_2$.

- Both M_1 and M_2 are computationally equal to Σ-types. The proof of this case is similar to the case that both M_1 and M_2 are computationally equal to Π-types.

□

4.2. Admissibility of Substitution and Transitivity

In the presentation of coercive subtyping in [Luo99], substitution and transitivity are two of the basic rules in the theoretical framework. However, in an implementation of coercive subtyping, if there are infinitely many coercions, these rules usually cannot be directly implemented. For this reason, among others, proving the admissibility of such rules (or their elimination) is always an important task for any subtyping systems.

In our system, we do not take substitution and transitivity as basic rules, but we prove that they are admissible when we extend a WDC by the Π and Σ-subtyping rules.

Theorem 4.2.1 ***(Substitution)*** *If* $\Gamma, x : K, \Gamma' \vdash M_1 <_d M_2 : Type \in \mathcal{C}_\mathcal{M}$ *and* $\Gamma \vdash k : K$, *then* $\Gamma, [k/x]\Gamma' \vdash [k/x]M_1 <_{[k/x]d} [k/x]M_2 : Type \in \mathcal{C}_\mathcal{M}$.

Proof. By induction on derivations.

We consider only the case of the congruence rule (*Cong*) as an example of showing the proof, that is, the last rule of the derivation of $\Gamma, x : K, \Gamma' \vdash M_1 <_d M_2$ is the following:

$$\frac{\begin{array}{c}\Gamma, x : K, \Gamma' \vdash M_1' <_{d'} M_2' \\ \Gamma, x : K, \Gamma' \vdash M_1' = M_1 \quad \Gamma, x : K, \Gamma' \vdash M_2' = M_2 \\ \Gamma, x : K, \Gamma' \vdash d' = d : (M_1')M_2'\end{array}}{\Gamma, x : K, \Gamma' \vdash M_1 <_d M_2}$$

By the induction hypothesis, we have

$$\Gamma, [k/x]\Gamma' \vdash [k/x]M_1' <_{[k/x]d'} [k/x]M_2'$$

By the property of conservativity of $T[\mathcal{R}]_0$ over T and the substitution rules in T, we have $\Gamma, [k/x]\Gamma' \vdash [k/x]M_1' = [k/x]M_1$, $\Gamma, [k/x]\Gamma' \vdash [k/x]M_2' = [k/x]M_2$ and $\Gamma, [k/x]\Gamma' \vdash [k/x]d' = [k/x]d$. Therefore, by the congruence rule, we have

$$\Gamma, [k/x]\Gamma' \vdash [k/x]M_1 <_{[k/x]d} [k/x]M_2$$

□

Now let's consider the theorem of the admissibility of transitivity. In order to prove this theorem, we also need to prove the theorem of weakening.

Theorem 4.2.2 ***(Weakening)*** *If* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, $\Gamma \subseteq \Gamma'$ *and* Γ' *is valid then* $\Gamma' \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$.

Proof. By induction on derivations.

The theorem of weakening in type theory T and the property of conservativity of $T[\mathcal{R}]_0$ over T are also needed in this proof. □

To prove the admissibility of transitivity, the usual measures (e.g. the size of types concerned) do not seem to work (or even to be definable), since types essentially involve computations. We use a measure developed by Aspinall, Companoni and Chen [Che98], which considers only subtyping judgements in a derivation, defined as follows.

Definition 4.2.3 ***(depth)*** *Let* D *be a derivation of a subtyping judgement of the form* $\Gamma \vdash A <_c B : Type$.

$$D: \quad \frac{S_1 \quad \ldots \quad S_n \quad T_1 \quad \ldots \quad T_m}{\Gamma \vdash A <_c B : Type}$$

where $S_1,\ldots,S_n$ *are derivations of subtyping judgements of the form* $\Gamma \vdash M_1 <_d M_2 : Type$ *and,* $T_1,\ldots, T_m$ *are derivations of other forms of judgements. Then we define*

$$depth(D) =_{df} 1 + max\{depth(S_1), \ldots, depth(S_n)\}$$

Specially, if $n = 0$ *then* $depth(D) =_{df} 1$.

The following lemmas show that, from a derivation D of a subtyping judgement J one can always get a derivation D' of the judgement obtained from J by context replacement such that D and D' have the same depth.

Lemma 4.2.4 *If* $\vdash \Gamma = \Gamma'$, $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, *and* D *is a derivation of* $\Gamma \vdash M_1 <_d M_2 : Type$, *then*

1. $\Gamma' \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, *and*
2. *there is a derivation* D' *of* $\Gamma' \vdash M_1 <_d M_2 : Type$ *such that* $depth(D) = depth(D')$.

Proof. By induction on derivations.

- The derivation D is

  $$\frac{\Gamma \vdash M_1 <_d M_2 : Type \ \in \ \mathcal{C}}{\Gamma \vdash M_1 <_d M_2 : Type}$$

  By Lemma 3.4.2, we have $\Gamma' \vdash M_1 <_d M_2 : Type \in \mathcal{C}$ and let D' be

  $$\frac{\Gamma' \vdash M_1 <_d M_2 : Type \ \in \ \mathcal{C}}{\Gamma' \vdash M_1 <_d M_2 : Type}$$

  Then $depth(D) = depth(D') = 1$.
- The last rule of derivation D is

  $$\frac{\begin{array}{c}\Gamma \vdash M_1' <_{d'} M_2' \\ \Gamma \vdash M_1' = M_1 \quad \Gamma \vdash M_2' = M_2 \quad \Gamma \vdash d' = d : (M_1')M_2'\end{array}}{\Gamma \vdash M_1 <_d M_2}$$

  Then, $depth(D) = depth(D_1) + 1$ where D_1 is the derivation of judgement $\Gamma \vdash M_1' <_{d'} M_2'$ in D.

  By the induction hypothesis, we have

  1. $\Gamma' \vdash M_1' <_{d'} M_2' \in \mathcal{C}_{\mathcal{M}}$, and

2. there is a derivation D_2 of $\Gamma' \vdash M_1' <_{d'} M_2'$ such that $depth(D_1) = depth(D_2)$.

By the theorem of context replacement by equal kinds in T and conservativity of $T[\mathcal{R}]_0$ over T, we have $\Gamma' \vdash M_1' = M_1$, $\Gamma' \vdash M_2' = M_2$ and $\Gamma' \vdash d' = d : (M_1')M_2'$. Therefore, using the congruence rule, we have a derivation D'

$$\frac{\begin{array}{c}\Gamma' \vdash M_1' <_{d'} M_2' \\ \Gamma' \vdash M_1' = M_1 \quad \Gamma' \vdash M_2' = M_2 \quad \Gamma' \vdash d' = d : (M_1')M_2'\end{array}}{\Gamma' \vdash M_1 <_d M_2}$$

and $depth(D') = depth(D_2) + 1$. So, $depth(D) = depth(D')$.

- For other cases, similar arguments are sufficient.

$\square$

Lemma 4.2.5 *If* $\Gamma, x : K, \Gamma' \vdash M_1 <_{c_1} M_2 : Type \in \mathcal{C}$ *and* $\Gamma \vdash c_2 : (K')K$ *then*

$$\Gamma, y : K', [c_2(y)/x]\Gamma' \vdash [c_2(y)/x]M_1 <_{[c_2(y)/x]c_1} [c_2(y)/x]M_2 : Type \in \mathcal{C}$$

Proof. By weakening and substitution in the definition of WDC. $\square$

Lemma 4.2.6 *If* $\Gamma, x : K, \Gamma' \vdash M_1 <_{c_1} M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, $\Gamma \vdash c_2 : (K')K$, *and* D *is a derivation of* $\Gamma, x : K, \Gamma' \vdash M_1 <_{c_1} M_2 : Type$, *then*

1. $\Gamma, y : K', [c_2(y)/x]\Gamma' \vdash [c_2(y)/x]M_1 <_{[c_2(y)/x]c_1} [c_2(y)/x]M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, *and*
2. *there is a derivation* D' *of*

$$\Gamma, y : K', [c_2(y)/x]\Gamma' \vdash [c_2(y)/x]M_1 <_{[c_2(y)/x]c_1} [c_2(y)/x]M_2 : Type$$

such that $depth(D) = depth(D')$.

Proof. By induction on derivations and Lemma 4.2.5. The theorem of weakening and substitution in type theory T and the property of conservativity of $T[\mathcal{R}]_0$ over T are also needed in this proof. $\square$

Now, we prove the admissibility of the transitivity rule.

Theorem 4.2.7 *(Transitivity) If* $\Gamma \vdash M_1 <_{d_1} M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, $\Gamma \vdash M_2' <_{d_2} M_3 : Type \in \mathcal{C}_{\mathcal{M}}$ *and* $\Gamma \vdash M_2 = M_2' : Type$, *then* $\Gamma \vdash M_1 <_{d_2 \circ d_1} M_3 : Type \in \mathcal{C}_{\mathcal{M}}$.

Proof. By induction on $depth(D) + depth(D')$, where D and D' are derivations of $\Gamma \vdash M_1 <_{d_1} M_2$ and $\Gamma \vdash M_2' <_{d_2} M_3$, respectively.

In the base case *i.e.* $depth(D) = depth(D') = 1$, we have that the judgements $\Gamma \vdash M_1 <_{d_1} M_2$ and $\Gamma \vdash M_2' <_{d_2} M_3$ are both in $\mathcal{C}$. By Lemma 3.4.2, we have $\Gamma \vdash M_1 <_{d_2 \circ d_1} M_3 \in \mathcal{C}$.

In the step case, if $\Gamma \vdash M_1 <_{d_1} M_2$ and $\Gamma \vdash M_2' <_{d_2} M_3$ are both in $\mathcal{C}$, then a similar argument as the base case suffices. Otherwise, we have that either $\Gamma \vdash M_1 <_{d_1} M_2$ or $\Gamma \vdash M_2' <_{d_2} M_3$ is not in $\mathcal{C}$. Therefore, by Lemma 4.1.2 and the assumption of $\Gamma \vdash M_2 = M_2'$, all of M_1, M_2, M_2' and M_3 are computationally equal to Π-types or Σ-types. We consider only the case that they are equal to Π-types. Suppose that the derivation D and D' be of the following forms (we consider only the more difficult example among the combinations of Π-subtyping rules):

$$\dfrac{\dfrac{\begin{array}{cc} \vdots & \vdots \\ D_1 & D_2 \\ \vdots & \vdots \\ \Gamma \vdash A_2 <_{c_1} A_1 & \Gamma, x : A_2 \vdash B_1(c_1(x)) <_{e_1[x]} B_2(x) \end{array}}{\Gamma \vdash \Pi(A_1, B_1) <_{d_1'} \Pi(A_2, B_2)} \quad \begin{array}{c} \vdots \\ ...(Congruence\ \ rule)... \\ \vdots \end{array}}{\Gamma \vdash M_1 <_{d_1} M_2}$$

where $\Gamma \vdash \Pi(A_1, B_1) = M_1$, $\Gamma \vdash \Pi(A_2, B_2) = M_2$, $\Gamma \vdash d_1' = d_1$ and

$$d_1' = [f : \Pi(A_1, B_1)]\lambda(A_2, B_2, [x : A_2]e_1[x](app(A_1, B_1, f, c_1(x))))$$

and

$$\dfrac{\dfrac{\begin{array}{cc} \vdots & \vdots \\ D_1' & D_2' \\ \vdots & \vdots \\ \Gamma \vdash A_3 <_{c_2} A_2' & \Gamma, x : A_3 \vdash B_2'(c_2(x)) <_{e_2[x]} B_3(x) \end{array}}{\Gamma \vdash \Pi(A_2', B_2') <_{d_2'} \Pi(A_3, B_3)} \quad \begin{array}{c} \vdots \\ ...(Congruence\ \ rule)... \\ \vdots \end{array}}{\Gamma \vdash M_2' <_{d_2} M_3}$$

where $\Gamma \vdash \Pi(A_2', B_2') = M_2'$, $\Gamma \vdash \Pi(A_3, B_3) = M_3$, $\Gamma \vdash d_2' = d_2$ and

$$d_2' = [f : \Pi(A_2', B_2')]\lambda(A_3, B_3, [x : A_3]e_2[x](app(A_2', B_2', f, c_2(x))))$$

We obviously have $depth(D_1) < depth(D)$ and $depth(D_2) < depth(D)$ because D_1 and D_2 are sub-derivations of D; $depth(D_1') < depth(D')$ and $depth(D_2') < depth(D')$ because D_1' and D_2' are sub-derivations of D'.

Now, since $\Gamma \vdash M_2 = M_2'$, we have by Church-Rosser theorem of T and conservativity of $T[\mathcal{R}]_0$ over T, $\Gamma \vdash A_2 = A_2'$ and $\Gamma \vdash B_2 = B_2'$. Since $\Gamma \vdash A_3 <_{c_2} A_2'$ we have $\Gamma \vdash c_2 : (A_3)A_2'$ and $\Gamma \vdash c_2 : (A_3)A_2$. Since $\Gamma, x : A_2 \vdash B_1(c_1(x)) <_{e_1[x]} B_2(x)$, by Lemma 4.2.6, we have $\Gamma, x : A_3 \vdash B_1(c_1(c_2(x))) <_{e_1[c_2(x)]} B_2(c_2(x))$ and there is a derivation D_3 of the judgement $\Gamma, x : A_3 \vdash B_1(c_1(c_2(x))) <_{e_1[c_2(x)]} B_2(c_2(x))$ such that $depth(D_3) = depth(D_2)$.

Now, we have

$$depth(D_1) + depth(D_1') < depth(D) + depth(D')$$

$$depth(D_3) + depth(D_2') < depth(D) + depth(D')$$

By the induction hypothesis, we have that $\Gamma \vdash A_3 <_{c_1 \circ c_2} A_1 \in \mathcal{C}_{\mathcal{M}}$. Since $\Gamma \vdash B_2 = B_2' : (A_2)Type$ and $\Gamma \vdash c_2 : (A_3)A_2$, we have $\Gamma, x : A_3 \vdash B_2(c_2(x)) = B_2'(c_2(x))$. By the induction hypothesis again, we have

$$\Gamma, x : A_3 \vdash B_1(c_1(c_2(x))) <_{e_2[x] \circ e_1[c_2(x)]} B_3(x) \in \mathcal{C}_{\mathcal{M}}$$

So by the Domain-Codomain rule (the third rule in Figure 3.5.1), we have $\Gamma \vdash \Pi(A_1, B_1) <_{d_3} \Pi(A_3, B_3) \in \mathcal{C}_{\mathcal{M}}$, where

$$\begin{aligned} d_3 \;&=_{df}\; [f : \Pi(A_1, B_1)]\lambda(A_3, B_3, \\ &\qquad [x : A_3]e_2[x](e_1[c_2(x)](app(A_1, B_1, f, c_1(c_2(x)))))) \end{aligned}$$

Then

$$\begin{aligned} d_2 \circ d_1 &= [f : \Pi(A_1, B_1)]d_2(d_1(f)) \\ &= [f : \Pi(A_1, B_1)]d_2'(d_1'(f)) \\ &= [f : \Pi(A_1, B_1)]\lambda(A_3, B_3, \\ &\quad [x : A_3]e_2[x](app(A_2', B_2', d_1'(f), c_2(x)))) \\ &= [f : \Pi(A_1, B_1)]\lambda(A_3, B_3, \\ &\quad [x : A_3]e_2[x](e_1[c_2(x)](app(A_1, B_1, f, c_1(c_2(x)))))) \\ &= d_3 \end{aligned}$$

Finally, by the congruence rule, we have $\Gamma \vdash M_1 <_{d_2 \circ d_1} M_3 \in \mathcal{C}_{\mathcal{M}}$. □

Corollary 4.2.8 *$\mathcal{C}_{\mathcal{M}}$ is a WDC.*

Proof. By Lemma 4.1.4 and Theorems 4.1.5, 4.2.1, 4.2.2 and 4.2.7. □

4.3. Algorithm for the coercion search

We have proved the coherence and admissibility of substitution and transitivity for the subtyping rules of Π-types and Σ-types. We can be sure that the coercion search is decidable for $\mathcal{C}_{\mathcal{M}}$ if it is decidable in $\mathcal{C}$. We shall in this section give a sound and complete algorithm to do so.

4.3.1. Algorithm $Alg(\Gamma, M_1, M_2)$ for $T[\mathcal{R}]_0$

If it is decidable to check whether there is a judgement $\Gamma \vdash A <_c B \in \mathcal{C}$ when arbitrary Γ, A and B are given, then we say that *the Coercion Search is decidable* in $\mathcal{C}$.

Supposing the coercion search is decidable in $\mathcal{C}$, we give an algorithm $Alg(\Gamma, M_1, M_2)$ for $\mathcal{C}_{\mathcal{M}}$ to check whether there is a judgement $\Gamma \vdash M_1 <_d M_2 \in \mathcal{C}_{\mathcal{M}}$ when arbitrary Γ, M_1 and M_2 are given. If so, $Alg(\Gamma, M_1, M_2) := d$' for some d' and $\Gamma \vdash d = d'$, otherwise $Alg(\Gamma, M_1, M_2) :=\perp$.

1. If Γ is a valid context, M_1 and M_2 are well-typed type then go to 2. Otherwise $Alg(\Gamma, M_1, M_2) :=\perp$.
2. If there is a judgement $\Gamma \vdash M_1 <_d M_2 \in \mathcal{C}$ then $Alg(\Gamma, M_1, M_2) := d$. Otherwise, go to 3.

3. Compute M_1 and M_2 to weak normal form $wnf(M_1)$ and $wnf(M_2)$. If both $wnf(M_1)$ and $wnf(M_2)$ are Π-types or Σ-types then go to 4. Otherwise $Alg(\Gamma, M_1, M_2) :=\perp$.
4. If $wnf(M_1) \equiv \Pi(A_1, B_1)$ and $wnf(M_2) \equiv \Pi(A_2, B_2)$ then go to 5. Otherwise $wnf(M_1) \equiv \Sigma(A_1, B_1)$ and $wnf(M_2) \equiv \Sigma(A_2, B_2)$ go to 6.
5. If $\Gamma \vdash A_1 = A_2$ and $Alg((\Gamma, x : A_2), B_1(x), B_2(x)) := e[x]$ $(x \notin FV(\Gamma))$, then

$$\begin{aligned} Alg(\Gamma, M_1, M_2) \quad := \quad & [f : \Pi(A_1, B_1)]\lambda(A_2, B_2, \\ & [x : A_1]e[x] \circ app(A_1, B_1, f, x)) \end{aligned}$$

If $Alg(\Gamma, A_2, A_1) := c$ and $\Gamma, x : A_2 \vdash B_1(c(x)) = B_2(x)$, then

$$\begin{aligned} Alg(\Gamma, M_1, M_2) \quad := \quad & [f : \Pi(A_1, B_1)]\lambda(A_2, B_2 \circ c, \\ & app(A_1, B_1, f) \circ c) \end{aligned}$$

If $Alg(\Gamma, A_2, A_1) := c$ and $Alg((\Gamma, x : A_2), B_1(c(x)), B_2(x)) := e[x]$, then

$$\begin{aligned} Alg(\Gamma, M_1, M_2) \quad := \quad & [f : \Pi(A_1, B_1)]\lambda(A_2, B_2, \\ & [x : A_2]e[x](app(A_1, B_1, f, c(x)))) \end{aligned}$$

Otherwise $Alg(\Gamma, M_1, M_2) :=\perp$.
6. If $\Gamma \vdash A_1 = A_2$ and $Alg((\Gamma, x : A_2), B_1(x), B_2(x)) := e[x]$, then

$$\begin{aligned} Alg(\Gamma, M_1, M_2) \quad := \quad & [x : \Sigma(A_1, B_1)]pair(A_2, B_2, \\ & \pi_1(A_1, B_1, x), \\ & e[\pi_1(A_1, B_1, x)](\pi_2(A_1, B_1, x))) \end{aligned}$$

If $Alg(\Gamma, A_1, A_2) := c$ and $\Gamma, x : A_1 \vdash B_1(x) = B_2(c(x))$, then

$$\begin{aligned} Alg(\Gamma, M_1, M_2) \quad := \quad & [x : \Sigma(A_1, B_1)]pair(A_2, B_2, \\ & c(\pi_1(A_1, B_1, x)), \\ & \pi_2(A_1, B_1, x)) \end{aligned}$$

If $Alg(\Gamma, A_1, A_2) := c$ and $Alg((\Gamma, x : A_1), B_1(x), B_2(c(x))) := e[x]$, then

$$\begin{aligned} Alg(\Gamma, M_1, M_2) \quad := \quad & [x : \Sigma(A_1, B_1)]pair(A_2, B_2, \\ & c(\pi_1(A_1, B_1, x)), \\ & e[\pi_1(A_1, B_1, x)](\pi_2(A_1, B_1, x))) \end{aligned}$$

Otherwise $Alg(\Gamma, M_1, M_2) :=\perp$.

4.3.2. Soundness and Completeness

Theorem 4.3.1 ***(Soundness)*** *If* $Alg(\Gamma, M_1, M_2) =\perp$ *then there isn't any judgement* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$. *If* $Alg(\Gamma, M_1, M_2) := d$ *then there is a judgement* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$.

Proof. For the first part, we proceed by contradiction and prove that if $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$ then $Alg(\Gamma, M_1, M_2) \neq\perp$. For the second part, we follow the algorithm step by step and construct a derivation of $\Gamma \vdash M_1 <_d M_2 : Type$. □

Theorem 4.3.2 ***(Completeness)*** *For any judgement* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$, *there is a* d' *such that* $Alg(\Gamma, M_1, M_2) = d'$ *and* $\Gamma \vdash d = d' : (M_1)M_2$.

Proof. By induction on derivations and Lemma 4.1.2, 4.1.3 and 4.1.4. □

4.3.3. Decidability of the Coercion Search in $T[\mathcal{R}]_0$

Theorem 4.3.3 *If the coercion search is decidable in* $\mathcal{C}$, *so is in* $\mathcal{C}_{\mathcal{M}}$, i.e. *it is decidable whether there is a judgement* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_{\mathcal{M}}$ *for arbitrary* Γ, M_1 *and* M_2.

Proof. By Theorem 4.3.1 and Theorem 4.3.2. □

4.4. Subtyping rules for ST-form

In the section 4.1 and 4.2, we have proved the coherence and admissibility of the transitivity rule for the subtyping rules of Π-types and Σ-types. The question now is: do we have suitable subtyping rules for other parameterised inductive types, the system extended by which also keeps the good properties, such as coherence and admissibility of the transitivity rule (*Trans*)? The answer is yes. In this section, we shall

give two more examples to demonstrate how coercions are defined in the subtyping rules for those parameterised inductive types generated by ST-form.

$$\frac{\Gamma \vdash A <_{c_1} A' : Type \quad \Gamma \vdash B <_{c_2} B' : Type \quad \Gamma \vdash C <_{c_3} C' : Type}{\Gamma \vdash Trio(A,B,C) <_{d_1} Trio(A',B',C') : Type}$$

where

$$\begin{aligned} d_1 &= [z : Trio(A,B,C)]trio(A',B',C',c_1(\pi_{Trio1}(A,B,C,z)), \\ &\quad c_2(\pi_{Trio2}(A,B,C,z)), c_3(\pi_{Trio3}(A,B,C,z))) \end{aligned}$$

$$\frac{\Gamma \vdash A <_{c_1} A' : Type \quad \Gamma \vdash B = B' : Type \quad \Gamma \vdash C = C' : Type}{\Gamma \vdash Trio(A,B,C) <_{d_2} Trio(A',B',C') : Type}$$

where the definition of d_2 is similar to that of d_1, just replacing c_2 and c_3 with identity functions. We shall omit the definitions of the coercions in the following rules.

$$\frac{\Gamma \vdash A = A' : Type \quad \Gamma \vdash B <_{c_2} B' : Type \quad \Gamma \vdash C = C' : Type}{\Gamma \vdash Trio(A,B,C) <_{d_3} Trio(A',B',C') : Type}$$

$$\frac{\Gamma \vdash A = A' : Type \quad \Gamma \vdash B = B' : Type \quad \Gamma \vdash C <_{c_3} C' : Type}{\Gamma \vdash Trio(A,B,C) <_{d_4} Trio(A',B',C') : Type}$$

$$\frac{\Gamma \vdash A <_{c_1} A' : Type \quad \Gamma \vdash B <_{c_2} B' : Type \quad \Gamma \vdash C = C' : Type}{\Gamma \vdash Trio(A,B,C) <_{d_5} Trio(A',B',C') : Type}$$

$$\frac{\Gamma \vdash A <_{c_1} A' : Type \quad \Gamma \vdash B = B' : Type \quad \Gamma \vdash C <_{c_3} C' : Type}{\Gamma \vdash Trio(A,B,C) <_{d_6} Trio(A',B',C') : Type}$$

$$\frac{\Gamma \vdash A = A' : Type \quad \Gamma \vdash B <_{c_2} B' : Type \quad \Gamma \vdash C <_{c_3} C' : Type}{\Gamma \vdash Trio(A,B,C) <_{d_7} Trio(A',B',C') : Type}$$

FIGURE 4.4.1. Subtyping rules for non-dependent trio

Example 4.4.1 *On page 39, we defined the projection operators (π_{Trio1}, π_{Trio2} and π_{Trio3}) for the type of non-dependent trio (Trio). Now we use these projection operators to define coercions for the subtyping rules*

as in Figure 4.4.1. As we proved for Π*-types and* Σ*-types, the coherence holds and the normal transitivity rule is admissible if we add the subtyping rules for Trio-types into the system* $T[\mathcal{R}]_0$. *We omit the proof here.*

Remark 4.4.2 *One may choose inductively defined coercions, for example, re-define* d_1 *as*

$$\begin{aligned} d_1' \;=_{df}\;& \mathcal{E}_{Trio}(A, B, C, [z : Trio(A, B, C)]Trio(A', B', C'), \\ & [a : A][b : B][c : C]trio(A', B', C', c_1(a), c_2(b), c_3(c))) \end{aligned}$$

However, the transitivity rule (Trans) fails to be admissible and the reason is the same as that for Π*-types and* Σ*-types in Example 3.5.2 and 3.5.3.*

Notation 4.4.3 *We shall write* $\Gamma \vdash A \leq_c B : Type$ *to indicate that both* $\Gamma \vdash A <_c B : Type$ *and* $\Gamma \vdash A \doteq B : Type$ *may happen.*
If $\Gamma \vdash A = B : Type$ *then* $c \equiv id_A =_{df} [x : A]x$.
Note that $\Gamma \vdash A \leq_c B : Type$ *itself is not a judgement.*

With the above notation, we can simply use the following form to represent all seven rules in Figure 4.4.1.

$$\frac{\Gamma \vdash A \leq_{c_1} A' : Type \quad \Gamma \vdash B \leq_{c_2} B' : Type \quad \Gamma \vdash C \leq_{c_3} C' : Type}{\Gamma \vdash Trio(A, B, C) <_{d_{Trio}} Trio(A', B', C') : Type}$$

where c_i $(i = 1, 2, 3)$ is a coercion or an identity function, and at least one c_i is a coercion, and

$$\begin{aligned} d_{Trio} \;=\;& [z : NT(A, B, C)]trio(A', B', C', c_1(\pi_{Trio1}(A, B, C, z)), \\ & c_2(\pi_{Trio2}(A, B, C, z)), c_3(\pi_{Trio3}(A, B, C, z))) \end{aligned}$$

satisfying the following equation:

$$d_{Trio}(trio(A, B, C, a, b, c)) = trio(A', B', C', c_1(a), c_2(b), c_3(c))$$

Example 4.4.4 *On page 40, we defined the function operators* (π_{SPL1} *and* π_{SPL2}) *for the type of pairs in which the first component are functions and the second are objects of a type. Now we use these function operators to define coercions for the subtyping rules as follows.*

$$\frac{\Gamma \vdash A' \leq_{c_1} A : Type \quad \Gamma \vdash B \leq_{c_2} B' : Type \quad \Gamma \vdash C \leq_{c_3} C' : Type}{\Gamma \vdash SPL(A, B, C) <_{d_{SPL}} SPL(A', B', C') : Type}$$

where c_i $(i = 1, 2, 3)$ is a coercion or an identity function, and at least one c_i is a coercion, and

$$\begin{aligned} d_{SPL} &= [z : SPL(A,B,C)]spl(A',B',C', \\ &\quad [x : A']c_2(\pi_{SPL1}(A,B,C,z,c_1(x))), c_3(\pi_{SPL2}(A,B,C,z,))) \end{aligned}$$

satisfying the following equation:

$$d_{SPL}(spl(A,B,C,g,c)) = spl(A',B',C',c_2 \circ g \circ c_1, c_3(c))$$

Remark 4.4.5 *From these examples, we may see that the function operators play a very important role in the definitions of coercions. The transitivity rule is admissible for the subtyping rules in these examples, and the proof method is the same as that in section 4.2.*

In general, we have the following conjecture:

- For the parameterised inductive types generated by ST-form, which have only one constructor, if the coercions of the subtyping rules are defined by using their function operators, then the coherence of the system $T[\mathcal{R}]_0$ holds and the normal transitivity rule is admissible.

Although it is complex to give a general form of subtyping rules for parameterised inductive types generated by ST-form, we can clearly see why the normal transitivity rule is admissible. If a coercion d is defined by using function operators then for a ***variable*** x, we can compute $d(x)$ to a ***canonical object***. For example,

$$\begin{aligned} d_{Trio}(x) &= trio(A',B',C',c_1(\pi_{Trio1}(A,B,C,x)), \\ &\quad c_2(\pi_{Trio2}(A,B,C,x)), c_3(\pi_{Trio3}(A,B,C,x))) \end{aligned}$$

Because of this property, the normal transitivity rule is admissible. Contrarily, if d is defined inductively, $d(x)$ cannot be computed further if x is a variable. This is also the reason why the normal transitivity rule is not admissible.

CHAPTER 5

Weak Transitivity

In this chapter, we study the notion of *Weak Transitivity*, consider suitable subtyping rules for certain parameterised inductive types and prove its coherence and the admissibility of substitution and weak transitivity.

In Chapter 4, we studied the property of the subtyping rules for Π-types and Σ-types. A common factor of these two data types is that they have only one constructor and some special function operators over them can be defined, π_1 and π_2 for Σ-types and *app* for Π-types. We don't have to define the coercions inductively and instead, define them by using the special function operators. Hence the normal transitivity rule $(Trans)$ is admissible.

Now, a question is: is the transitivity rule still admissible for those inductive types that consist of more than one constructor? We will give an example[1] in the following section to answer this question.

5.1. A problem with transitivity

The normal transitivity rule

$$(Trans) \qquad \frac{\Gamma \vdash A <_c B : Type \quad \Gamma \vdash B <_{c'} C : Type}{\Gamma \vdash A <_{c' \circ c} C : Type}$$

as presented in Chapter 3 basically says that the composition of two coercions is also the coercion corresponding to transitivity.

However, the above transitivity rule is sometimes too strong (in intensional type theories). For some parameterised inductive data types together with their natural subtyping rules, especially when an inductive type has more than one constructor, the above rule fails to be admissible or eliminatable. We give the following example to show the problem.

[1] There are three key examples in this chapter. Understanding these examples is a good way to understand this chapter concerning weak transitivity.

Example 5.1.1 *This is the first key example to show the problem with transitivity. If we introduce the subtyping rule for lists then the transitivity rule (Trans) fails to be admissible and, if we add Trans into the system, the coherence requirement fails to be satisfied.*

We introduce the following subtyping rule for the inductive data type of lists $List(A)$ parameterised by its element type A.

$$\frac{\Gamma \vdash A <_c B : Type}{\Gamma \vdash List(A) <_{d_{List}} List(B) : Type}$$

where $d_{List} = map_{List}(A, B, c)$ (the detailed definitions of List and map_{List} is on page 32) such that

$$d_{List}(nil(A)) = nil(B)$$

$$d_{List}(cons(A, a, l)) = cons(B, c(a), d_{List}(l))$$

Then the transitivity rule (Trans) fails to be admissible and, if we add it into the system, the coherence requirement fails to be satisfied.

To see this, suppose we have $\Gamma \vdash F <_{c_1} E : Type$ and $\Gamma \vdash E <_{c_2} N : Type$, and by the transitivity rule (Trans), we also have $\Gamma \vdash F <_{c_2 \circ c_1} N : Type$.

By the above subtyping rule for lists, we have respectively

$$\Gamma \vdash List(F) <_{d_1} List(E) : Type$$

$$\Gamma \vdash List(E) <_{d_2} List(N) : Type$$

$$\Gamma \vdash List(F) <_{d_3} List(N) : Type$$

where

$$\begin{aligned} d_1 &= map_{List}(F, E, c_1) \\ d_2 &= map_{List}(E, N, c_2) \\ d_3 &= map_{List}(F, N, c_2 \circ c_1) \end{aligned}$$

By the transitivity rule (Trans), we also have

$$\Gamma \vdash List(F) <_{d_2 \circ d_1} List(N) : Type$$

Now, the problem is that, in an intensional type theory, d_3 and $d_2 \circ d_1$ are not computationally equal i.e.

$$\Gamma \not\vdash d_3 = d_2 \circ d_1 : (List(F))List(N)$$

This means that we have two coercions (d_3 and $d_2 \circ d_1$) between $List(F)$ and $List(N)$, but they are not computationally equal (and hence coherence fails), although we know that they are propositionally equal in the sense that the following proposition is provable in an intensional type theory:

$$\forall l : List(F).Eq(List(N), d_3(l), d_2(d_1(l)))$$

5.2. Weak transitivity

Rather than the (strong) transitivity rule ($Trans$), we introduce a new concept, *Weak Transitivity,* which can informally be represented by the following rule:

$$(WTrans) \qquad \frac{\Gamma \vdash A <_c B : Type \quad \Gamma \vdash B <_{c'} C : Type}{\Gamma \vdash A <_{c''} C : Type}$$

This rule says that, if $A <_c B$ and $B <_{c'} C$, then $A <_{c''} C$ for some coercion c''. The essential difference compared with the (strong) transitivity rule ($Trans$) is that we are only more concerned about the existence of c'' and such weak transitivity should be better suited to a wider application; that is, many natural subtyping rules (for example, the subtyping rule for lists) are suitable for weak transitivity ($WTrans$) but not for the (strong) transitivity rule ($Trans$).

5.2.1. Meta-level equality requirement

We don't want the coercion c'' in the weak transitivity rule ($WTrans$) to be an arbitrary one. Otherwise, this coercion could be very bizarre and lose the general meaning. In the strong transitivity rule ($Trans$), c'' is the composition of c' and c ($c' \circ c$). In the weak transitivity rule ($WTrans$), we require that c'' must somehow be equal to $c' \circ c$. There are two choices: one is propositional equality in the sense that the proposition $\forall x : A.Eq(C, c''(x), c'(c(x)))$ is provable in an intensional type theory; another is extensional equality in the sense that c'' and $c' \circ c$ are judgementally equal in an extensional type theory. Of course, if the proposition $\forall x : A.Eq(C, c''(x), c'(c(x)))$ is provable in an intensional type theory, c'' and $c' \circ c$ are judgementally equal in an extensional type theory. However, for some inductive data types with their subtyping rules, c'' and $c' \circ c$ are not propositionally equal. We give the following example to explain why we regard the extensional equality of c'' and

$c' \circ c$ in the weak transitivity rule ($WTrans$) as a meta-level equality requirement.

Example 5.2.1 *This is the second key example concerning the meta-level equality requirement. Consider the following derivations regarding the subtype relation between function types ($A \to B$) parameterised by type A and B. These derivations basically say that if B is a subtype of B' then $A \to B$ is subtype of $A \to B'$. (The constructor, eliminator and computation rule for the function types ($\to$) can be found on page 33)*

$$\frac{\Gamma \vdash B <_{c_1} B' : Type}{\Gamma \vdash A \to B <_{d_1} A \to B' : Type}$$
$$\frac{\Gamma \vdash B' <_{c_2} B'' : Type}{\Gamma \vdash A \to B' <_{d_2} A \to B'' : Type}$$
$$\frac{\Gamma \vdash B <_{c_3} B'' : Type}{\Gamma \vdash A \to B <_{d_3} A \to B'' : Type}$$

where d_1, d_2 and d_3 satisfy the following equations:

$$d_1(lam(A, B, g)) = lam(A, B', [x : A]c_1(g(x)))$$
$$d_2(lam(A, B', h)) = lam(A, B'', [x : A]c_2(h(x)))$$
$$d_3(lam(A, B, g)) = lam(A, B'', [x : A]c_3(g(x)))$$

Then we have

$$d_2(d_1(lam(A, B, g))) = lam(A, B'', [x : A]c_2(c_1(g(x))))$$

Now, let's compare d_3 and $d_2 \circ d_1$ and, the terms in the right hand side $lam(A, B'', [x : A]c_3(g(x)))$ and $lam(A, B'', [x : A]c_2(c_1(g(x))))$. Even if we assume that c_3 and $c_2 \circ c_1$ are propositionally equal i.e. *we have a proof of $\forall x : A.Eq(B'', c_3(x), c_2(c_1(x)))$, it is impossible to prove that $lam(A, B'', [x : A]c_3(g(x)))$ and $lam(A, B'', [x : A]c_2(c_1(g(x))))$ are equal. Hence it is impossible to prove the proposition $\forall f : A \to B.Eq(A \to B'', d_3(f), d_2(d_1(f)))$,* i.e. *$d_3$ and $d_2 \circ d_1$ are not propositionally equal.*

Remark 5.2.2 *It is worth remarking that, in the above example, if we consider extensional equality and assume that c_3 is extensionally equal to $c_2 \circ c_1$, then d_3 and $d_2 \circ d_1$ are extensionally equal.*

5.2.2. Coercion dependency

Through my investigation, I also found out that weak transitivity does not hold for all combinations of the subtyping rules for parameterised inductive types. For example, it's admissibility fails for subtyping rules of Σ-types. There are three subtyping rules for Σ-types as in Figure 3.5.2. We list two of them here; one is

$$\frac{\Gamma \vdash A <_c A' : Type \quad \Gamma, x : A \vdash B(x) <_{e[x]} B'(c(x)) : Type}{\Gamma \vdash \Sigma(A, B) <_d \Sigma(A', B') : Type}$$

and another is

$$\frac{\Gamma \vdash A <_c A' : Type \quad \Gamma, x : A \vdash B(x) = B'(c(x)) : Type}{\Gamma \vdash \Sigma(A, B) <_d \Sigma(A', B') : Type}$$

which is equivalent to the First-Component rule in Figure 3.5.2. From the above two rules, we can see that the coercion c in the first premise occurs in the second premise. We call this *Coercion Dependency.* The weak transitivity cannot be proved. For instance, in order to prove that $\Sigma(A_1, B_1) < \Sigma(A_2, B_2)$ and $\Sigma(A_2, B_2) < \Sigma(A_3, B_3)$ imply $\Sigma(A_1, B_1) < \Sigma(A_3, B_3)$ (coercions and some other details are omitted here), we would proceed by induction on derivations. One of the cases is that the last steps of the derivations of $\Sigma(A_1, B_1) < \Sigma(A_2, B_2)$ and $\Sigma(A_2, B_2) < \Sigma(A_3, B_3)$ use the second rule above:

$$\frac{A_1 <_{c_1} A_2 \quad x : A_1 \vdash B_1(x) = B_2(c_1(x))}{\Sigma(A_1, B_1) < \Sigma(A_2, B_2)}$$

and

$$\frac{A_2 <_{c_2} A_3 \quad y : A_2 \vdash B_2(y) = B_3(c_2(y))}{\Sigma(A_2, B_2) < \Sigma(A_3, B_3)}$$

By induction hypothesis, $A_1 <_{c_3} A_3$ is derivable for some c_3, but c_3 is not (necessarily) computationally equal to $c_2 \circ c_1$.
Since $x : A_1 \vdash c_1(x) : A_2$ and $y : A_2 \vdash B_2(y) = B_3(c_2(y))$
we have $x : A_1 \vdash B_2(c_1(x)) = B_3(c_2(c_1(x)))$ and hence
$x : A_1 \vdash B_1(x) = B_3(c_2(c_1(x)))$ is derivable.
However, $x : A_1 \vdash B_1(x) = B_3(c_3(x))$ is not necessarily derivable and how to derive $\Sigma(A_1, B_1) < \Sigma(A_3, B_3)$ becomes a problem of the proof.

In fact, the following counter example shows that weak transitivity fails when we combine the subtyping rules for Σ-types and lists.

Example 5.2.3 *This is the third key example regarding weak transitivity. If we combine the subtyping rules for lists and Σ-types then weak*

transitivity fails, i.e. *even if* $M_1 <_{e_1} M_2$ *and* $M_2 <_{e_2} M_3$ *are derivable, but* $M_1 <_{e_3} M_3$ *is not derivable for any* e_3.

Assume that we have some type constants A_1, A_2, A_3 *and a constant* B_3 *of kind* $(List(A_3))Type$ *in an empty context.*

$$\begin{aligned} A_1 &: Type \\ A_2 &: Type \\ A_3 &: Type \\ B_3 &: (List(A_3))Type \end{aligned}$$

We also assume that we have the following three coercions in the empty context. A WDC C is generated by these coercions and the congruence rule (Cong).

$$\vdash A_1 <_{c_1} A_2 : Type$$

$$\vdash A_2 <_{c_2} A_3 : Type$$

$$\vdash A_1 <_{c_2 \circ c_1} A_3 : Type$$

By the subtyping rule for lists, we have:

$$\vdash List(A_1) <_{d_1} List(A_2) : Type$$

$$\vdash List(A_2) <_{d_2} List(A_3) : Type$$

$$\vdash List(A_1) <_{d_3} List(A_3) : Type$$

where d_1, d_2 *and* d_3 *are defined as the same as in Example 5.1.1. Note that* $\vdash d_3 \neq d_2 \circ d_1 : (List(A_1))List(A_3)$ i.e. d_3 *and* $d_2 \circ d_1$ *are NOT computationally equal.*

Since $B_3 \circ d_2 : (List(A_2))Type$, *by the First-Component rule for* Σ*-types, we have:*

$$\vdash \Sigma(List(A_1), B_3 \circ d_2 \circ d_1) <_{e_1} \Sigma(List(A_2), B_3 \circ d_2) : Type$$

$$\vdash \Sigma(List(A_2), B_3 \circ d_2) <_{e_2} \Sigma(List(A_3), B_3) : Type$$

Here, we omit the definition of e_1 *and* e_2.

Now, the question is: is the following judgement derivable for some e_3 *because the above two are derivable?*

$$\vdash \Sigma(List(A_1), B_3 \circ d_2 \circ d_1) <_{e_3} \Sigma(List(A_3), B_3) : Type$$

The answer is NO. We prove the answer by contradiction. If it is derivable, then the derivations must have the following form:

$$\frac{\vdots \qquad \vdots}{\dfrac{\dfrac{\vdash A <_c A' : Type \quad \vdash B : (A')Type}{\vdash \Sigma(A, B \circ c) <_{e_4'} \Sigma(A', B) : Type}}{\vdots}}$$

$$...(Congruence \quad rules)...$$

$$\frac{\vdots}{\vdash \Sigma(List(A_1), B_3 \circ d_2 \circ d_1) <_{e_3} \Sigma(List(A_3), B_3) : Type}$$

where $\vdash \Sigma(List(A_1), B_3 \circ d_2 \circ d_1) = \Sigma(A, B \circ c)$ *and* $\vdash \Sigma(List(A_3), B_3) = \Sigma(A', B)$.

By the properties of Church-Rosser of the original type theory, we have $\vdash List(A_1) = A$, $\vdash B_3 \circ d_2 \circ d_1 = B \circ c : (A)Type$, $\vdash List(A_3) = A'$ *and* $\vdash B_3 = B : (A')Type$. *Since* B_3 *is a constant, the normal form of* B *is* B_3. *Hence* $\vdash c = d_2 \circ d_1 : (A)A'$ *(computationally). Since the coherence of the system can be proved by induction on derivations as in Section 4.1, we have* $\vdash c = d_3 : (A)A'$. *Therefore, we have* $\vdash d_3 = d_2 \circ d_1 : (A)A'$. *This is a contradiction.* □

The fact that weak transitivity fails because of coercion dependency leads us to consider the subtyping rules for some restricted forms of schemata which disallow that a coercion in one premise occurs in another premise.

5.3. Weak transitivity schemata

Now we give a definition of *WT-schema.* Consider parameterised inductive types generated by the following form (under a valid context Γ):

$$\mathcal{T} =_{df} [Y_1 : P_1]...[Y_n : P_n]\mathcal{M}[\overline{\Theta}]$$

where P_1 is a kind in Γ, P_2 is a kind in $\Gamma, Y_1 : P_1$ and so on, and P_n is a kind in $\Gamma, Y_1 : P_1, ..., Y_{n-1} : P_{n-1}$; $\overline{\Theta} \equiv < \Theta_1, ..., \Theta_m >$ $(m \in \omega)$ is a finite sequence of WT-schemata in $\Gamma, Y_1 : P_1, ..., Y_n : P_n$ with respect to a placeholder X of *Type*. In order to define WT-schema, we first define WT small kind and WT strictly positive operator.

Notation 5.3.1 *We shall write $\overline{Y} \in FV(M)$ and $\overline{Y} \notin FV(M)$ to mean that 'some of the parameters occur free in M' and 'none of the parameters occurs free in M', respectively.*

Definition 5.3.2 ***(WT small kind)*** *A WT small kind K in Γ, with respect to the parameters $Y_1, ..., Y_n$, is one of the following form:*

1. $K \equiv El(A)$,
2. $K \equiv (y : K_1)K_2$ where
   (a) if $y \notin FV(K_2)$ then K_1 and K_2 are WT small kinds in Γ.
   (b) if $y \in FV(K_2)$ then none of the parameters occur free in K_1 (*i.e.* $\overline{Y} \notin FV(K_1)$) and K_2 is a WT small kind in $\Gamma, y : K_1$.

Definition 5.3.3 ***(WT strictly positive operator)*** *A WT strictly positive operator in Γ, with respect to the placeholder X of Type and the parameters $Y_1, ..., Y_n$, is of one of the following forms:*

1. $\Phi \equiv X$, or
2. $\Phi \equiv (x : K)\Phi_0$, where K is a WT small kind in Γ and Φ_0 is a WT strictly positive operator in $\Gamma, x : K$; and if $x \in FV(\Phi_0)$ then none of the parameters occur free in K *i.e.* $\overline{Y} \notin FV(K)$.

Definition 5.3.4 ***(WT-schema)*** *A WT-schema Θ in Γ, with respect to the placeholder X of Type and the parameters $Y_1, ..., Y_n$, is of one of the following forms:*

1. $\Theta \equiv X$, or
2. $\Theta \equiv (x : K)\Theta_0$, where K is a WT small kind in Γ and Θ_0 is a WT-schema in $\Gamma, x : K$; and if $x \in FV(\Theta_0)$ then none of the parameters occurs free in K *i.e.* $\overline{Y} \notin FV(K)$.
3. $\Theta \equiv (x : \Phi)\Theta_0$, where $x \notin FV(\Theta_0)$, Φ is a WT strictly positive operator in Γ and Θ_0 is an WT-schema in Γ.

Remark 5.3.5 *We have the following property for a WT-schema Θ.*

If $(x : M_1)M_2$ is a subterm of Θ and x occurs free in M_2, then M_1 does not contain any of the parameters.

The above notion of WT-schema covers a large class of parameterised inductive data types such as lists, (non-dependent) function types, binary trees in Example 2.3.7. We give two examples here, *Maybe* and *Either* types, which are frequently found in functional programming languages such as Haskell [Tho99].

Example 5.3.6 *With the general methods given in Section 2.3, constants and computation rules for Maybe and Either-types are declared as follows.*

1. *Maybe types:* $Maybe =_{df} [A : Type]\mathcal{M}[X, (A)X]$
   Declare the following constants:

$$
\begin{aligned}
Maybe &: (A : Type)Type \\
nothing &: (A : Type)Maybe(A) \\
just &: (A : Type)(A)Maybe(A) \\
\mathcal{E}_{Maybe} &: (A : Type)(C : (Maybe(A))Type) \\
&\quad (C(nothing(A)))((a : A)C(just(A, a))) \\
&\quad (z : Maybe(A))C(z)
\end{aligned}
$$

   and assert the following computation rules:

$$
\begin{aligned}
\mathcal{E}_{Maybe}(A, C, c, f, nothing(A)) &= c : C(nothing(A)) \\
\mathcal{E}_{Maybe}(A, C, c, f, just(A, a)) &= f(a) : C(just(A, a))
\end{aligned}
$$

2. *Disjoint union:* $Either =_{df} [A : Type][B : Type]\mathcal{M}[(A)X, (B)X]$
   Declare the following constants:

$$
\begin{aligned}
Either &: (A : Type)(B : Type)Type \\
left &: (A : Type)(B : Type)(A)Either(A, B) \\
right &: (A : Type)(B : Type)(B)Either(A, B) \\
\mathcal{E}_{Either} &: (A : Type)(B : Type) \\
&\quad (C : (Either(A, B))Type) \\
&\quad ((a : A)C(left(A, B, a))) \\
&\quad ((b : B)C(right(A, B, b))) \\
&\quad (z : Either(A, B))C(z)
\end{aligned}
$$

   and assert the following computation rules:

$$
\begin{aligned}
&\mathcal{E}_{Either}(A, B, C, f_1, f_2, left(A, B, a)) \\
&\quad = f_1(a) : C(left(A, B, a))
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}_{Either}(A, B, C, f_1, f_2, right(A, B, b)) \\
&\quad = f_2(b) : C(right(A, B, b))
\end{aligned}
$$

Remark 5.3.7 *WT-schema excludes those parameterised inductive data types such as Σ-types and Π-types because their subtyping rules have coercion dependency.*

5.4. General subtyping rules for WT-schemata

In this section, we consider how to define subtyping rules and the associated coercions for any parameterised types generated by the form:

$$\mathcal{T} =_{df} [Y_1 : P_n]...[Y_n : P_n]\mathcal{M}[\overline{\Theta}]$$

where $\overline{\Theta} \equiv <\Theta_1, ..., \Theta_m> (m \in \omega)$ is a finite sequence of WT-schemata defined in last section.

Before we give a general form of subtyping rules we give the following examples to demonstrate what the subtyping rules and associated coercions are.

Example 5.4.1 *In this example, we give subtyping rules and associated coercions for lists, Maybe types, binary trees, Either types and Function types. Their constructors, eliminators and computation rules can be found in Example 2.3.7 and Example 5.3.6.*

1. *As given in Section 5.1, the subtyping rule for lists is:*

$$\frac{\Gamma \vdash A <_c B : Type}{\Gamma \vdash List(A) <_{d_{List}} List(B) : Type}$$

where

$$\begin{aligned} d_{List} &=_{df} map(A, B, c) \\ &=_{df} \mathcal{E}_{List}(A, [l : List(A)]List(B), nil(B), \\ &\qquad [a : A][l : List(A)][l' : List(B)]cons(B, c(a), l')) \end{aligned}$$

such that

$$\begin{aligned} d_{List}(nil(A)) &= nil(B) \\ d_{List}(cons(A, a, l)) &= cons(B, c(a), d_{List}(l)) \end{aligned}$$

2. *Subtyping rule for Maybe types:*

$$\frac{\Gamma \vdash A <_c B : Type}{\Gamma \vdash Maybe(A) <_{d_{Maybe}} Maybe(B) : Type}$$

where

$$d_{Maybe} =_{df} \mathcal{E}_{Maybe}(A, [z : Maybe(A)]Maybe(B), nothing(B), [a : A]just(B, c(a)))$$

such that

$$\begin{aligned} d_{Maybe}(nothing(A)) &= nothing(B) \\ d_{Maybe}(just(A, a)) &= just(B, c(a)) \end{aligned}$$

3. *Subtyping rule for Binary trees:*

$$\frac{\Gamma \vdash A <_c B : Type}{\Gamma \vdash BTree(A) <_{d_{BTree}} BTree(B) : Type}$$

where

$$\begin{aligned} d_{BTree} =_{df} \; & \mathcal{E}_{BTree}(A, [z : BTree(A)]BTree(B), \\ & empty(B), [a : A][t_1 : BTree(A)][t_1' : BTree(B)] \\ & [t_2 : BTree(A)][t_2' : BTree(B)]mk(B, c(a), t_1', t_2')) \end{aligned}$$

such that

$$\begin{aligned} d_{BTree}(empty(A)) &= empty(B) \\ d_{BTree}(mk(A, a, t_1, t_2)) &= mk(B, c(a), d_{BTree}(t_1), d_{BTree}(t_2)) \end{aligned}$$

4. *Subtyping rules for Either types:*

$$\frac{\Gamma \vdash A <_{c_1} A' : Type \quad \Gamma \vdash B = B' : Type}{\Gamma \vdash Either(A, B) <_{d_{Either1}} Either(A', B') : Type}$$

$$\frac{\Gamma \vdash A = A' : Type \quad \Gamma \vdash B <_{c_2} B' : Type}{\Gamma \vdash Either(A, B) <_{d_{Either2}} Either(A', B') : Type}$$

$$\frac{\Gamma \vdash A <_{c_1} A' : Type \quad \Gamma \vdash B <_{c_2} B' : Type}{\Gamma \vdash Either(A, B) <_{d_{Either3}} Either(A', B') : Type}$$

where

$$\begin{aligned} d_{Either3} =_{df} \; & \mathcal{E}_{Either}(A, B, [z : Either(A, B)]Either(A', B'), \\ & [a : A]left(A', B', c_1(a)), [b : B]right(A', B', c_2(b))) \end{aligned}$$

such that

$$\begin{aligned} d_{Either3}(left(A, B, a)) &= left(A', B', c_1(a)) \\ d_{Either3}(right(A, B, b)) &= right(A', B', c_2(b)) \end{aligned}$$

The definitions of $d_{Either1}$ *and* $d_{Either2}$ *are similar to* $d_{Either3}$.

5. *Subtyping rules for function types:*

$$\frac{\Gamma \vdash A' <_{c_1} A : Type \quad \Gamma \vdash B = B' : Type}{\Gamma \vdash A \to B <_{d_{(\to)1}} A' \to B' : Type}$$

$$\frac{\Gamma \vdash A = A' : Type \quad \Gamma \vdash B <_{c_2} B' : Type}{\Gamma \vdash A \to B <_{d_{(\to)2}} A' \to B' : Type}$$

$$\frac{\Gamma \vdash A' <_{c_1} A : Type \quad \Gamma \vdash B <_{c_2} B' : Type}{\Gamma \vdash A \to B <_{d_{(\to)3}} A' \to B' : Type}$$

where

$$\begin{aligned} d_{(\to)3} \;=_{df}\; & \mathcal{E}_{(\to)}(A, B, [z : A \to B](A' \to B'), \\ & [g : (A)B]lam(A', B', c_2 \circ g \circ c_1)) \end{aligned}$$

such that

$$d_{(\to)3}(lam(A, B, g)) = lam(A', B', c_2 \circ g \circ c_1)$$

The definitions of $d_{(\to)1}$ *and* $d_{(\to)2}$ *are similar to* $d_{(\to)3}$. □

From these examples, we can see that some of the parameters are covariant while some are contravariant. Formal definitions of covariance and contravariance are given as follows.

Definition 5.4.2 ***(Covariance and Contravariance)*** *Let A be a type, K a WT small kind, Φ a WT strictly positive operator, Θ a WT-schema and $\overline{\Theta}$ a finite sequence of inductive WT-schemata.*

- $K^c(A)$ *and* $K^{ct}(A)$ *are to verify whether A is covariant or contravariant in K. $K^c(A) = True$ means that A is covariant in K, and $K^{ct}(A) = True$ means that A is contravariant in K.*
  1. *If* $K \equiv El(B)$ *and*
     (a) *if* $A \not\equiv B$ *then*
     $$K^c(A) = True \qquad K^{ct}(A) = True$$
     (b) *if* $A \equiv B$ *then*
     $$K^c(A) = True \qquad K^{ct}(A) = False$$
  2. *If* $K \equiv (x : K_1)K_2$ *then*
     $$\begin{aligned} K^c(A) &= K_1^{ct}(A) \wedge K_2^c(A) \\ K^{ct}(A) &= K_1^c(A) \wedge K_2^{ct}(A) \end{aligned}$$

where $\wedge$ is the common logical operator 'and'.

- $\Phi^c(A)$ *and* $\Phi^{ct}(A)$ *are to verify whether* A *is covariant or contravariant in* Φ. $\Phi^c(A) = True$ *means that* A *is covariant in* Φ*, and* $\Phi^{ct}(A) = True$ *means that* A *is contravariant in* Φ.

  1. *If* $\Phi \equiv X$ *then*

  $$\Phi^c(A) = True \qquad \Phi^{ct}(A) = True$$

  2. *If* $\Phi \equiv (x : K)\Phi_0$ *then*

  $$\begin{aligned} \Phi^c(A) &= K^c(A) \wedge \Phi_0^c(A) \\ \Phi^{ct}(A) &= K^{ct}(A) \wedge \Phi_0^{ct}(A) \end{aligned}$$

- $\Theta^c(A)$ *and* $\Theta^{ct}(A)$ *are to verify whether* A *is covariant or contravariant in* Θ. $\Theta^c(A) = True$ *means that* A *is covariant in* Θ*, and* $\Theta^{ct}(A) = True$ *means that* A *is contravariant in* Θ.

  1. *If* $\Theta \equiv X$ *then*

  $$\Theta^c(A) = True \qquad \Theta^{ct}(A) = True$$

  2. *If* $\Theta \equiv (x : K)\Theta_0$ *then*

  $$\begin{aligned} \Theta^c(A) &= K^c(A) \wedge \Theta_0^c(A) \\ \Theta^{ct}(A) &= K^{ct}(A) \wedge \Theta_0^{ct}(A) \end{aligned}$$

  3. *If* $\Theta \equiv (\Phi)\Theta_0$ *then*

  $$\begin{aligned} \Theta^c(A) &= \Phi^{ct}(A) \wedge \Theta_0^c(A) \\ \Theta^{ct}(A) &= \Phi^c(A) \wedge \Theta_0^{ct}(A) \end{aligned}$$

- $\overline{\Theta}^c(A)$ *and* $\overline{\Theta}^{ct}(A)$ *are to verify whether Type* A *is covariant or contravariant in* $\overline{\Theta}$. $\overline{\Theta}^c(A) = True$ *means that* A *is covariant in* $\overline{\Theta}$*, and* $\overline{\Theta}^{ct}(A) = True$ *means that* A *is contravariant in* $\overline{\Theta}$.

  $$\begin{aligned} \overline{\Theta}^c(A) &= \Theta_1^c(A) \wedge ... \wedge \Theta_m^c(A) \\ \overline{\Theta}^{ct}(A) &= \Theta_1^{ct}(A) \wedge ... \wedge \Theta_m^{ct}(A) \end{aligned}$$

We say a type A *in* $\overline{\Theta}$ *is* **covariant** *if* $\overline{\Theta}^c(A) = True$ *and* $\overline{\Theta}^{ct}(A) = False$*; and we say a type* A *in* $\overline{\Theta}$ *is* **contravariant** *if* $\overline{\Theta}^{ct}(A) = True$ *and* $\overline{\Theta}^c(A) = False$. □

Now, we give a general definition of subtyping rules and its associated coercions. The general form of subtyping rules for $\mathcal{T}$ is

$$(WTRuleForm) \qquad \frac{premises}{\Gamma \vdash \mathcal{T}(\overline{A}) <_{d_{\mathcal{T}}} \mathcal{T}(\overline{B}) : Type}$$

where $\overline{A} = A_1, ..., A_n$ and $\overline{B} = B_1, ..., B_n$ are fresh and distinct schematic letters. Intuitively, we associate $\mathcal{T}$ with subtyping rules whose conclusion is of the form $\Gamma \vdash \mathcal{T}(\overline{A}) <_{d_{\mathcal{T}}} \mathcal{T}(\overline{B}) : Type$. The coercion $d_{\mathcal{T}}$ is defined by induction on $\mathcal{T}(\overline{A})$ and maps the canonical objects of $\mathcal{T}(\overline{A})$ to the corresponding canonical objects of $\mathcal{T}(\overline{B})$. For example, $d_{List} = map_{List}(A, B, c)$ in the subtyping rule for lists.

In order to find out the premises, we first give a notational definition, *premise set*, as follows.

Notation 5.4.3 *We shall often write $D[\overline{A}]$ for $[A_1/Y_1, ..., A_n/Y_n]D$.*

Definition 5.4.4 ***(premise set)***

- *For any small kind K in Γ, we define $prem_\Gamma(K)$ as follows:*
  1. $K \equiv El(D)$
     (a) *if $\overline{Y} \notin FV(D)$ then $prem_\Gamma(K) = \emptyset$*
     (b) *if $\overline{Y} \in FV(D)$ then $prem_\Gamma(K) = \{(\Gamma, D[\overline{A}], D[\overline{B}])\}$*
  2. $K \equiv (y : K_1)K_2$
     (a) *if $y \notin FV(K_2)$ then $prem_\Gamma(K) = \overline{prem_\Gamma(K_1)} \cup prem_\Gamma(K_2)$, where*

     $$\overline{prem_\Gamma(K_1)} =_{df} \{(\Gamma, B, A) \mid (\Gamma, A, B) \in prem_\Gamma(K_1)\}$$

     (b) *if $y \in FV(K_2)$ then $prem_\Gamma(K) = prem_{\Gamma, y:K_1}(K_2)$. Note that in this case, if K is in a WT-schema, $\overline{Y} \notin FV(K_1)$.*
- *For any WT-schema Θ in Γ, we define $prem_\Gamma(\Theta)$ as follows:*
  1. *$\Theta \equiv X$, then $prem_\Gamma(\Theta) = \emptyset$*
  2. $\Theta \equiv (x : K)\Theta_0$
     (a) *if $x \notin FV(\Theta_0)$ then $prem_\Gamma(\Theta) = prem_\Gamma(K) \cup prem_\Gamma(\Theta_0)$*
     (b) *if $x \in FV(\Theta_0)$ then $prem_\Gamma(\Theta) = prem_{\Gamma, x:K}(\Theta_0)$. Note that in this case, since Θ is in a WT-schema, $\overline{Y} \notin FV(K)$.*
  3. *$\Theta \equiv (x : \Phi)\Theta_0$, then $prem_\Gamma(\Theta) = \overline{prem_\Gamma(\Phi)} \cup prem_\Gamma(\Theta_0)$*

- *For any sequence of WT-schemata in* Γ, $\overline{\Theta} \equiv <\Theta_1, ..., \Theta_m>$, *we define*

$$prem_\Gamma(\overline{\Theta}) = \cup_{i=1}^{m} prem_\Gamma(\Theta_i)$$

Now, suppose there are v elements $prem_\Gamma(\overline{\Theta})$ and we give an order to the elements:

$$(\Gamma_1, C_1, D_1), ..., (\Gamma_v, C_v, D_v)$$

Then the sequence of the premises in the form $WTRuleForm$ is:

$$\Gamma_1 \vdash C_1 \leq_{c_1} D_1 : Type, ..., \Gamma_v \vdash C_v \leq_{c_v} D_v : Type$$

where $c_1, ..., c_v$ are fresh and distinct schematic letters.

Having defined the general forms of the premises, we now define a general form of the coercion $d_{\mathcal{T}}$. We first introduce the following notational definitions.

Definition 5.4.5 *For small kinds* K_1 *and* K_2, $Func[K_1, K_2]$ *is defined as follows.*

- $K_1 \equiv El(C)$ *and* $K_2 \equiv El(D)$
  1. *If* $\Gamma \vdash C \leq_c D : Type$ *is in the sequence of premises, then* $Func[K_1, K_2] = c$.
  2. *If* $C \equiv D$, *then* $Func[K_1, K_2] = id_C = [x : K_1]x$.
  3. *Otherwise,* $Func[K_1, K_2]$ *is undefined.*
- $K_1 \equiv (y : K_{11})K_{12}$ *and* $K_2 \equiv (y : K_{21})K_{22}$. *If both* $Func[K_{21}, K_{11}]$ *and* $Func[K_{12}, K_{22}]$ *are defined and let*

$$\begin{aligned} k_1 &= Func[K_{21}, K_{11}] \\ k_2 &= Func[K_{12}, K_{22}] \end{aligned}$$

then

$$Func[K_1, K_2] = [g : K_1][y : K_{21}]k_2(g(k_1(y)))$$

- *Otherwise,* $Func[K_1, K_2]$ *is undefined.*

Remark 5.4.6 *In general, when c in the form* $\Gamma \vdash C \leq_c D : Type$ *is of kind* $(C)D$, $Func[K_1, K_2]$ *is of kind* $(K_1)K_2$ *if it is definable.*

Notation 5.4.7 *Let* $Y_1, ..., Y_n$ *be the parameters and* Ψ *either a WT strictly positive operator or a WT-schema. We shall write* $\Psi[\overline{A}]$ *for* $[A_1/Y_1, ..., A_n/Y_n]\Psi$, $\Psi[\overline{B}]$ *for* $[B_1/Y_1, ..., B_n/Y_n]\Psi$, *and* $\Psi[\overline{B}][\mathcal{T}(\overline{B})]$ *for* $[B_1/Y_1, ..., B_n/Y_n, \mathcal{T}(\overline{B})/X]\Psi$.

Definition 5.4.8 *Let Φ be WT strictly positive operator, Θ a WT-schema.*

- *for any $f : \Phi[\overline{A}][\mathcal{T}(\overline{B})]$, define $\Phi^k(f)$ of kind $\Phi[\overline{B}][\mathcal{T}(\overline{B})]$ as follows:*
  1. *if $\Phi \equiv X'$ then $\Phi^k(f) = f$*
  2. *if $\Phi \equiv (x : K)\Phi_0$ then*

$$\Phi^k(f) = [x : K[\overline{B}]]\Phi_0^k(f(Func[K[\overline{B}], K[\overline{A}]](x)))$$

- *for any $g : \Theta[\overline{B}][\mathcal{T}(\overline{B})]$, define $\Theta^\lambda(g)$ as follows :*
  1. *if $\Theta \equiv X$ then $\Theta^\lambda(g) = g$*
  2. *if $\Theta \equiv (x : K)\Theta_0$ then*

$$\Theta^\lambda(g) = [x : K[\overline{A}]]\Theta_0^\lambda(g(Func[K[\overline{A}], K[\overline{B}]](x)))$$

  3. *if $\Theta \equiv (x : \Phi)\Theta_0$ then*

$$\begin{aligned}\Theta^\lambda(g) \quad = \quad & [x : \Phi[\overline{A}][\mathcal{T}(\overline{A})]][x' : \Phi[\overline{A}][\mathcal{T}(\overline{B})]] \\ & \Theta_0^\lambda(g(\Phi^k[x']))\end{aligned}$$

□

Then, using the above notational definitions, we define the coercion $d_\mathcal{T}$ in the form $WTRuleForm$.

$$d_\mathcal{T} =_{df} \mathcal{E}_\mathcal{T}(\overline{A}, C, \Theta_1^\lambda(l_1(\overline{B})), ..., \Theta_m^\lambda(l_m(\overline{B})))$$

where $C \equiv [z : \mathcal{T}(\overline{A})]\mathcal{T}(\overline{B})$ and, l_j $(j = 1, ..., m)$ and $\mathcal{E}_\mathcal{T}$ are the introduction operators and the elimination operator of $\mathcal{T}$, respectively (see Section 2.3 for details).

Now, we are ready to specify the subtyping rules from the form $WTRuleForm$. Let the sequence of the premises be:

$$\Gamma_1 \vdash C_1 \leq_{c_1} D_1 : Type, ..., \Gamma_v \vdash C_v \leq_{c_v} D_v : Type$$

Then we will generate $2^v - 1$ subtyping rules for the parameterised inductive data types, each of which has v premises. The premises for each rule are obtained by changing $\leq_{c_i}$ into either $=$ or $<_{c_i}$. Different combinations give different sequences of premises, and hence different rules, except that there must be at least one premise that has the form $\Gamma \vdash C <_c D : Type$. For example, without losing generality, if $\leq_{c_i}$ in the first r premises are changed into $=$, and the left into $<_{c_i}$, then a

subtyping rule will be

$$\frac{\Gamma_1 \vdash C_1 = D_1 : Type, ..., \Gamma_r \vdash C_r = D_r : Type \quad \Gamma_{r+1} \vdash C_{r+1} \leq_{c_{r+1}} D_{r+1} : Type, ..., \Gamma_v \vdash C_v \leq_{c_v} D_v : Type}{\Gamma \vdash \mathcal{T}(\overline{A}) <_e \mathcal{T}(\overline{B}) : Type}$$

where $e \equiv [id_{C_1}/c_1, ..., id_{C_r}/c_r]d_{\mathcal{T}}$.

Remark 5.4.9 *Some types in WT-schemata are neither covariant nor contravariant. This causes that some rules may have contradictory premises. For example, for the inductive type $\mathcal{T}(Y) =_{df} \mathcal{M}[((Y)Y)X]$ parameterised by type variable Y, let $\Theta = ((Y)Y)X$, we have $\Theta^c(Y) = False$ and $\Theta^{ct}(Y) = False$. One of the subtyping rules is*

$$\frac{\Gamma \vdash A <_{c_1} B : Type \quad \Gamma \vdash B <_{c_2} A : Type}{\Gamma \vdash \mathcal{T}(A) <_{d_{\mathcal{T}}} \mathcal{T}(B) : Type}$$

Since the premises in such rules are contradictory (and never satisfied), they can never be applied. So, we assume that all the types that contain parameters in WT-schemata used later are either covariant or contravariant.

Justification of the coercion $d_{\mathcal{T}}$

The coercion $d_{\mathcal{T}}$ as defined in the form *WTRuleForm* sends the canonical objects of $\mathcal{T}(\overline{A})$ to the corresponding canonical objects in $\mathcal{T}(\overline{B})$. For example, the coercion d_{List} in the subtyping rule for *List* satisfies that

$$\begin{aligned} d_{List}(nil(A)) &= nil(B) \\ d_{List}(cons(A, a, l)) &= cons(B, c(a), d_{List}(l)) \end{aligned}$$

In the following lemma, we prove this is in general the case. We first give a definition of $\Theta^u(\overline{A}, \overline{B})$.

Definition 5.4.10 *Let Θ be a WT-schema and assume that Θ be of the form $(x_1 : M_1)...(x_t : M_t)X$ and $x_1, ..., x_t$ are fresh variables. $\Theta^u(\overline{A}, \overline{B})$ is a sequence of arguments:*

1. if $\Theta \equiv X$ then $\Theta^u(\overline{A}, \overline{B}) =<>$
2. if $\Theta \equiv (x_r : K)\Theta_0$ $(r = 1, ..., t)$, then

$$\Theta^u(\overline{A}, \overline{B}) =< Func[K[\overline{A}], K[\overline{B}]](x_r), \Theta_0^u(\overline{A}, \overline{B}) >$$

3. if $\Theta \equiv (x_r : \Phi)\Theta_0$ $(r = 1, ..., t)$ then

$$\Theta^u(\overline{A}, \overline{B}) = < \Phi^k[(\Phi[\overline{A}])^{\natural}[d_{\mathcal{T}}, x_r]], \Theta_0^u(\overline{A}, \overline{B}) >$$

Lemma 5.4.11 *$d_{\mathcal{T}}(l_j(\overline{A}, \Theta_j^v)) = l_j(\overline{B}, \Theta_j^u(\overline{A}, \overline{B}))$, where Θ_j^v as defined in Section 2.3.*

Proof. By the definition of $d_{\mathcal{T}}$ and the computation rules for $\mathcal{T}$, we have

$$\begin{aligned} & d_{\mathcal{T}}(l_j(\overline{A}, \Theta_j^v)) \\ = \ & \mathcal{E}_{\mathcal{T}}(\overline{A}, C, \Theta_1^\lambda(l_1(\overline{B})), ..., \Theta_m^\lambda(l_m(\overline{B})), l_j(\overline{A}, \Theta_j^v)) \\ = \ & \Theta_j^\lambda(l_j(\overline{B}))((\Theta_j[\overline{A}])^\sharp) \end{aligned}$$

We need to prove only that

$$\Theta_j^\lambda(l_j(\overline{B}))((\Theta_j[\overline{A}])^\sharp) = l_j(\overline{B}, \Theta_j^u(\overline{A}, \overline{B}))$$

Note that $l_j(\overline{B}) : \Theta_j[\overline{B}][\mathcal{T}(\overline{B})]$ where

$$\Theta_j[\overline{B}][\mathcal{T}(\overline{B})] =_{df} [B_1/Y_1, ..., B_n/Y_n, \mathcal{T}(\overline{B})/X]\Theta_j$$

Now, we generalise the problem; prove that for any WT-schema Θ and $g : \Theta[\overline{B}][\mathcal{T}(\overline{B})]$, we have

$$\Theta^\lambda(g)((\Theta[\overline{A}])^\sharp) = g(\Theta^u(\overline{A}, \overline{B}))$$

Assume that Θ be of the form $(x_1 : M_1)...(x_t : M_t)X$ and $x_1, ..., x_t$ are fresh variables. Do induction on the structures of WT-schema.

1. If $\Theta \equiv X$ then by the definition of Θ^λ, $\Theta^\sharp$ and Θ^u we have

$$\begin{aligned} \Theta^\lambda(g) &= g \\ (\Theta[\overline{A}])^\sharp &= <> \\ \Theta^u(\overline{A}, \overline{B}) &= <> \end{aligned}$$

   Obviously, $\Theta^\lambda(g)((\Theta[\overline{A}])^\sharp) = g(\Theta^u(\overline{A}, \overline{B})) = g$.

2. If $\Theta \equiv (x_r : K)\Theta_0$ $(r = 1, ..., t)$, then

$$\begin{aligned} \Theta^\lambda(g) &= [x_r : K[\overline{A}]]\Theta_0^\lambda(g(\kappa(x_r))) \\ (\Theta[\overline{A}])^\sharp &= < x_r, (\Theta_0[\overline{A}])^\sharp > \\ \Theta^u(\overline{A}, \overline{B}) &= < \kappa(x_r), \Theta_0^u(\overline{A}, \overline{B}) > \end{aligned}$$

where $\kappa \equiv Func[K[\overline{A}], K[\overline{B}]]$
So, $\Theta^\lambda(g)((\Theta[\overline{A}])^\sharp) = \Theta_0^\lambda(g(\kappa(x_r)))((\Theta_0[\overline{A}])^\sharp)$.
Since $g(\kappa(x_r)) : [\kappa(x_r)/x_r]\Theta_0[\overline{B}][\mathcal{T}(\overline{B})]$, by the induction hypothesis, we have $\Theta_0^\lambda(g(\kappa(x_r)))((\Theta_0[\overline{A}])^\sharp) = g(\kappa(x_r), \Theta_0^u(\overline{A}, \overline{B}))$.
Therefore, $\Theta^\lambda(g)((\Theta[\overline{A}])^\sharp) = g(\Theta^u(\overline{A}, \overline{B}))$.

3. if $\Theta \equiv (x_r : \Phi)\Theta_0$ $(r = 1, ..., t)$, then

$$\begin{aligned} \Theta^\lambda(g) &= [x_r : \Phi[\overline{A}][\mathcal{T}(\overline{A})]][x' : \Phi[\overline{A}][\mathcal{T}(\overline{B})]] \\ &\quad \Theta_0^\lambda(g(\Phi^k(x'))) \\ (\Theta[\overline{A}])^\sharp &= < x_r, (\Phi[\overline{A}])^\natural[d_\mathcal{T}, x_r], (\Theta_0[\overline{A}])^\sharp > \\ \Theta^u(\overline{A}, \overline{B}) &= < \Phi^k((\Phi[\overline{A}])^\natural[d_\mathcal{T}, x_r]), \Theta_0^u(\overline{A}, \overline{B}) > \end{aligned}$$

So,

$$\Theta^\lambda(g)((\Theta[\overline{A}])^\sharp) = \Theta_0^\lambda(g(\Phi^k((\Phi[\overline{A}])^\natural[d_\mathcal{T}, x_r])))((\Theta_0[\overline{A}])^\sharp)$$

Let $t = \Phi^k((\Phi[\overline{A}])^\natural[d_\mathcal{T}, x_r])$. Then $g(t) : [t/x_r]\Theta_0[\overline{B}][\mathcal{T}(\overline{B})]$, by the induction hypothesis, we have

$$\begin{aligned} &\Theta_0^\lambda(g(\Phi^k[\overline{A}, \overline{B}, (\Phi[\overline{A}])^\natural[d_\mathcal{T}, x]]))((\Theta_0[\overline{A}])^\sharp) \\ &= g(\Phi^k[\overline{A}, \overline{B}, (\Phi[\overline{A}])^\natural[d_\mathcal{T}, x]], \Theta_0^u(\overline{A}, \overline{B})) \end{aligned}$$

Therefore, $\Theta^\lambda(g)((\Theta[\overline{A}])^\sharp) = g(\Theta^u(\overline{A}, \overline{B}))$.

□

Now, let's consider another property of the definition of the coercion $d_\mathcal{T}$ in the rule *WTRuleForm*. It satisfies the extensional equality requirement, for example, $map_{List}(A, C, g)$ and $map_{List}(B, C, e) \circ map_{List}(A, B, c)$ are extensionally equal if g and $e \circ c$ are extensionally equal. The following lemma will show that this is in general the case.

Lemma 5.4.12 *By the form WTRuleForm, suppose we have the following:*

$$\frac{premises_1}{\Gamma \vdash \mathcal{T}(A_1, ..., A_n) <_{d_1} \mathcal{T}(B_1, ..., B_n) : Type}$$

$$\frac{premises_2}{\Gamma \vdash \mathcal{T}(B_1, ..., B_n) <_{d_2} \mathcal{T}(C_1, ..., C_n) : Type}$$

$$\frac{premises_3}{\Gamma \vdash \mathcal{T}(A_1, ..., A_n) <_{d_3} \mathcal{T}(C_1, ..., C_n) : Type}$$

where d_1, d_2 and d_3 are defined according to the definition of $d_{\mathcal{T}}$ in the form WTRuleForm with respect to their premises respectively, and the premises are described as follows:

1. *If the i-th premise in the form WTRuleForm is covariant,* i.e. *it is obtained from a covariant type D by substituting parameters, then* $\Gamma_i \vdash D[\overline{A}] \leq_{c_i} D[\overline{B}] : Type$ *is in* $premises_1$, $\Gamma_i \vdash D[\overline{B}] \leq_{e_i} D[\overline{C}] : Type$ *is in* $premises_2$ *and* $\Gamma_i \vdash D[\overline{A}] \leq_{g_i} D[\overline{C}] : Type$ *is in* $premises_3$ *for some* c_i, e_i *and* g_i*; and* g_i *and* $e_i \circ c_i$ *are extensionally equal.*
2. *If the i-th premise in the form WTRuleForm is contravariant,* i.e. *it is obtained from a contravariant type D by substituting parameters, then* $\Gamma_i \vdash D[\overline{B}] \leq_{c_i} D[\overline{A}] : Type$ *is in* $premises_1$, $\Gamma_i \vdash D[\overline{C}] \leq_{e_i} D[\overline{B}] : Type$ *is in* $premises_2$ *and* $\Gamma_i \vdash D[\overline{C}] \leq_{g_i} D[\overline{A}] : Type$ *is in* $premises_3$ *for some* c_i, e_i *and* g_i*; and* g_i *and* $c_i \circ e_i$ *are extensionally equal.*

Then, d_3 *and* $d_2 \circ d_1$ *are extensionally equal.*

Proof. First, by induction on constructors of type $\mathcal{T}(A_1, ..., A_n)$.

For any canonical object $l_j(\overline{A}, \Theta_j^v)$ of type $\mathcal{T}(\overline{A})$, by Lemma 5.4.11, we have

$$d_1(l_j(\overline{A}, \Theta_j^v)) = l_j(\overline{B}, \Theta_j^u(\overline{A}, \overline{B}))$$

$$\begin{aligned} d_2(d_1(l_j(\overline{A}, \Theta_j^v))) &= d_2(l_j(\overline{B}, \Theta_j^u(\overline{A}, \overline{B}))) \\ &= l_j(\overline{C}, [\Theta_j^u(\overline{A}, \overline{B})/\Theta_j^v]\Theta_j^u(\overline{B}, \overline{C})) \end{aligned}$$

and

$$d_3(l_j(\overline{A}, \Theta_j^v)) = l_j(\overline{C}, \Theta_j^u(\overline{A}, \overline{C}))$$

Now, we need to prove that $l_j(\overline{C}, [\Theta_j^u(\overline{A}, \overline{B})/\Theta_j^v]\Theta_j^u(\overline{B}, \overline{C}))$ and $l_j(\overline{C}, \Theta_j^u(\overline{A}, \overline{C}))$ are extensionally equal. To prove this, we prove that for any WT-schema Θ, every element in $[\Theta^u(\overline{A}, \overline{B})/\Theta^v]\Theta^u(\overline{B}, \overline{C})$ is extensionally equal to the corresponding element in $\Theta^u(\overline{A}, \overline{C})$.

Assume that Θ be of the form $(x_1 : M_1)...(x_t : M_t)X$ and $x_1, ..., x_t$ are fresh variables. Do induction on the structures of WT-schema.

1. If $\Theta \equiv X$ then $\Theta^u(\overline{A}, \overline{B}) = \Theta^u(\overline{B}, \overline{C}) = \Theta^u(\overline{A}, \overline{C}) = < >$. Obviously, every element in $[\Theta^u(\overline{A}, \overline{B})/\Theta^v]\Theta^u(\overline{B}, \overline{C})$ is extensionally equal to the corresponding element in $\Theta^u(\overline{A}, \overline{C})$ because they are empty sequences.

2. If $\Theta \equiv (x_r : K)\Theta_0$ $(r = 1, ..., t)$ and let $\kappa_1 = Func[K[\overline{A}], K[\overline{B}]]$, $\kappa_2 = Func[K[\overline{B}], K[\overline{C}]]$ and $\kappa_3 = Func[K[\overline{A}], K[\overline{C}]]$, then

$$\begin{aligned}
\Theta^v &= < x_r, \Theta_0^v > \\
\Theta^u(\overline{A}, \overline{B}) &= < \kappa_1(x_r), \Theta_0^u(\overline{A}, \overline{B}) > \\
\Theta^u(\overline{B}, \overline{C}) &= < \kappa_2(x_r), \Theta_0^u(\overline{B}, \overline{C}) > \\
\Theta^u(\overline{A}, \overline{C}) &= < \kappa_3(x_r), \Theta_0^u(\overline{A}, \overline{C}) >
\end{aligned}$$

Therefore,

$$[\Theta^u(\overline{A}, \overline{B})/\Theta^v]\Theta^u(\overline{B}, \overline{C}) = \\ < \kappa_2(\kappa_1(x_r)), [\Theta_0^u(\overline{A}, \overline{B})/\Theta_0^v]\Theta_0^u(\overline{A}, \overline{B}) >$$

By the induction hypothesis of the structures of WT-schema, every element in $[\Theta_0^u(\overline{A}, \overline{B})/\Theta_0^v]\Theta_0^u(\overline{B}, \overline{C})$ is extensionally equal to the corresponding element in $\Theta_0^u(\overline{A}, \overline{C})$. If we can prove that κ_3 is extensionally equal to $\kappa_2 \circ \kappa_1$, then we know that every element in $[\Theta^u(\overline{A}, \overline{B})/\Theta^v]\Theta^u(\overline{B}, \overline{C})$ is extensionally equal to the corresponding element in $\Theta^u(\overline{A}, \overline{C})$.

Now, let's prove that κ_3 is extensionally equal to $\kappa_2 \circ \kappa_1$ by induction on structures of WT small kind K.

(a) If $K \equiv El(M)$ and

 (i) if M doesn't contain any parameter *i.e.* $\overline{Y} \notin FV(M)$ then $K[\overline{A}] \equiv K[\overline{B}] \equiv K[\overline{C}] \equiv K$ and hence $\kappa_1 = \kappa_2 = \kappa_3 = id_K = [x : K]x$. So, κ_3 is extensionally equal to $\kappa_2 \circ \kappa_1$.

 (ii) M contains any parameters *i.e.* $\overline{Y} \in FV(M)$.
 If $\Gamma_i \vdash M[\overline{A}] \leq_{c_i} M[\overline{B}] : Type$ is in $premises_1$,
 $\Gamma_i \vdash M[\overline{B}] \leq_{e_i} M[\overline{C}] : Type$ is in $premises_2$ and
 $\Gamma_i \vdash M[\overline{A}] \leq_{g_i} M[\overline{C}] : Type$ is in $premises_3$ then
 $\kappa_1 = c_i$, $\kappa_2 = e_i$ and $\kappa_3 = g_i$ and by the assumption, κ_3 is extensionally equal to $\kappa_2 \circ \kappa_1$.
 If $\Gamma_i \vdash M[\overline{B}] \leq_{c_i} M[\overline{A}] : Type$ is in $premises_1$,
 $\Gamma_i \vdash M[\overline{C}] \leq_{e_i} M[\overline{B}] : Type$ is in $premises_2$ and
 $\Gamma_i \vdash M[\overline{C}] \leq_{g_i} M[\overline{A}] : Type$ is in $premises_3$ then
 $\kappa_1 = e_i$, $\kappa_2 = c_i$ and $\kappa_3 = g_i$ and by the assumption, κ_3 is extensionally equal to $\kappa_2 \circ \kappa_1$.

(b) If $K \equiv (y : K_1)K_2$ then let $\kappa_{11} = Func[K_1[\overline{B}], K_1[\overline{A}]]$, $\kappa_{12} = Func[K_1[\overline{C}], K_1[\overline{B}]]$, $\kappa_{13} = Func[K_1[\overline{C}], K_1[\overline{A}]]$, and $\kappa_{21} = Func[K_2[\overline{A}], K_2[\overline{B}]]$, $\kappa_{22} = Func[K_2[\overline{B}], K_2[\overline{C}]]$ and $\kappa_{23} = Func[K_2[\overline{A}], K_2[\overline{C}]]$ and we have

$$\begin{aligned}
\kappa_1 &= [f : K[\overline{A}]][y : K_1[\overline{B}]]\kappa_{21}(f(\kappa_{11}(y))) \\
\kappa_2 &= [g : K[\overline{B}]][x : K_1[\overline{C}]]\kappa_{22}(g(\kappa_{12}(x))) \\
\kappa_3 &= [f : K[\overline{A}]][x : K_1[\overline{C}]]\kappa_{23}(f(\kappa_{13}(x)))
\end{aligned}$$

Therefore, we have

$$\kappa_2 \circ \kappa_1 = [f : K[\overline{A}]][x : K_1[\overline{C}]]\kappa_{22}(\kappa_{21}(f(\kappa_{11}(\kappa_{12}(x)))))$$

By the induction hypothesis, we have that κ_{23} is extensionally equal to $\kappa_{22} \circ \kappa_{21}$ and κ_{13} is extensionally equal to $\kappa_{11} \circ \kappa_{12}$. So, κ_3 is extensionally equal to $\kappa_2 \circ \kappa_1$.

3. If $\Theta \equiv (x_r : \Phi)\Theta_0$ $(r = 1, ..., t)$ then

$$\begin{aligned}
\Theta^v &= < x_r, \Theta_0^v > \\
\Theta^u(\overline{A}, \overline{B}) &= < \Phi^k((\Phi[\overline{A}])^\natural[d_1, x_r]), \Theta_0^u(\overline{A}, \overline{B}) > \\
\Theta^u(\overline{B}, \overline{C}) &= < \Phi^k((\Phi[\overline{B}])^\natural[d_2, x_r]), \Theta_0^u(\overline{B}, \overline{C}) > \\
\Theta^u(\overline{A}, \overline{C}) &= < \Phi^k((\Phi[\overline{A}])^\natural[d_3, x_r]), \Theta_0^u(\overline{A}, \overline{C}) >
\end{aligned}$$

Therefore,

$$\begin{gathered}
[\Theta^u(\overline{A}, \overline{B})/\Theta^v]\Theta^u(\overline{B}, \overline{C}) = \\
< \Phi^k((\Phi[\overline{B}])^\natural[d_2, \Phi^k((\Phi[\overline{A}])^\natural[d_1, x_r])]), \\
[\Theta_0^u(\overline{A}, \overline{B})/\Theta_0^v]\Theta_0^u(\overline{B}, \overline{C}) >
\end{gathered}$$

By the induction hypothesis of the structures of WT-schema, we have that every element in $[\Theta_0^u(\overline{A}, \overline{B})/\Theta_0^v]\Theta_0^u(\overline{B}, \overline{C})$ is extensionally equal to the corresponding element in $\Theta_0^u(\overline{A}, \overline{C})$. If we can prove that

$$\Phi^k((\Phi[\overline{A}])^\natural[d_3, x_r])$$

and

$$\Phi^k((\Phi[\overline{B}])^\natural[d_2, \Phi^k((\Phi[\overline{A}])^\natural[d_1, x_r])])$$

are extensionally equal, then we know that every element in $[\Theta^u(\overline{A}, \overline{B})/\Theta^v]\Theta^u(\overline{B}, \overline{C})$ is extensionally equal to the corresponding element in $\Theta^u(\overline{A}, \overline{C})$.

Now, let's prove that, for any x and y, if x and y are extensionally equal, then

$$\Phi^k((\Phi[\overline{A}])^\natural[d_3, x])$$

and

$$\Phi^k((\Phi[\overline{B}])^\natural[d_2, \Phi^k((\Phi[\overline{A}])^\natural[d_1, y])])$$

are extensionally equal, by induction on the structures of the WT strictly positive operator.

(a) If $\Phi \equiv X$ then we have

$$\begin{aligned}(\Phi[\overline{A}])^\natural[d_1, y] &= d_1(y)\\ (\Phi[\overline{A}])^\natural[d_3, x] &= d_3(x)\end{aligned}$$

So,

$$\Phi^k((\Phi[\overline{A}])^\natural[d_3, x]) = d_3(x)$$

and

$$\Phi^k((\Phi[\overline{B}])^\natural[d_2, \Phi^k((\Phi[\overline{A}])^\natural[d_1, y])]) \\ = d_2(d_1(y))$$

By the induction hypothesis of constructors of type $\mathcal{T}(A_1, ..., A_n)$, we have that $d_3(x)$ is extensionally equal to $d_2(d_1(y))$.

(b) if $\Phi \equiv (x : K)\Phi_0$ then let $\kappa_1 = Func[K[\overline{B}], K[\overline{A}]]$, $\kappa_2 = Func[K[\overline{C}], K[\overline{B}]]$ and $\kappa_3 = Func[K[\overline{C}], K[\overline{A}]]$, and we have

$$\begin{aligned}(\Phi[\overline{A}])^\natural[d_1, y] &= [a : K[\overline{A}]](\Phi_0[\overline{A}])^\natural[d_1, y(a)]\\ (\Phi[\overline{A}])^\natural[d_3, x] &= [a : K[\overline{A}]](\Phi_0[\overline{A}])^\natural[d_3, x(a)]\end{aligned}$$

So,

$$\Phi^k((\Phi[\overline{A}])^\natural[d_3, x]) \\ = [z : K[\overline{C}]]\Phi_0^k((\Phi_0[\overline{A}])^\natural[d_3, x(\kappa_3(z))])$$

and

$$\begin{aligned}&\Phi^k((\Phi[\overline{B}])^\natural[d_2, \Phi^k((\Phi[\overline{A}])^\natural[d_1, y])])\\ &= \Phi^k((\Phi[\overline{B}])^\natural[d_2, [b : K[\overline{B}]]\Phi_0^k((\Phi_0[\overline{A}])^\natural[d_1, y(\kappa_1(b))])]])\\ &= \Phi^k([b : K[\overline{B}]](\Phi_0[\overline{B}])^\natural[d_2, \Phi_0^k((\Phi_0[\overline{A}])^\natural[d_1, y(\kappa_1(b))])]])\\ &= [z : K[\overline{C}]]\Phi_0^k((\Phi_0[\overline{B}])^\natural[d_2, \Phi_0^k((\Phi_0[\overline{A}])^\natural[d_1, y(\kappa_1(\kappa_2(z)))])])\end{aligned}$$

As proved before, we have that κ_3 and $\kappa_1 \circ \kappa_2$ are extensionally equal. So, $x(\kappa_3(z))$ and $y(\kappa_1(\kappa_2(z)))$ are extensionally equal. Now, by the induction hypothesis of the structures of the WT

strictly positive operator, we have that

$$\Phi_0^k((\Phi_0[\overline{A}])^\natural[d_3, x(\kappa_3(z))])$$

and

$$\Phi_0^k((\Phi_0[\overline{B}])^\natural[d_2, \Phi_0^k((\Phi_0[\overline{A}])^\natural[d_1, y(\kappa_1(\kappa_2(z)))])])$$

are extensionally equal. Therefore, we have that

$$\Phi^k((\Phi[\overline{A}])^\natural[d_3, x])$$

and

$$\Phi^k((\Phi[\overline{B}])^\natural[d_2, \Phi^k((\Phi[\overline{A}])^\natural[d_1, y])])$$

are extensionally equal.

So, for any canonical object $l_j(\overline{A}, \Theta_j^v)$ of type $\mathcal{T}(\overline{A})$, we have $d_3(l_j(\overline{A}, \Theta_j^v))$ and $d_2(d_1(l_j(\overline{A}, \Theta_j^v)))$ are extensionally equal. Therefore, d_3 and $d_2 \circ d_1$ are extensionally equal. □

5.5. Coherence

In this section, we show that the coherence of subtyping rules holds for the inductive types generated by WT-schemata. Some related properties are also proved.

Note that the set $\mathcal{R}$ of subtyping rules consists of the rule $(WDCrule)$ and the subtyping rules for parameterised inductive types generated by *WT-schemata* and, the system $T[\mathcal{R}]_0$ also includes the congruence rule $(Cong)$. Furthermore, we assume that for any judgement $\Gamma \vdash A <_c B : Type \in \mathcal{C}$, neither A nor B is computationally equal to any $\mathcal{T}_j$-type, where $\mathcal{T}_j$ is a type constructor such as $List$, $Either$ and $Maybe$, and $\mathcal{T}_i \not\equiv \mathcal{T}_j$ if $i \neq j$ (for example, $\mathcal{T}_1 \equiv List$, $\mathcal{T}_2 \equiv Either$ and $\mathcal{T}_3 \equiv Maybe$). We also assume that the original type theory T has good properties, in particular the Church-Rosser property and the property of context replacement by equal kinds.

We also denote by $\mathcal{C}_M$ the set of the derivable subtyping judgements of the form $\Gamma \vdash M <_d M' : Type$ in $T[\mathcal{R}]_0$; that is, $\Gamma \vdash M <_d M' : Type \in \mathcal{C}_M$ if and only if $\Gamma \vdash M <_d M' : Type$ is derivable in $T[\mathcal{R}]_0$.

Lemma 5.5.1 *If $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_M$ then both M_1 and M_2 are computationally equal to $\mathcal{T}_j$-type (*i.e. *the normal forms of M_1 and M_2 have same type constructor) or $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}$.*

Proof. By induction on derivations.

If the last rule is one of the $\mathcal{T}_j$-subtyping rules, then we know that both M_1 and M_2 are $\mathcal{T}_j$-type.

Now, suppose that the last rule is the congruence rule, that is

$$\frac{\Gamma \vdash M_1' <_{d'} M_2' : Type \quad \Gamma \vdash M_1 = M_1' : Type \quad \Gamma \vdash M_2 = M_2' : Type \quad \Gamma \vdash d = d' : (M_1)M_2}{\Gamma \vdash M_1 <_d M_2 : Type}$$

By the induction hypothesis, both M_1' and M_2' are computationally equal to $\mathcal{T}_j$-type or $\Gamma \vdash M_1' <_{d'} M_2' : Type \in \mathcal{C}$.

If both M_1' and M_2' are computationally equal to $\mathcal{T}_j$-type, then both M_1 and M_2 are computationally equal to $\mathcal{T}_j$-type.

If $\Gamma \vdash M_1' <_{d'} M_2' : Type \in \mathcal{C}$, then $\Gamma \vdash M_1 <_d M_2 \in \mathcal{C}$ since $\mathcal{C}$ is a WDC. □

Theorem 5.5.2 *If* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_M$ *then* $\Gamma \not\vdash M_1 = M_2 : Type$.

Proof. By induction on derivations. □

Lemma 5.5.3 ***(Context equality)*** *If* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_M$ *and* $\vdash \Gamma = \Gamma'$ *then* $\Gamma' \vdash M_1 <_d M_2 : Type \in \mathcal{C}_M$.

Proof. By induction on derivations. □

Theorem 5.5.4 ***(Weakening)*** *If* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_M$, $\Gamma \subseteq \Gamma'$ *and* Γ' *is valid then* $\Gamma' \vdash M_1 <_d M_2 : Type \in \mathcal{C}_M$.

Proof. By induction on derivations. □

Theorem 5.5.5 ***(Coherence)*** *If* $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}_M$, $\Gamma \vdash M_1' <_{d'} M_2' : Type \in \mathcal{C}_M$, $\Gamma \vdash M_1 = M_1' : Type$ *and* $\Gamma \vdash M_2 = M_2' : Type$ *then* $\Gamma \vdash d = d' : (M_1)M_2$.

Proof. By induction on derivations.

By Lemma 5.5.1, we need to consider only two cases:

- $\Gamma \vdash M_1 <_d M_2 \in \mathcal{C}$.
  Since $\Gamma \vdash M_1 = M_1'$, $\Gamma \vdash M_2 = M_2'$ and $\mathcal{C}$ is a WDC, we have $\Gamma \vdash M_1' <_{d'} M_2' \in \mathcal{C}$. Therefore, $\Gamma \vdash d = d'$ by Lemma 3.4.2.
- Both M_1 and M_2 are computationally equal to $\mathcal{T}_j$-type.
  Since $\Gamma \vdash M_1 = M_1'$ and $\Gamma \vdash M_2 = M_2'$, both M_1' and M_2' are

computationally equal to $\mathcal{T}_j$-type.

The derivations of $\Gamma \vdash M_1 <_d M_2$ and $\Gamma \vdash M_1' <_{d'} M_2'$ must have the following forms:

$$\frac{\dfrac{\dfrac{premises_1}{\Gamma \vdash \mathcal{T}_j(A_1,...,A_n) <_{d_1} \mathcal{T}_j(B_1,...,B_n)}}{...(Congruence\ \ rule)...}}{\Gamma \vdash M_1 <_d M_2}$$

where $\Gamma \vdash \mathcal{T}_j(A_1,...,A_n) = M_1$, $\Gamma \vdash \mathcal{T}_j(B_1,...,B_n) = M_2$ and $\Gamma \vdash d = d_1$; and

$$\frac{\dfrac{\dfrac{premises_2}{\Gamma \vdash \mathcal{T}_j(A_1',...,A_n') <_{d_2} \mathcal{T}_j(B_1',...,B_n')}}{...(Congruence\ \ rule)...}}{\Gamma \vdash M_1' <_{d'} M_2'}$$

where $\Gamma \vdash \mathcal{T}_j(A_1',...,A_n') = M_1'$, $\Gamma \vdash \mathcal{T}_j(B_1',...,B_n') = M_2'$ and $\Gamma \vdash d' = d_2$.

Now we prove that the subtyping rules used to derive

$$\Gamma \vdash \mathcal{T}_j(A_1,...,A_n) <_{d_1} \mathcal{T}_j(B_1,...,B_n)$$

and

$$\Gamma \vdash \mathcal{T}_j(A_1',...,A_n') <_{d_2} \mathcal{T}_j(B_1',...,B_n')$$

must be the same. That is,

- If $\Gamma_i \vdash E_i = F_i$ is in $premises_1$ then $\Gamma_i \vdash E_i' = F_i'$ is in $premises_2$ and,
- If $\Gamma_i \vdash E_i <_{c_i} F_i$ is in $premises_1$then $\Gamma_i \vdash E_i' <_{c_i'} F_i'$ is in $premises_2$ for some E_i', F_i' and c_i'.

Since $\Gamma \vdash M_1 = M_1'$ and $\Gamma \vdash M_2 = M_2'$, we have

$$\Gamma \vdash \mathcal{T}_j(A_1,...,A_n) = \mathcal{T}_j(A_1',...,A_n')$$

and

$$\Gamma \vdash \mathcal{T}_j(B_1, ..., B_n) = \mathcal{T}_j(B'_1, ..., B'_n)$$

Since $T[\mathcal{R}]_0$ is a conservative extension of T and T has the Church-Rosser property, we have $\Gamma \vdash A_i = A'_i$ and $\Gamma \vdash B_i = B'_i$. Since E_i and E'_i are obtained form a type D by substituting parameters, we have $\Gamma_i \vdash E_i = E'_i$ and similarly $\Gamma_i \vdash F_i = F'_i$. By Theorem 5.5.2, if $\Gamma_i \vdash E_i <_{c_i} F_i$ is in $premises_1$, then $\Gamma_i \not\vdash E_i = F_i$ and hence $\Gamma_i \not\vdash E'_i = F'_i$. So, $\Gamma_i \vdash E'_i <_{c'_i} F'_i$ is in $premises_2$ for some c'_i.

Now, by the induction hypothesis, we have $\Gamma \vdash c_i = c'_i$ in cases that $\Gamma_i \vdash E_i <_{c_i} F_i$ is in $premises_1$ and $\Gamma_i \vdash E'_i <_{c'_i} F'_i$ is in $premises_2$.

Therefore, since d_1 and d_2 are given by the same rule, we have $\Gamma \vdash d_1 = d_2$ and hence $\Gamma \vdash d = d'$.

□

5.6. Admissibility of Substitution and Weak Transitivity

In this section, we show that the substitution and weak transitivity rules are admissible for the subtyping rules of the inductive types generated by WT-schemata.

Theorem 5.6.1 ***(Substitution)*** *If* $\Gamma, x : K, \Gamma' \vdash M_1 <_d M_2 : Type \in \mathcal{C}_M$ *and* $\Gamma \vdash k : K$ *then* $\Gamma, [k/x]\Gamma' \vdash [k/x]M_1 <_{[k/x]d} [k/x]M_2 : Type \in \mathcal{C}_M$.

Proof. By induction on derivations. □

Theorem 5.6.2 ***(Weak Transitivity)*** *If* $\Gamma \vdash M_1 <_{d_1} M_2 : Type \in \mathcal{C}_M$, $\Gamma \vdash M'_2 <_{d_2} M_3 : Type \in \mathcal{C}_M$ *and* $\Gamma \vdash M_2 = M'_2 : Type$ *then* $\Gamma \vdash M_1 <_{d_3} M_3 : Type \in \mathcal{C}_M$ *for some* d_3 *and* d_3 *is extensionally equal to* $d_2 \circ d_1$.

Proof. By induction on derivations.

By Lemma 5.5.1, we need to consider only two cases:

- $\Gamma \vdash M_1 <_d M_2 \in \mathcal{C}$.

  Then both M_1 and M_2 are not computationally equal to $\mathcal{T}_j$-type. Since $\Gamma \vdash M_2 = M'_2$, we have $\Gamma \vdash M'_1 <_{d'} M'_2 \in \mathcal{C}$ by Lemma 5.5.1. Therefore, $\Gamma \vdash M_1 <_{d_3} M_3 \in \mathcal{C}$ for some d_3. Therefore, $\Gamma \vdash M_1 <_{d_3} M_3 \in \mathcal{C}$ for some d_3 and d_3 is computationally equal to $d_2 \circ d_1$ by Lemma 3.4.2.

- Both M_1 and M_2 are computationally equal to $\mathcal{T}_i$-type.
  The derivation of $\Gamma \vdash M_1 <_d M_2$ must have the following form:

$$\frac{\frac{\vdots}{premises_1}}{\Gamma \vdash \mathcal{T}_j(A_{11}, ..., A_{1n}) <_{d_1'} \mathcal{T}_j(A_{21}, ..., A_{2n})}$$

$$\vdots$$

$$...(Congruence \quad rule)...$$

$$\frac{\vdots}{\Gamma \vdash M_1 <_{d_1} M_2}$$

where $\Gamma \vdash \mathcal{T}_j(A_{11}, ..., A_{1n}) = M_1$, $\Gamma \vdash \mathcal{T}_j(A_{21}, ..., A_{2n}) = M_2$ and $\Gamma \vdash d_1 = d_1'$.
Since $\Gamma \vdash M_2 = M_2'$ and M_2 is computationally equal to $\mathcal{T}_i$-type, both M_2' and M_3 are computationally equal to $\mathcal{T}_i$-type. Therefore, the derivation of $\Gamma \vdash M_2' <_{d_2} M_3 : Type$ must be of the form:

$$\frac{\frac{\vdots}{premises_2}}{\Gamma \vdash \mathcal{T}_j(A'_{21}, ..., A'_{2n}) <_{d_2'} \mathcal{T}_j(A_{31}, ..., A_{3n})}$$

$$\vdots$$

$$...(Congruence \quad rule)...$$

$$\frac{\vdots}{\Gamma \vdash M_2' <_{d_2} M_3}$$

where $\Gamma \vdash \mathcal{T}_j(A'_{21}, ..., A'_{2n}) = M_2'$, $\Gamma \vdash \mathcal{T}_j(A_{31}, ..., A_{3n}) = M_3$ and $\Gamma \vdash d_1 = d_1'$.
Since $\Gamma \vdash M_2 = M_2'$, we have $\Gamma \vdash \mathcal{T}_j(A_{21}, ..., A_{2n}) = \mathcal{T}_j(A'_{21}, ..., A'_{2n})$. Since $T[\mathcal{R}]_0$ is a conservative extension of T and T has the Church-Rosser property, we have $\Gamma \vdash A_{2i} = A'_{2i}$.
Now, let us consider the i-th premise in $premises_1$ and $premises_2$ and analyse one difficult case that $\Gamma_i \vdash E_i <_{c_i} F_i$ is in $premises_1$ and $\Gamma_i \vdash G_i <_{e_i} H_i$ in $premises_2$.

  1. If the i-th premise in the form $WTRuleForm$ is obtained from a covariant type D, then $F_i \equiv D[\overline{A_2}]$ and $G_i \equiv D[\overline{A_2'}]$. Since $\Gamma \vdash A_{2i} = A'_{2i}$ for every i, we have $\Gamma_i \vdash F_i = G_i$. By inductive

hypothesis, we have $\Gamma_i \vdash E_i <_{g_i} H_i$ for some g_i and, g_i and $e_i \circ c_i$ are extensionally equal.

2. If the i-th premise in the form $WTRuleForm$ is obtained from a contravariant type D, then $E_i \equiv D[\overline{A_2}]$ and $H_i \equiv D[\overline{A'_2}]$. Since $\Gamma \vdash A_{2i} = A'_{2i}$ for every i, we have $\Gamma_i \vdash E_i = H_i$. By induction hypothesis, we have $\Gamma_i \vdash G_i <_{g_i} F_i$ for some g_i and, g_i and $c_i \circ e_i$ are extensionally equal.

By one of the subtyping rules for $\mathcal{T}_j$, we have
$\Gamma \vdash \mathcal{T}_j(A_{11}, ..., A_{1n}) <_{d_3} \mathcal{T}_j(A_{31}, ..., A_{3n})$ and by Lemma 5.4.12, we have d_3 is extensionally equal to $d'_2 \circ d'_1$.

By the congruence rule, we have $\Gamma \vdash M_1 <_{d_3} M_3$ and d_3 is extensionally equal to $d_2 \circ d_1$.

□

Corollary 5.6.3 ***(Extensional equality requirement)***
If $\Gamma \vdash M_1 <_{d_1} M_2 : Type \in \mathcal{C}_M$, $\Gamma \vdash M_2 <_{d_2} M_3 : Type \in \mathcal{C}_M$ and $\Gamma \vdash M_1 <_{d_3} M_3 : Type \in \mathcal{C}_M$ then d_3 and $d_2 \circ d_1$ are extensionally equal.

Proof. By Theorem 5.6.2 and Theorem 5.5.5. □

5.7. Extension of WT-schemata

One may extend WT-schemata so that some families of inductive types can also be covered. For example, the type of vectors is defined as follows:

$$Vec =_{df} [A : Type]\mathcal{M}[X(O), (n : N)(A)(X(n))X(S(n))]$$

where X is a placeholder of kind $(N)Type$, N is the type of natural numbers, O and S are constructors for zero and the successor respectively. A common subtyping rule for vectors is the following:

$$\frac{\Gamma \vdash n : N \quad \Gamma \vdash A <_c B : Type}{\Gamma \vdash Vec(A, n) <_{d(n)} Vec(B, n) : Type}$$

where

$$\begin{aligned} d(O, vnil(A)) &= vnil(B) \\ d(S(m), vcons(A, m, a, l)) &= vcons(B, m, c(a), d(m, l)) \end{aligned}$$

and *vnil* and *vcons* are the constructors of vectors introduced as usual (see Example 2.3.7 for details).

Adding this subtyping rule into $\mathcal{R}$, all the good properties are kept, *i.e.,* $\mathcal{R}$ is still coherent, the substitution rule is admissible, weak transitivity holds and the equality requirement is satisfied.

Now, we give a formal definition of extended WT-schemata.

Definition 5.7.1 ***(Extended WT strictly positive operator and WT-schema)*** *Let Γ be a valid context, $S_1, ..., S_k$ ($k \in \omega$) be kinds in Γ,* i.e. *judgement $\Gamma \vdash S_i$ Kind is derivable ($i = 1, ..., k$), X be a placeholder of kind $(s_1 : S_1)...(s_k : S_k)Type$.*

- *A WT strictly positive operator in Γ, with respect to X and the parameters $Y_1, ..., Y_n$, is of one of the following forms:*
  1. *$\Phi \equiv X(s_1, ..., s_k)$, where $\Gamma \vdash s_i : S_i$ ($i = 1, ..., k$), or*
  2. *$\Phi \equiv (x : K)\Phi_0$, where K is a WT small kind in Γ and Φ_0 is a WT strictly positive operator in $\Gamma, x : K$, and if $x \in FV(\Phi_0)$ then none of the parameters occur free in K* i.e. $\overline{Y} \notin FV(K)$.
- *A WT-schema Θ in Γ, with respect to X and the parameters $Y_1, ..., Y_n$, is of one of the following forms:*
  1. *$\Theta \equiv X(s_1, ..., s_k)$, where $\Gamma \vdash s_i : S_i$ ($i = 1, ..., k$), or*
  2. *$\Theta \equiv (x : K)\Theta_0$, where K is a WT small kind in Γ and Θ_0 is a WT-schema in $\Gamma, x : K$, and if $x \in FV(\Theta_0)$ then none of the parameters occur free in K* i.e. $\overline{Y} \notin FV(K)$.
  3. *$\Theta \equiv (x : \Phi)\Theta_0$, where $x \notin FV(\Theta_0)$, Φ is a WT strictly positive operator in Γ and Θ_0 is an WT-schema in Γ.*

Remark 5.7.2 *As mentioned in Section 5.2.2, WT-schemata avoid coercion dependency between premises such as the subtyping rules for Σ-types to make sure that there is no coercion in one premise that occurs in another premise. The above definition also captures this idea, for example, there is no coercion dependency in the subtyping rule for the type of vectors.*

5.8. Discussion: new computation rules

The normal transitivity rule in coercive subtyping has been proved to be admissible for the subtyping rules of some parameterised inductive types such as Σ-types. The main reason that it can be proved is

because such inductive types have only one constructor and some important function operators such as π_1 and π_2 can be defined, and one can use these operators to define nice coercions. However, for many inductive types that have more than one constructor, the transitivity rule fails to be admissible. Weak transitivity is introduced and it is admissible for a large class of parameterised inductive types such as lists. Weak transitivity holds because it requires only the existence of coercion and extensional equality.

However, there are two problems. One regards subkinding rules. In the coercive subtyping system with strong (or normal) transitivity, the subkinding rules in Figure 3.2.2 are included. However, the coercive subtyping system with weak transitivity includes only the subkinding rules in Figure 3.2.1, and the subkinding rules in Figure 3.2.2 are excluded. One of the reasons is that, for example, in the subkinding rule

$$\frac{\Gamma \vdash K_1' <_{c_1} K_1 \quad \Gamma, x' : K_1' \vdash [c_1(x')/x]K_2 <_{c_2} K_2'}{\Gamma \vdash (x : K_1)K_2 <_{[f:(x:K_1)K_2][x':K_1']c_2(f(c_1(x')))} (x : K_1')K_2'}$$

the coercion c_1 in the first premise occurs in the second premise. Hence the weak transitivity rule (WTK) for subkinding

$$(WTK) \qquad \frac{\Gamma \vdash K <_c K' \quad \Gamma \vdash K' <_{c'} K''}{\Gamma \vdash K <_{c''} K'}$$

fails to be admissible. In fact, one can construct a counter example as we did in Example 5.2.3.

Another problem regards the combination of the subtyping rules for inductive data types. As we showed in Example 5.2.3, neither the strong (or normal) transitivity rule nor the weak transitivity rule can be admissible when we combine some natural subtyping rules, for example the subtyping rules for Σ-types and lists.

In this section, we discuss the new computation rules for parameterised inductive types. If these new computation rules are added to the original type theory, the above two problems will be solved and the extended type theory is expected to keep some important meta-properties such as Strong Normalisation and Church-Rosser. This leads us to fundamental future research that is important for coercive subtyping as well as for type theory itself.

5.8.1. New computation rule for lists

In this sub-section, we give a new computation rule for lists as an example and discuss some important meta-properties for the extended type theory.

In Example 5.1.1, we have seen that the normal transitivity rule $(Trans)$ is not admissible for the subtyping rule of lists because $map_{List}(B, C, c_2) \circ map_{List}(A, B, c_1)$ and $map_{List}(A, C, c_2 \circ c_1)$ are not computationally equal although they are extensionally equal. Now, if we add a new equation rule for lists

$$map_{List}(B, C, c_2) \circ map_{List}(A, B, c_1) = map_{List}(A, C, c_2 \circ c_1)$$

in the original type theory, the normal transitivity rule $(Trans)$ is admissible for the subtyping rules of lists and Σ-types. To prove this, we need the definition of *depth* in Definition 4.2.3 and the proof method is the same as that of Theorem 4.2.7.

According to the new equation rule, we also introduce a new computation (or reduction) rule for lists.

$$\begin{array}{ll} & \mathcal{E}_{List}(B, [l' : List(B)]List(C), nil(C), \\ & [b : B][l' : List(B)][l'' : List(C)]cons(C, t_2, l''), \\ & \mathcal{E}_{List}(A, [l : List(A)]List(B), nil(B), \\ & [a : A][l : List(A)][l' : List(B)]cons(B, t_1, l'), x_0)) \\ \Rightarrow & \mathcal{E}_{List}(A, [l : List(A)]List(C), nil(C), \\ & [a : A][l : List(A)][l'' : List(C)]cons(C, [t_1/b]t_2, l''), x_0) \end{array}$$

Notice that if $t_1 \equiv c_1(a)$ and $t_2 \equiv c_2(b)$, then $[t_1/b]t_2 \equiv c_2(c_1(a))$.

Remark 5.8.1 *We have the following remarks:*

- *It is better not to regard the new equation rule for lists as a computation (or reduction) rule. Otherwise, the property of Church-Rosser may fail. For instance, let's consider a term M that is the left hand side of the new computation rule by replacing A, B, C by N and t_1 by $([x : N]O)(a)$ and t_2 by $([x : N]O)(b)$ (*i.e. *$A \equiv B \equiv C \equiv N$ and $t_1 \equiv ([x : N]O)(a)$, $t_2 \equiv ([x : N]O)(b)$). If we regard the new equation rule for lists as a computation (or reduction) rule, there*

are at least two ways to reduce M. One way is to reduce t_1 and t_2 to O, then we get a normal form M_1. Another way is to use the new rule and reduce M to a normal form M_2. Since M_1 and M_2 are different normal forms of M, the property of Church-Rosser fails.

- *A computation (or reduction) rule is also an equation rule.*
- *We have proved that $map_{List}(B, C, c_2) \circ map_{List}(A, B, c_1)$ and $map_{List}(A, C, c_2 \circ c_1)$ are extensionally equal. Adding the equation rule for lists will not violate the logical consistency of the extended type theory.*
- *For the new computation rule for lists, we must require that variables l and l'' don't occur free in t_1 and, variables l' and l'' don't occur free in t_2 to guarantee the left hand side is really extensionally equal to the right hand side.*
- *We believe that some important meta-properties such as Strong Normalisation and Church-Rosser still hold after adding the new computation rule. Proving such meta-properties is out of the scope of this thesis.*

5.8.2. New computation rules in general

Consider a general form of parameterised inductive types:

$$\mathcal{T} =_{df} [Y_1 : P_1]...[Y_p : P_p]\mathcal{M}[\overline{\Theta}]$$

where $p \neq 0$ and $\overline{\Theta} \equiv <\Theta_1, ..., \Theta_s> \ (s \in \omega)$ a finite sequence of inductive schemata and P_i $(i = 1, ..., p)$ kinds (not necessarily small). We may introduce a new computation rule for $\mathcal{T}$ and add it into the original type theory as we did for lists. Details are omitted here.

Conjecture *After adding new computation rules for parameterised inductive data types, the extended type theory has all the properties which the original type theory has, such as the properties of Strong Normalisation, Church-Rosser, Subject Reduction, etc.*

CHAPTER 6

Combining Incoherent Coercions for Σ-types

In this chapter, we will consider a very useful coercion, π_1, the first projection of Σ-types. With this coercion, it is very easy to express some mathematical properties. For example, it is used significantly in Bailey's PhD thesis [Bai98] for formalisation of mathematics.

In Chapter 4, coherence was proved for the component-wise subtyping rule of Σ-types. However, when these subtyping rules are combined with the subtyping rule for the first projection, coherence fails to hold. A counter example will be given in the next section to illustrate this problem and explain the solution. We shall introduce a new subtyping relation and give a new formulation of coercive subtyping, to ensure that there is only one coercion (with respect to computational equality) between any two types (if there is one at all). This new formulation not only satisfies the coherence requirements but also enjoys other properties, particularly the admissibility of substitution and transitivity because such properties are important for an implementation of coercive subtyping.

6.1. The Coherence Problem

In this section, we give an example to illustrate the coherence problem of the component-wise subtyping rules for Σ-types and the subtyping rule of its first projection. Because of the coherence problem, we cannot uniformly use these two sets of subtyping relations in a single system.

6.1.1. Subtyping rules for Σ-types

As studied in Section 3.5, there are three component-wise subtyping rules for Σ-types. One of these rules is the following.

$$(First\,Component\,rule)\quad \frac{\Gamma \vdash A <_c A' : Type \quad \Gamma \vdash B : (A')Type}{\Gamma \vdash \Sigma(A, B \circ c) <_{d_1} \Sigma(A', B) : Type}$$

where $d_1 = [z : \Sigma(A, B \circ c)]pair(A', B, c(\pi_1(A, B \circ c, z)), \pi_2(A, B \circ c, z))$.

The coercion of the first projection is very useful for formalisation of mathematics [Bai98]. Formally, the subtyping rule is the following:

$$(\pi_1 rule) \qquad \frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : (A)Type}{\Gamma \vdash \Sigma(A,B) <_{\pi_1(A,B)} A : Type}$$

With this coercion, it is very easy to express some mathematical properties. For example, the type of collection of groups is a subtype of the type of semi-groups (*i.e.* a group is also a semi-group). Any functional operator with the domain of semi-groups can be applied to any group with a coercion.

6.1.2. A counter example

If the subtyping rule $(\pi_1 rule)$ and the component-wise subtyping rules for Σ-types are combined together, we have the following two derivations.

The first derivation is

$$\frac{\dfrac{\Gamma \vdash A : Type \quad \Gamma \vdash B : (A)Type}{\Gamma \vdash \Sigma(A,B) : Type} \qquad \dfrac{\Gamma \vdash B : (A)Type}{\Gamma \vdash B \circ \pi_1(A,B) : (\Sigma(A,B))Type}}{\Gamma \vdash \Sigma(\Sigma(A,B), B \circ \pi_1(A,B)) <_{d_1} \Sigma(A,B) : Type}$$

where the rule $(\pi_1 rule)$ is used in the last step.

The second derivation is

$$\frac{(\pi_1 rule)\dfrac{\Gamma \vdash A : Type \quad \Gamma \vdash B : (A)Type}{\Gamma \vdash \Sigma(A,B) <_{\pi_1(A,B)} A : Type} \qquad \Gamma \vdash B : (A)Type}{\Gamma \vdash \Sigma(\Sigma(A,B), B \circ \pi_1(A,B)) <_{d_2} \Sigma(A,B) : Type}$$

where the rule $(\pi_1 rule)$ is used in the first step and the First Component rule is used in the last step.

There are two coercions d_1 and d_2 from type $\Sigma(\Sigma(A,B), B\circ\pi_1(A,B))$ to type $\Sigma(A,B)$[1] and we have the following equations (some details are omitted here)

$$\begin{aligned} d_1(pair(pair(a,b_1),b_2)) &= pair(a,b_1) \\ d_2(pair(pair(a,b_1),b_2)) &= pair(a,b_2) \end{aligned}$$

[1]There are two different coercions from $(A \times B) \times B$ to $A \times B$ if A and B are types, where $A \times B$ is for $\Sigma(A,[x:A]B)$ and $x \notin FV(B)$.

We can see that d_1 and d_2 are neither computationally nor extensionally equal. Hence, the vital requirement of a coercive subtyping system, coherence, fails.

6.1.3. Informal explanation of the solution

From the above counter example, we see that the existence of the two derivations makes the system incoherent. To make it coherent, a natural way is to block one of the derivations. The first one cannot be blocked, otherwise we lose the meaning that the first projection (π_1) is regarded as coercion. Hence we can only block the second derivation. More precisely, we must not allow $\Gamma \vdash A <_c A' : Type$ to be used as the first premise of the component-wise subtyping rules if it is (directly) derived from the $\pi_1 rule$. In other words, a condition of the component-wise subtyping rules is that the first premise is not (directly) derived from the $\pi_1 rule$. There are several attempts to satisfy this condition, one of which is to consider a notion of size as a side-condition because A is a sub-term of $\Sigma(A, B)$ in the conclusion of $\pi_1 rule$, and their sizes are intuitively different. However, the well-definedness of size is problematic when we present the whole subtyping system (see the discussion section for more details).

In the next section, rather than thinking of any side-conditions, we introduce a new subtyping relation ($\prec$) to represent coercion π_1. This new subtyping relation will never appear in the first premises of the component-wise subtyping rules and hence the unwanted derivations such as the second one in the counter example are blocked.

To make the subtyping system coherent is one thing; to make it also enjoy the property of the admissibility of transitivity is another. During our investigation, we experienced that some formulations satisfy the property of coherence, but not the admissibility of transitivity. The formulation in the next section will enjoy all these properties.

6.2. A formal presentation

In this section, we shall give a formal presentation of a new subtyping relation and related subtyping rules. The coherence condition will also be redefined.

6.2.1. A new subtyping relation

We have seen the problem with the combination of the component-wise subtyping rules and the subtyping rule of the first projection. Now, we introduce a new relation to solve this problem, and consider a new system $T[\mathcal{R}\pi_1]$, which is an extension of coercive subtyping with the judgement form:

- $\Gamma \vdash A \prec_c B : Type$ asserts that type A is a subtype of type B with a coercion c.

As we will see later, subtyping relation $<$ and $\prec$ are different. $\prec$ represents the idea that π_1 is regarded as a coercion, but $<$ does not.

The coercive definition rules

The main idea of coercive subtyping can informally be represented by the following coercive definition rule (contexts are omitted):

$$\frac{K <_c K' \quad k : K \quad f : (x : K')K''}{f(k) = f(c(k)) : [c(k)/x]K''}$$

The same idea is followed for the new subtyping relation. A new basic subkinding rule for $\prec$ is the following:

$$\frac{A \prec_c B : Type}{El(A) <_c El(B)}$$

By the coercive definition rule, we have the following derivable rule:

$$\frac{A \prec_c B : Type \quad k : El(A) \quad f : (x : El(B))K}{f(k) = f(c(k)) : [c(k)/x]K}$$

which says that if $A \prec_c B$, any functional operator f with domain B can be applied to any object x of A and, $f(x) = f(c(x))$.

We present the new subtyping system in two stages: first an intermediate system $T[\mathcal{R}\pi_1]_0$ and the definition of coherence, and then the system $T[\mathcal{R}\pi_1]$.

6.2.2. The systems $T[\mathcal{R}\pi_1]_0$ and $T[\mathcal{R}\pi_1]$

Formally, $T[\mathcal{R}\pi_1]_0$ is an extension of type theory T (only) with the following rules:

- A set $\mathcal{R}$ of basic subtyping rules whose conclusions are subtyping judgements of the form $\Gamma \vdash A <_c B : Type$.

- The following congruence rule for subtyping judgements

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

- The new subtyping rules for the first projection in Figure 6.2.1, whose conclusions are of the form $\Gamma \vdash A \prec_c B : Type$.

Notation 6.2.1 *we shall use* $\Gamma \vdash A \propto_c B : Type$ *to represent* $\Gamma \vdash A <_c B : Type$ *or* $\Gamma \vdash A \prec_c B : Type$. *For example,*

$$\frac{\Gamma \vdash A \propto_c B : Type}{J}$$

actually represents two rules

$$\frac{\Gamma \vdash A <_c B : Type}{J} \quad and \quad \frac{\Gamma \vdash A \prec_c B : Type}{J}$$

and

$$\frac{\Gamma \vdash A \propto_c B : Type \quad \Gamma' \vdash A' \propto_{c'} B' : Type}{J}$$

actually represents four rules.
We shall also say that A ***is a subtype of*** *B or there is a coercion c from A to B if* $\Gamma \vdash A \propto_c B : Type$.

New subtyping rule for the first projection:

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : (A)Type}{\Gamma \vdash \Sigma(A, B) \prec_{\pi_1(A,B)} A : Type}$$

$$\frac{\Gamma \vdash A \propto_c A' : Type \quad \Gamma \vdash B : (A)Type}{\Gamma \vdash \Sigma(A, B) \prec_{c \circ \pi_1(A,B)} A' : Type}$$

New congruence rule:

$$\frac{\Gamma \vdash A \prec_c B : Type \qquad \Gamma \vdash A = A' : Type \quad \Gamma \vdash B = B' : Type \quad \Gamma \vdash c = c' : (A)B}{\Gamma \vdash A' \prec_{c'} B' : Type}$$

FIGURE 6.2.1. New subtyping rules for the first projection

Remark 6.2.2 *We have the following remarks:*

- *The basic understanding of the new subtyping rules for the first projection is that $\Sigma(A, B)$ is a subtype of A' if $A = A'$ or A is a subtype of A'.*
- *The two subtyping relations $<$ and $\prec$ are considered simultaneously and they contribute to each other.*
- *New substitution and transitivity rules for subtyping relations $<$ and $\prec$ will be given later and, we will prove that they are admissible. We do not include them in $T[\mathcal{R}\pi_1]_0$.*

In $T[\mathcal{R}\pi_1]_0$, the subtyping judgements do not contribute to any derivation of a judgement of any other forms in the original type theory T. Therefore, we have the following lemma.

Lemma 6.2.3 *$T[\mathcal{R}\pi_1]_0$ is a conservative extension of T.*

Now, we define the coherence requirement for the new coercive subtyping system in the following.

Definition 6.2.4 (Coherence condition of $T[\mathcal{R}\pi_1]_0$) *We say that $T[\mathcal{R}\pi_1]_0$ is coherent if it has the following properties.*

1. $\Gamma \vdash A \propto_c B : Type$ *implies* $\Gamma \vdash A : Type$, $\Gamma \vdash B : Type$, *and* $\Gamma \vdash c : (A)B$.
2. $\Gamma \vdash A \propto_c B : Type$ *implies* $\Gamma \not\vdash A = B : Type$.
3. $\Gamma \vdash A <_c B : Type$ *and* $\Gamma \vdash A <_{c'} B : Type$ *imply* $\Gamma \vdash c = c' : (A)B$.
4. $\Gamma \vdash A \prec_c B : Type$ *and* $\Gamma \vdash A \prec_{c'} B : Type$ *imply* $\Gamma \vdash c = c' : (A)B$.
5. *(Disjointedness)* $\Gamma \vdash A <_c B : Type$ *implies* $\Gamma \not\vdash A \prec_{c'} B : Type$ *for any c', and vice versa,* $\Gamma \vdash A \prec_c B : Type$ *implies* $\Gamma \not\vdash A <_{c'} B : Type$ *for any c'.*

Remark 6.2.5 *One may consider a more general coherence condition such as, if $\Gamma \vdash A \propto_c B : Type$ and $\Gamma \vdash A \propto_{c'} B : Type$ then $\Gamma \vdash c = c' : (A)B$. This will include the case in which both $\Gamma \vdash A <_c B : Type$ and $\Gamma \vdash A \prec_c B : Type$ may happen. However, one of the reasons we need the new subtyping relation ($\prec$) is deliberately to make sure that $\Gamma \vdash A <_c B : Type$ and $\Gamma \vdash A \prec_c B : Type$ may never hold at the same time for any A and B. Disjointedness is regarded as part of the coherence condition.*

The system $T[\mathcal{R}\pi_1]$

The system $T[\mathcal{R}\pi_1]$ is obtained from $T[\mathcal{R}\pi_1]_0$ by adding the inference rules in Figure 3.2.1 and in Figure 3.2.2 and the following *new basic subkinding rule.*

$$(\textit{New Basic Subkinding Rule}) \qquad \frac{\Gamma \vdash A \prec_c B : Type}{\Gamma \vdash El(A) <_c El(B)}$$

There is only one subkinding judgement form $\Gamma \vdash K <_c K'$, although there are two subtyping judgement forms $\Gamma \vdash A <_c B : Type$ and $\Gamma \vdash A \prec_c B : Type$. At the kind level, we are more concerned with the existence of a coercion no matter from which form it is derived at the type level.

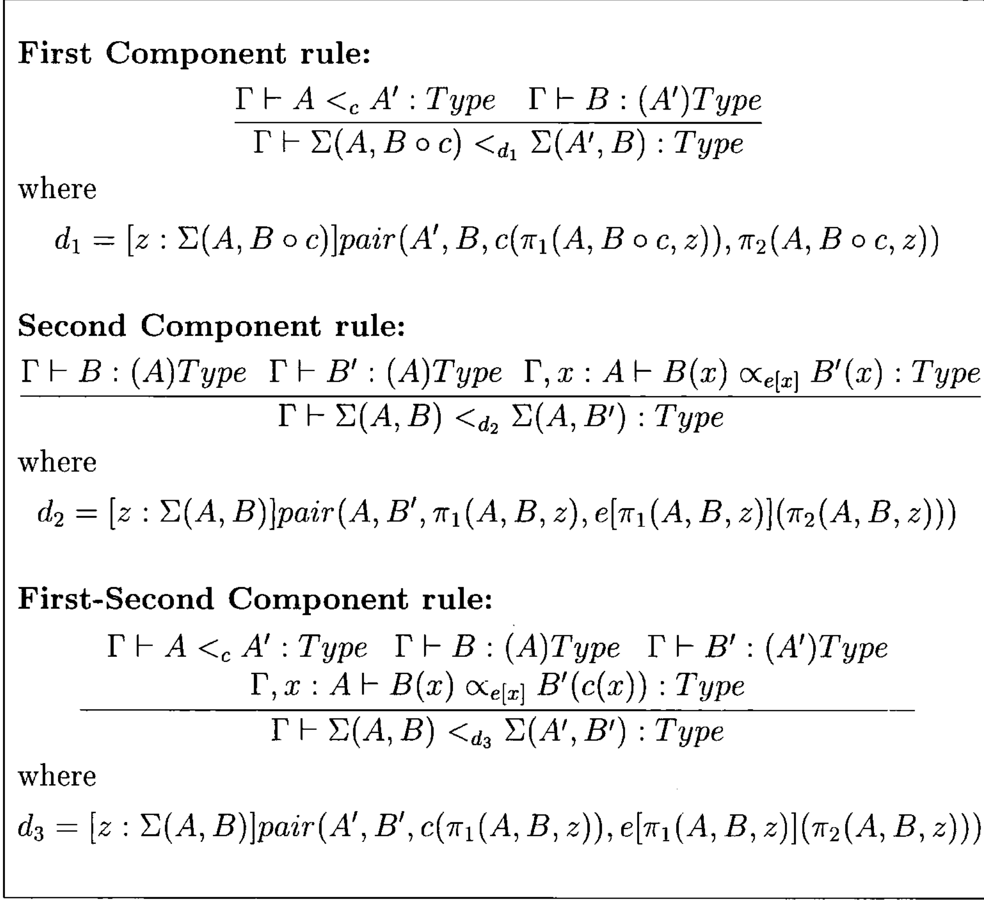
Remark 6.2.6 *The main result of* [SL02] *is essentially that coherence of subtyping rules does imply conservativity. In Section 6.4, we shall also prove the coherence of* $T[\mathcal{R}\pi_1]_0$. *So,* $T[\mathcal{R}\pi_1]$ *is also expected to be a conservative extension of* T.

6.3. New subtyping rules for inductive types

Now, we give the component-wise subtyping rules for Σ-types and the rules for Π-types in Figure 6.3.1 and Figure 6.3.2 to demonstrate what the subtyping rules should be for the new subtyping relation.

Remark 6.3.1 *We have the following remarks:*

- *In Figure 6.3.1 and Figure 6.3.2, the conclusions of the rules are always of the form* $\Gamma \vdash A <_c B : Type$, *no matter whether the premises are of the form* $\Gamma \vdash A <_c B : Type$ *or* $\Gamma \vdash A \prec_c B : Type$.
- *The essence of the new subtyping relation is that, the judgement form* $\Gamma \vdash A \prec_c B : Type$ *is never used in the premises of the first component of the component-wise subtyping rules in Figure 6.3.1. Hence the second derivation of the counter example in section 6.1 is blocked.*
- *The basic understanding of the new subtyping rules for* Π*-types is that* $\Pi(A, B)$ *is a subtype of* $\Pi(A', B')$ *if* A' *is a subtype of* A *and* B *is a sub-family of* B' *(we omit other cases such as:* $\Pi(A, B)$ *is a subtype of* $\Pi(A, B')$ *if* B *is a sub-family of* B'*).*
- *For the new component-wise subtyping rules for* Σ*-types, because of the incoherence when* π_1 *is also regarded as a coercion, we need*



First Component rule:

$$\frac{\Gamma \vdash A <_c A' : Type \quad \Gamma \vdash B : (A')Type}{\Gamma \vdash \Sigma(A, B \circ c) <_{d_1} \Sigma(A', B) : Type}$$

where

$$d_1 = [z : \Sigma(A, B \circ c)]pair(A', B, c(\pi_1(A, B \circ c, z)), \pi_2(A, B \circ c, z))$$

Second Component rule:

$$\frac{\Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A)Type \quad \Gamma, x : A \vdash B(x) \propto_{e[x]} B'(x) : Type}{\Gamma \vdash \Sigma(A, B) <_{d_2} \Sigma(A, B') : Type}$$

where

$$d_2 = [z : \Sigma(A, B)]pair(A, B', \pi_1(A, B, z), e[\pi_1(A, B, z)](\pi_2(A, B, z)))$$

First-Second Component rule:

$$\frac{\Gamma \vdash A <_c A' : Type \quad \Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A')Type \qquad \Gamma, x : A \vdash B(x) \propto_{e[x]} B'(c(x)) : Type}{\Gamma \vdash \Sigma(A, B) <_{d_3} \Sigma(A', B') : Type}$$

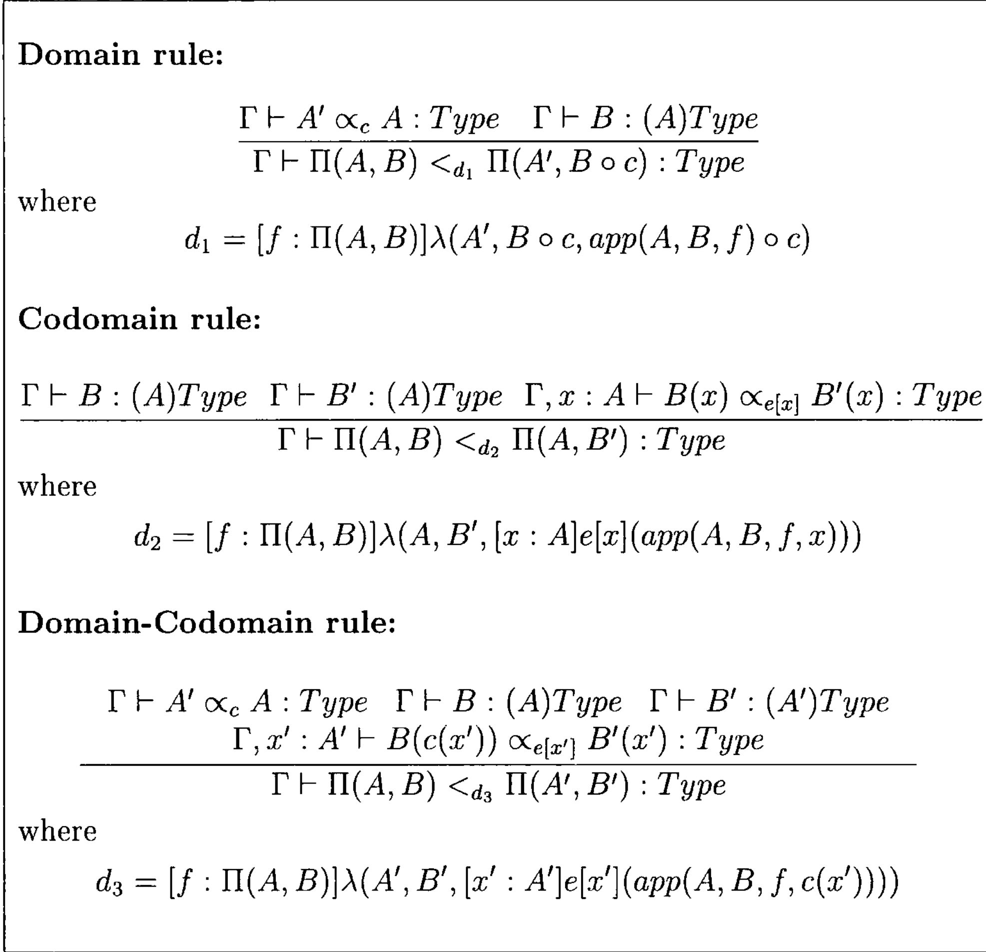
where

$$d_3 = [z : \Sigma(A, B)]pair(A', B', c(\pi_1(A, B, z)), e[\pi_1(A, B, z)](\pi_2(A, B, z)))$$

FIGURE 6.3.1. New component-wise subtyping rules for Σ-types

to have a stricter understanding, that is, $\Sigma(A, B)$ is a subtype of $\Sigma(A', B')$ if A is a subtype of A' and B is a sub-family of B' and the sizes of A and A' are the same (size is defined in Definition 6.4.3). In the following section, we will prove that the sizes of A and B are the same if $\Gamma \vdash A <_c B : Type$ and, the size of A is bigger than the size of B if $\Gamma \vdash A \prec_c B : Type$.

The subtyping system we present here covers all the coercions derived from the component-wise subtyping rules and the subtyping rule for the first projection when they are used separately. Actually, it has more coercions. For example, if A, B and C are different types, we can have a coercion from $A \times (B \times C)$ to $A \times B$ because there is a coercion from $B \times C$ to B. However, we can never derive a coercion from $A \times (B \times C)$ to $A \times B$ by the component-wise subtyping rules or



Domain rule:

$$\frac{\Gamma \vdash A' \propto_c A : Type \quad \Gamma \vdash B : (A)Type}{\Gamma \vdash \Pi(A, B) <_{d_1} \Pi(A', B \circ c) : Type}$$

where

$$d_1 = [f : \Pi(A, B)]\lambda(A', B \circ c, app(A, B, f) \circ c)$$

Codomain rule:

$$\frac{\Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A)Type \quad \Gamma, x : A \vdash B(x) \propto_{e[x]} B'(x) : Type}{\Gamma \vdash \Pi(A, B) <_{d_2} \Pi(A, B') : Type}$$

where

$$d_2 = [f : \Pi(A, B)]\lambda(A, B', [x : A]e[x](app(A, B, f, x)))$$

Domain-Codomain rule:

$$\frac{\Gamma \vdash A' \propto_c A : Type \quad \Gamma \vdash B : (A)Type \quad \Gamma \vdash B' : (A')Type \quad \Gamma, x' : A' \vdash B(c(x')) \propto_{e[x']} B'(x') : Type}{\Gamma \vdash \Pi(A, B) <_{d_3} \Pi(A', B') : Type}$$

where

$$d_3 = [f : \Pi(A, B)]\lambda(A', B', [x' : A']e[x'](app(A, B, f, c(x'))))$$

FIGURE 6.3.2. New subtyping rules for Π-types

the subtyping rule for the first projection separately. What we have excluded are those coercions that need component-wise subtyping rules for Σ-types but the sizes of their first components are different. For example, we don't have a coercion from $(A \times B) \times C$ to $A \times C$ because the sizes of $A \times B$ and A are different although there is a coercion from $A \times B$ to A.

6.4. Coherence of $T[\mathcal{R}\pi_1]_0$

Now, we prove the coherence of $T[\mathcal{R}\pi_1]_0$, which essentially says that coercions between any two types must be unique. The set $\mathcal{R}$ of basic subtyping consists of the rule $(WDCrule)$ where $\mathcal{C}$ in the rule is a set of well-defined coercions (WDC) and the new subtyping rules for Σ-types and Π-types (in Figure 6.3.1 and Figure 6.3.2) and, the system $T[\mathcal{R}\pi_1]_0$

also includes the congruence rule $(Cong)$ and the new subtyping rules in Figure 6.2.1. Furthermore, we assume that for any judgement $\Gamma \vdash A <_c B : Type \in \mathcal{C}$, neither A nor B is computationally equal to a Σ-type or Π-type. We also assume that the original type theory T has good properties, in particular the properties of Church-Rosser and Strong Normalisation and the property of context replacement by equal kinds.

We give a definition of $size(A)$ that only counts how many times π_1 can be applied for an object of type A. In order to define $size$, we define $presize$ first.

Definition 6.4.1 ***(presize)*** *Let $\Gamma \vdash M : Type$ be a derivable judgement in $T[\mathcal{R}\pi_1]_0$ and M a normal form (*i.e. *$M \equiv nf(M)$),*

1. *if M is not a Σ-type then $presize(M) =_{df} 0$,*
2. *if $M \equiv \Sigma(A, B)$ then $presize(M) =_{df} presize(A) + 1$.*

Remark 6.4.2 *For the second case, because M is a normal form, so is A. Therefore presize is well-defined.*

Definition 6.4.3 ***(size)*** *The definition of size in $T[\mathcal{R}\pi_1]_0$:*
Let $\Gamma \vdash M : Type$ be a derivable judgement in $T[\mathcal{R}\pi_1]_0$,

$$size(M) =_{df} presize(nf(M))$$

where $nf(M)$ means the normal form of M.

Remark 6.4.4 *$T[\mathcal{R}\pi_1]_0$ is a conservative extension of T. Therefore, every well-typed term in T has its unique normal form. So, the value of $size(M)$ is unique and size is well-defined.*

Lemma 6.4.5 *In $T[\mathcal{R}\pi_1]_0$, if $\Gamma \vdash M_1 = M_2 : Type$ is derivable then $size(M_1) = size(M_2)$.*

Proof. $T[\mathcal{R}\pi_1]_0$ is a conservative extension of T and T has properties of Church-Rosser and strong normalisation, therefore, M_1 and M_2 have the same normal form, *i.e.* $nf(M_1) \equiv nf(M_2)$. □

Lemma 6.4.6 *Let $\Gamma \vdash M : Type$ be a derivable judgement in $T[\mathcal{R}\pi_1]_0$.*

- *if M is not computationally equal to a Σ-type then $size(M) = 0$ and,*

- *if* $\Gamma \vdash M = \Sigma(A, B) : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$ *then*

$$size(M) = size(A) + 1$$

Proof. By the definition of *size* and Lemma 6.4.5. □

Lemma 6.4.7 *In* $T[\mathcal{R}\pi_1]_0$, *if* $\Gamma \vdash M_1 <_d M_2 : Type$ *is derivable then* $size(M_1) = size(M_2)$.

Proof. By induction on derivations and using Lemma 6.4.5 and Lemma 6.4.6. Note that $size(M_1) = size(M_2) = 0$ if the last rule of the derivation of $\Gamma \vdash M_1 <_d M_2 : Type$ is one of the rules for Π-types. □

Lemma 6.4.8 *In* $T[\mathcal{R}\pi_1]_0$, *if* $\Gamma \vdash M_1 \prec_c M_2 : Type$ *is derivable then* $size(M_1) > size(M_2)$.

Proof. By induction on derivations and Lemma 6.4.5, Lemma 6.4.6 and Lemma 6.4.7. □

The following theorems prove the coherence of $T[\mathcal{R}\pi_1]_0$.

Theorem 6.4.9

- *If* $\Gamma \vdash M_1 \propto_c M_2 : Type$ *then* $\Gamma \vdash M_1 : Type$, $\Gamma \vdash M_2 : Type$ *and* $\Gamma \vdash c : (M_1)M_2 : Type$.
- *If* $\Gamma \vdash M_1 \propto_c M_2 : Type$ *then* $\Gamma \nvdash M_1 = M_2 : Type$.
- *(Disjointedness) If* $\Gamma \vdash M_1 \prec_c M_2 : Type$ *then*
  $\Gamma \nvdash M_1 <_d M_2 : Type$ *for any d; and vice versa,*
  if $\Gamma \vdash M_1 <_c M_2 : Type$, *then* $\Gamma \nvdash M_1 \prec_d M_2 : Type$ *for any d.*

Proof. By induction on derivations, the definition of WDC, Lemma 6.4.7 and Lemma 6.4.8. □

Notation 6.4.10 *We shall simply write* $\Gamma \vdash J$ *when it is a derivable judgement in* $T[\mathcal{R}\pi_1]_0$. *Sometimes, we shall also write* $\Gamma \vdash A \propto_c B$ *for* $\Gamma \vdash A \propto_c B : Type$, *and* $\Gamma \vdash k_1 = k_2$ *for* $\Gamma \vdash k_1 = k_2 : K$, *when no confusion may occur.*

Theorem 6.4.11 *If* $\vdash \Gamma = \Gamma'$, $\Gamma \vdash M_1 = M_1' : Type$, $\Gamma \vdash M_2 = M_2' : Type$, *and*

1. $\Gamma \vdash M_1 <_d M_2 : Type$ *and* $\Gamma' \vdash M_1' <_{d'} M_2' : Type$, *or*
2. $\Gamma \vdash M_1 \prec_d M_2 : Type$ *and* $\Gamma' \vdash M_1' \prec_{d'} M_2' : Type$

then $\Gamma \vdash d = d' : (M_1)M_2$.

Proof. By induction on derivations. A most important argument in this proof is that any derivations of $\Gamma \vdash M_1 <_d M_2$ and $\Gamma' \vdash M_1' <_d M_2'$, or $\Gamma \vdash M_1 \prec_d M_2$ and $\Gamma' \vdash M_1' \prec_{d'} M_2'$ must contain sub-derivations whose last rules are the same rule, followed by a finite number of applications of the congruence rules. In the following, we choose one case to demonstrate how the proof proceeds. The proofs of other cases are similar.

Suppose the derivation of $\Gamma \vdash M_1 \prec_d M_2$ is of the following form. It contains a sub-derivation whose last rule is one of the subtyping rules for the first projection followed by a finite number of applications of the new congruence rule in Figure 6.2.1.

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\vdots & & \vdots \\ \Gamma \vdash A_1 <_c A_2 & & \Gamma \vdash B_1 : (A_1)Type\end{matrix}}{\Gamma \vdash \Sigma(A_1, B_1) \prec_{d_1} A_2}}{\vdots}}{\begin{matrix}...(New\ \ congruence\ \ rule)... \\ \vdots\end{matrix}} \quad \Rightarrow \quad \Gamma \vdash M_1 \prec_d M_2$$

where $\Gamma \vdash \Sigma(A_1, B_1) = M_1$, $\Gamma \vdash A_2 = M_2$, $\Gamma \vdash d_1 = d$ and

$$d_1 = c \circ \pi_1(A_1, B_1)$$

Now, it must be the case that any derivation of $\Gamma' \vdash M_1' \prec_{d'} M_2'$ is of the form:

$$\dfrac{\dfrac{\begin{matrix}\vdots & & \vdots \\ \Gamma' \vdash A_1' <_{c'} A_2' & & \Gamma' \vdash B_1' : (A_1')Type\end{matrix}}{\Gamma' \vdash \Sigma(A_1', B_1') \prec_{d_1'} A_2'}}{\begin{matrix}\vdots \\ ...(New\ \ congruence\ \ rule)... \\ \vdots\end{matrix}} \quad \Rightarrow \quad \Gamma' \vdash M_1' \prec_{d'} M_2'$$

where $\Gamma' \vdash \Sigma(A_1', B_1') = M_1'$, $\Gamma' \vdash A_2' = M_2'$, $\Gamma' \vdash d' = d_1'$ and

$$d_1' = c' \circ \pi_1(A_1', B_1')$$

In other words, any derivation of $\Gamma' \vdash M'_1 \prec_{d'} M'_2$ must contain a sub-derivation whose last rule is also the same subtyping rule as that in the derivation of $\Gamma \vdash M_1 \prec_d M_2$. To see this is the case, because $\Gamma' \vdash \Sigma(A'_1, B'_1) \prec_{d'_1} A'_2$ must be derived from one of the subtyping rules for the first projection 6.2.1, we have to show only that $\Gamma' \not\vdash A'_1 = A'_2$ and $\Gamma' \not\vdash A'_1 \prec_e A'_2$ for any e. Since $\Gamma \vdash M_1 = M'_1$ and $\Gamma \vdash M_2 = M'_2$, we have $\Gamma \vdash \Pi(A_1, B_1) = \Pi(A'_1, B'_1)$ and $\Gamma \vdash A_2 = A'_2$. Hence, by Church-Rosser in T and conservativity of $T[\mathcal{R}]_0$ over T, we have $\Gamma \vdash A_1 = A'_1$ as well. As $\Gamma \vdash A_1 <_c A_2$, we have $\Gamma \not\vdash A_1 = A_2$ and $size(A_1) = size(A_2)$ by Theorem 6.4.9 and Lemma 6.4.7. So $\Gamma' \not\vdash A'_1 = A'_2$ because $\Gamma \vdash A_1 = A'_1$ and $\Gamma \vdash A_2 = A'_2$, and $\Gamma' \not\vdash A'_1 \prec_e A'_2$ for any e by Lemma 6.4.8.

Now, since the derivations must be of the above forms, by the induction hypothesis, we have $\Gamma \vdash c = c'$. So, $\Gamma \vdash d_1 = d'_1$ and hence $\Gamma \vdash d = d'$. □

6.5. Admissibility of substitution and transitivity

Now, we give the subtyping rules of substitution and transitivity and prove that these rules are admissible. In an implementation of coercive subtyping, these rules are ignored simply because they cannot be directly implemented. For this reason, among others, proving the admissibility of such rules (or their elimination) is always an important task for any subtyping system.

Admissible substitution rules

The substitution rules are as follows, which are what we expect normally.

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

$$\frac{\Gamma, x : K, \Gamma' \vdash A \prec_c B : Type \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]A \prec_{[k/x]c} [k/x]B : Type}$$

Admissible transitivity rules

We give the following four transitivity rules that are basically saying that if there are coercions c and c' from type A to B and from type B

to C, then $c' \circ c$ is a coercion from type A to C.

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

$$\frac{\Gamma \vdash A \prec_{c_1} B : Type \quad \Gamma \vdash B \prec_{c_2} C : Type}{\Gamma \vdash A \prec_{c_2 \circ c_1} C : Type}$$

$$\frac{\Gamma \vdash A <_{c_1} B : Type \quad \Gamma \vdash B \prec_{c_2} C : Type}{\Gamma \vdash A \prec_{c_2 \circ c_1} C : Type}$$

$$\frac{\Gamma \vdash A \prec_{c_1} B : Type \quad \Gamma \vdash B <_{c_2} C : Type}{\Gamma \vdash A \prec_{c_2 \circ c_1} C : Type}$$

Remark 6.5.1 *The above transitivity rules are sufficient and correct, in the sense that, first, they capture the meaning of transitivity, and second, they enjoy the properties in Lemma 6.4.7 and Lemma 6.4.8. Other rules of different combinations such as the rule*

$$\frac{\Gamma \vdash A <_{c_1} B : Type \quad \Gamma \vdash B <_{c_2} C : Type}{\Gamma \vdash A \prec_{c_2 \circ c_1} C : Type}$$

are not correct and are contradictory to the above properties. (According to the premises in the above rule, $size(A) = size(B) = size(C)$, but according to the conclusion, $size(A) > size(C)$.)

Theorem 6.5.2 ***(Substitution in** $T[\mathcal{R}\pi_1]_0$**)** If $\Gamma \vdash k : K$ and*

1. *if $\Gamma, x : K, \Gamma' \vdash M_1 <_c M_2 : Type$, then*
   $\Gamma, [k/x]\Gamma' \vdash [k/x]M_1 <_{[k/x]c} [k/x]M_2 : Type$, and
2. *if $\Gamma, x : K, \Gamma' \vdash M_1 \prec_c M_2 : Type$, then*
   $\Gamma, [k/x]\Gamma' \vdash [k/x]M_1 \prec_{[k/x]c} [k/x]M_2 : Type$.

Proof. By induction on derivations. □

In order to prove the admissibility of the transitivity rules, we also need to prove the theorem regarding weakening.

Theorem 6.5.3 ***(Weakening in** $T[\mathcal{R}\pi_1]_0$**)** If $\Gamma \subseteq \Gamma'$, Γ' is valid and*

1. *if $\Gamma \vdash M_1 <_c M_2 : Type$ then $\Gamma' \vdash M_1 <_c M_2 : Type$, and*
2. *if $\Gamma \vdash M_1 \prec_c M_2 : Type$ then $\Gamma' \vdash M_1 \prec_c M_2 : Type$.*

Proof. By induction on derivations. □

To prove the admissibility of the transitivity rules, the usual methods (*e.g.* by induction on derivations) do not seem to work. We develop a new measure (*Depth*) that is an adoption of the measure (*depth*) developed by Chen, Aspinall and Companoni [Che98]. In the measure *Depth*, the subtyping judgements ($<$ and $\prec$) only count.

Definition 6.5.4 ***(Depth)*** *Let D be a derivation of a subtyping judgement of the form $\Gamma \vdash A <_c B : Type$ or $\Gamma \vdash A \prec_c B : Type$.*

$$D: \qquad \frac{S_1 \quad ... \quad S_n \quad T_1 \quad ... \quad T_m}{\Gamma \vdash A \propto_c B : Type}$$

where $\Gamma \vdash A \propto_c B : Type$ represents $\Gamma \vdash A <_c B : Type$ or $\Gamma \vdash A \prec_c B : Type$, S_1,...,S_n are derivations of subtyping judgements of the form $\Gamma \vdash M_1 <_d M_2 : Type$ or $\Gamma \vdash M_1 \prec_d M_2 : Type$ and, T_1,..., T_m are derivations of other forms of judgements,

$$Depth(D) =_{df} 1 + max\{Depth(S_1), ..., Depth(S_n)\}$$

Specially, if $n = 0$ then $Depth(D) =_{df} 1$.

The following lemmas show that, from a derivation D of a subtyping judgement J, one can always get a derivation D' of the judgement obtained from J by context replacement such that D and D' have the same depth.

Lemma 6.5.5 *If $\vdash \Gamma = \Gamma'$ and*

1. *if D is a derivation of $\Gamma \vdash M_1 <_d M_2 : Type$, then there is a derivation D' of $\Gamma' \vdash M_1 <_d M_2 : Type$ such that $Depth(D) = Depth(D')$, or*
2. *if D is a derivation of $\Gamma \vdash M_1 \prec_d M_2 : Type$, then there is a derivation D' of $\Gamma' \vdash M_1 \prec_d M_2 : Type$ such that $Depth(D) = Depth(D')$.*

Proof. By induction on derivations. □

Lemma 6.5.6 *If $\Gamma \vdash c_2 : (K')K$ and,*

1. *if D is a derivation of $\Gamma, x : K, \Gamma' \vdash M_1 <_{c_1} M_2 : Type$, then there is a derivation D' of*

$$\Gamma, y : K', [c_2(y)/x]\Gamma' \vdash [c_2(y)/x]M_1 <_{[c_2(y)/x]c_1} [c_2(y)/x]M_2 : Type$$

such that $Depth(D) = Depth(D')$, or

2. *if D is a derivation of $\Gamma, x : K, \Gamma' \vdash M_1 \prec_{c_1} M_2 : Type$, then there is a derivation D' of*

$$\Gamma, y : K', [c_2(y)/x]\Gamma' \vdash [c_2(y)/x]M_1 \prec_{[c_2(y)/x]c_1} [c_2(y)/x]M_2 : Type$$

such that $Depth(D) = Depth(D')$.

Proof. By induction on derivations and Lemma 4.2.5. The theorem of weakening and substitution in type theory T and the property of conservativity of $T[\mathcal{R}\pi_1]_0$ over T are also needed in this proof. In the following, we choose one case to demonstrate how the proof proceeds.

Suppose that the last rule of the derivation D of $\Gamma, x : K, \Gamma' \vdash M_1 \prec_{c_1} M_2$ is

$$\frac{\Gamma, x : K, \Gamma' \vdash A <_c M_2 \quad \Gamma, x : K, \Gamma' \vdash B : (A)Type}{\Gamma, x : K, \Gamma' \vdash \Sigma(A, B) \prec_{c_1} M_2}$$

where $M_1 \equiv \Sigma(A, B)$ and $c_1 \equiv c \circ \pi_1(A, B)$. If we denote the derivation of $\Gamma, x : K, \Gamma' \vdash A <_c M_2$ as D_0, then we have $Depth(D) = Depth(D_0) + 1$ by the definition of *Depth*.

By the induction hypothesis, there is a derivation D_1 of

$$\Gamma, y : K', [c_2(y)/x]\Gamma' \vdash [c_2(y)/x]A <_{[c_2(y)/x]c} [c_2(y)/x]M_2$$

such that $Depth(D_0) = Depth(D_1)$.

Since $\Gamma \vdash c_2 : (K')K$, we know $\Gamma, y : K'$ is a valid context and $\Gamma, y : K' \vdash c_2(y) : K$ provided y is fresh. By the property of conservativity of $T[\mathcal{R}\pi_1]_0$ over T and the theorem of weakening in T, we have $\Gamma, y : K', x : K, \Gamma' \vdash B : (A)Type$. Then by the theorem of substitution in T, we have

$$\Gamma, y : K', [c_2(y)/x]\Gamma' \vdash [c_2(y)/x]B : ([c_2(y)/x]A)Type$$

Using the above subtyping rule, we have a derivation D' of

$$\Gamma, y : K', [c_2(y)/x]\Gamma' \vdash [c_2(y)/x]M_1 \prec_{[c_2(y)/x]c_1} [c_2(y)/x]M_2 : Type$$

and

$$Depth(D') = Depth(D_1) + 1 = Depth(D_0) + 1 = Depth(D)$$

□

Now, we prove the admissibility of the transitivity rules.

Theorem 6.5.7 *(Transitivity in $T[\mathcal{R}\pi_1]_0$)* *If* $\Gamma \vdash M_2 = M_2' : Type$ *and*

1. *if* $\Gamma \vdash M_1 <_{d_1} M_2 : Type$ *and* $\Gamma \vdash M_2' <_{d_2} M_3 : Type$, *then* $\Gamma \vdash M_1 <_{d_2 \circ d_1} M_3 : Type$, *and*
2. *if* $\Gamma \vdash M_1 \prec_{d_1} M_2 : Type$ *and* $\Gamma \vdash M_2' \prec_{d_2} M_3 : Type$, *then* $\Gamma \vdash M_1 \prec_{d_2 \circ d_1} M_3 : Type$.
3. *if* $\Gamma \vdash M_1 <_{d_1} M_2 : Type$ *and* $\Gamma \vdash M_2' \prec_{d_2} M_3 : Type$, *then* $\Gamma \vdash M_1 \prec_{d_2 \circ d_1} M_3 : Type$, *and*
4. *if* $\Gamma \vdash M_1 \prec_{d_1} M_2 : Type$ *and* $\Gamma \vdash M_2' <_{d_2} M_3 : Type$, *then* $\Gamma \vdash M_1 \prec_{d_2 \circ d_1} M_3 : Type$, *and*

Proof. By induction on $Depth(D) + Depth(D')$, where D is a derivation of $\Gamma \vdash M_1 <_{d_1} M_2 : Type$ or $\Gamma \vdash M_1 \prec_{d_1} M_2 : Type$, D' is a derivation of $\Gamma \vdash M_2' <_{d_2} M_3 : Type$ or $\Gamma \vdash M_2' \prec_{d_2} M_3 : Type$.

- In the base case *i.e.* $Depth(D) = Depth(D') = 1$, we consider the following four sub-cases:

  1. The derivations D and D' are:

  $$\frac{\Gamma \vdash M_1 <_{d_1} M_2 \in \mathcal{C}}{\Gamma \vdash M_1 <_{d_1} M_2}$$

  $$\frac{\Gamma \vdash M_2' <_{d_2} M_3 \in \mathcal{C}}{\Gamma \vdash M_2' <_{d_2} M_3}$$

  For this case, by Lemma 3.4.2, we have $\Gamma \vdash M_1 <_{d_2 \circ d_1} M_3 \in \mathcal{C}$.

  2. The derivations D and D' are:

  $$\frac{\Gamma \vdash M_1 <_{d_1} M_2 \in \mathcal{C}}{\Gamma \vdash M_1 <_{d_1} M_2}$$

  $$\frac{\Gamma \vdash M_3 : Type \quad \Gamma \vdash B : (M_3)Type}{\Gamma \vdash \Sigma(M_3, B) \prec_{d_2} M_3}$$

  where $M_2' \equiv \Sigma(M_3, B)$.

  Since $\Gamma \vdash M_1 <_{d_1} M_2 \in \mathcal{C}$, by the requirement of $\mathcal{C}$, M_2 is not computationally equal to a Σ-type. Since $\Gamma \vdash M_2 = M_2'$, M_2' cannot be a Σ-type. Therefore, this is an impossible case.

  3. The derivations D and D' are:

  $$\frac{\Gamma \vdash M_2 : Type \quad \Gamma \vdash B : (M_2)Type}{\Gamma \vdash \Sigma(M_2, B) \prec_{\pi_1(M_2,B)} M_2}$$

  $$\frac{\Gamma \vdash M_2' <_{d_2} M_3 \in \mathcal{C}}{\Gamma \vdash M_2' <_{d_2} M_3}$$

where $M_1 \equiv \Sigma(M_2, B)$ and $d_1 \equiv \pi_1(M_2, B)$.
Since $\Gamma \vdash M_2' <_{d_2} M_3 \in \mathcal{C}$ and $\Gamma \vdash M_2 = M_2'$, we have $\Gamma \vdash M_2 <_{d_2} M_3 \in \mathcal{C}$. Therefore we have the following derivation:

$$\frac{\dfrac{\Gamma \vdash M_2 <_{d_2} M_3 \in \mathcal{C}}{\Gamma \vdash M_2 <_{d_2} M_3} \quad \Gamma \vdash B : (M_2)Type}{\Gamma \vdash \Sigma(M_2, B) \prec_{d_2 \circ \pi_1(M_2,B)} M_3}$$

4. The derivations D and D' are:

$$\frac{\Gamma \vdash M_2 : Type \quad \Gamma \vdash B : (M_2)Type}{\Gamma \vdash \Sigma(M_2, B) \prec_{\pi_1(M_2,B)} M_2}$$

$$\frac{\Gamma \vdash M_3 : Type \quad \Gamma \vdash B' : (M_3)Type}{\Gamma \vdash \Sigma(M_3, B') \prec_{\pi_1(M_3,B')} M_3}$$

where $M_1 \equiv \Sigma(M_2, B)$, $M_2' \equiv \Sigma(M_3, B')$, $d_1 \equiv \pi_1(M_2, B)$ and $d_2 \equiv \pi_1(M_3, B')$.
Since $\Gamma \vdash M_2 = M_2'$, $M_2' \equiv \Sigma(M_3, B')$ and $\Gamma \vdash \Sigma(M_3, B') \prec_{\pi_1(M_3,B')} M_3$, by the new congruence rule, we have $\Gamma \vdash M_2 \prec_{\pi_1(M_3,B')} M_3$. Therefore, we have the following derivation:

$$\frac{\Gamma \vdash M_2 \prec_{\pi_1(M_3,B')} M_3 \quad \Gamma \vdash B : (M_2)Type}{\Gamma \vdash \Sigma(M_2, B) \prec_{\pi_1(M_3,B') \circ \pi_1(M_2,B)} M_3}$$

- In the step case, we choose one case to demonstrate how the proof proceeds. Suppose that the derivation D and D' be of the following forms:

$$\frac{\begin{array}{c}\vdots \\ D_1 \\ \vdots \\ \Gamma \vdash A_2 \prec_{c_1} A_1\end{array} \qquad \begin{array}{c}\vdots \\ D_2 \\ \vdots \\ \Gamma, x : A_2 \vdash B_1(c_1(x)) \prec_{e_1[x]} B_2(x)\end{array}}{\Gamma \vdash \Pi(A_1, B_1) <_{d_1'} \Pi(A_2, B_2)}$$

$$\frac{\begin{array}{c}\vdots \\ ...(Congruence \quad rule)... \\ \vdots\end{array}}{\Gamma \vdash M_1 <_{d_1} M_2}$$

where $\Gamma \vdash \Pi(A_1, B_1) = M_1$, $\Gamma \vdash \Pi(A_2, B_2) = M_2$, $\Gamma \vdash d_1' = d_1$ and

$d_1' = [f : \Pi(A_1, B_1)]\lambda(A_2, B_2, [x : A_2]e_1[x](app(A_1, B_1, f, c_1(x))))$

and

$$\frac{\begin{matrix} \vdots & & \vdots \\ D_1' & & D_2' \\ \vdots & & \vdots \\ \Gamma \vdash A_3 <_{c_2} A_2' & & \Gamma, x : A_3 \vdash B_2'(c_2(x)) <_{e_2[x]} B_3(x) \end{matrix}}{\dfrac{\Gamma \vdash \Pi(A_2', B_2') <_{d_2'} \Pi(A_3, B_3) \quad \vdots \quad ...(Congruence \quad rule)... \quad \vdots}{\Gamma \vdash M_2' <_{d_2} M_3}}$$

where $\Gamma \vdash \Pi(A_2', B_2') = M_2'$, $\Gamma \vdash \Pi(A_3, B_3) = M_3$, $\Gamma \vdash d_2' = d_2$ and

$d_2' = [f : \Pi(A_2', B_2')]\lambda(A_3, B_3, [x : A_3]e_2[x](app(A_2', B_2', f, c_2(x))))$

We obviously have $Depth(D_1) < Depth(D)$ and
$Depth(D_2) < Depth(D)$ because D_1 and D_2 are sub-derivations of D; $Depth(D_1') < Depth(D')$ and $Depth(D_2') < Depth(D')$ because D_1' and D_2' are sub-derivations of D'.
Now, since $\Gamma \vdash M_2 = M_2'$, we have by the Church-Rosser theorem of T and conservativity of $T[\mathcal{R}]_0$ over T, $\Gamma \vdash A_2 = A_2'$ and
$\Gamma \vdash B_2 = B_2'$. Since $\Gamma \vdash A_3 <_{c_2} A_2'$ we have $\Gamma \vdash c_2 : (A_3)A_2'$ and $\Gamma \vdash c_2 : (A_3)A_2$. Since $\Gamma, x : A_2 \vdash B_1(c_1(x)) \prec_{e_1[x]} B_2(x)$, by Lemma 6.5.6, we have $\Gamma, x : A_3 \vdash B_1(c_1(c_2(x))) \prec_{e_1[c_2(x)]} B_2(c_2(x))$ and there is a derivation D_3 of the judgement
$\Gamma, x : A_3 \vdash B_1(c_1(c_2(x))) \prec_{e_1[c_2(x)]} B_2(c_2(x))$
such that $Depth(D_3) = Depth(D_2)$.
Now, we have

$$Depth(D_1) + Depth(D_1') < Depth(D) + Depth(D')$$

$$Depth(D_3) + Depth(D_2') < Depth(D) + Depth(D')$$

By the induction hypothesis, we have that $\Gamma \vdash A_3 \prec_{c_1 \circ c_2} A_1$. Since $\Gamma \vdash B_2 = B_2' : (A_2)Type$ and $\Gamma \vdash c_2 : (A_3)A_2$, we have
$\Gamma, x : A_3 \vdash B_2(c_2(x)) = B_2'(c_2(x))$. By the induction hypothesis again, we have

$$\Gamma, x : A_3 \vdash B_1(c_1(c_2(x))) \prec_{e_2[x] \circ e_1[c_2(x)]} B_3(x)$$

So by the third rule in Figure 6.3.2, we have

$$\Gamma \vdash \Pi(A_1, B_1) <_{d_3} \Pi(A_3, B_3)$$

where

$$\begin{aligned} d_3 \;\; =_{df} \;\; & [f : \Pi(A_1, B_1)]\lambda(A_3, B_3, \\ & [x : A_3]e_2[x](e_1[c_2(x)](app(A_1, B_1, f, c_1(c_2(x)))))) \end{aligned}$$

Then

$$\begin{aligned} d_2 \circ d_1 &= [f : \Pi(A_1, B_1)]d_2(d_1(f)) \\ &= [f : \Pi(A_1, B_1)]d_2'(d_1'(f)) \\ &= [f : \Pi(A_1, B_1)]\lambda(A_3, B_3, \\ & \quad [x : A_3]e_2[x](app(A_2', B_2', d_1'(f), c_2(x)))) \\ &= [f : \Pi(A_1, B_1)]\lambda(A_3, B_3, \\ & \quad [x : A_3]e_2[x](e_1[c_2(x)](app(A_1, B_1, f, c_1(c_2(x)))))) \\ &= d_3 \end{aligned}$$

Finally, by the congruence rule, we have $\Gamma \vdash M_1 <_{d_2 \circ d_1} M_3$.

□

6.6. Algorithm for the coercion search in $T[\mathcal{R}\pi_1]_0$

Since we have proved the coherence and admissibility of substitution and transitivity for the system $T[\mathcal{R}\pi_1]_0$, we can give a sound and complete algorithm for the coercion search. If *the Coercion Search is decidable* in $\mathcal{C}$, it will also be decidable in $T[\mathcal{R}\pi_1]_0$.

6.6.1. Algorithm $ALG(\Gamma, M_1, M_2)$ for $T[\mathcal{R}\pi_1]_0$

Supposing the coercion search is decidable in $\mathcal{C}$, we give an algorithm $ALG(\Gamma, M_1, M_2)$ to check whether there is a judgement $\Gamma \vdash M_1 <_d M_2 : Type$ or $\Gamma \vdash M_1 \prec_d M_2 : Type$ when arbitrary Γ, M_1 and M_2 are given. If so, $ALG(\Gamma, M_1, M_2) := d'$ for some d' and $\Gamma \vdash d = d' : (M_1)M_2$, otherwise $ALG(\Gamma, M_1, M_2) :=\perp$.

The algorithm $ALG(\Gamma, M_1, M_2)$ will be mutually given with two other algorithms $Alg_1(\Gamma, M_1, M_2)$ and $Alg_2(\Gamma, M_1, M_2)$. The algorithm $Alg_1(\Gamma, M_1, M_2)$ will check if $\Gamma \vdash M_1 \prec_d M_2 : Type$ is derivable for

some d, while the algorithm $Alg_2(\Gamma, M_1, M_2)$ will check if
$\Gamma \vdash M_1 <_d M_2 : Type$ is derivable for some d.

- The algorithm $ALG(\Gamma, M_1, M_2)$:
  1. If Γ is valid context, M_1 and M_2 are well-typed then go to 2. Otherwise $ALG(\Gamma, M_1, M_2) :=\perp$.
  2. If $Alg_1(\Gamma, M_1, M_2) := d$ or $Alg_2(\Gamma, M_1, M_2) := d$ then $ALG(\Gamma, M_1, M_2) := d$. Otherwise $ALG(\Gamma, M_1, M_2) :=\perp$.
- The algorithm $Alg_1(\Gamma, M_1, M_2)$:
  1. Compute M_1 and M_2 to weak normal form $wnf(M_1)$ and $wnf(M_2)$. If $wnf(M_1)$ is a Σ-type, then go to 2. Otherwise, $Alg_1(\Gamma, M_1, M_2) :=\perp$.
  2. Suppose $wnf(M_1) \equiv \Sigma(A, B)$. If $\Gamma \vdash A = M_2$, then $Alg_1(\Gamma, M_1, M_2) := \pi_1(A, B)$. If $ALG(\Gamma, A, M_2) := c$ then $Alg_1(\Gamma, M_1, M_2) := c \circ \pi_1(A, B)$.
     Otherwise, $Alg_1(\Gamma, M_1, M_2) :=\perp$.
- The algorithm $Alg_2(\Gamma, M_1, M_2)$:
  1. If there is a judgement $\Gamma \vdash M_1 <_d M_2 \in \mathcal{C}$ then $Alg_2(\Gamma, M_1, M_2) := d$. Otherwise, go to 2.
  2. Compute M_1 and M_2 to weak normal form $wnf(M_1)$ and $wnf(M_2)$. If both $wnf(M_1)$ and $wnf(M_2)$ are Π-type or Σ-type then go to 3. Otherwise $Alg_2(\Gamma, M_1, M_2) :=\perp$.
  3. If $wnf(M_1) \equiv \Pi(A_1, B_1)$ and $wnf(M_2) \equiv \Pi(A_2, B_2)$ then go to 4. Otherwise $wnf(M_1) \equiv \Sigma(A_1, B_1)$ and $wnf(M_2) \equiv \Sigma(A_2, B_2)$ go to 5.
  4. If $\Gamma \vdash A_1 = A_2$ and $ALG((\Gamma, x : A_2), B_1(x), B_2(x)) := e[x]$ $(x \notin FV(\Gamma))$, then

     $$Alg_2(\Gamma, M_1, M_2) \;:=\; [f : \Pi(A_1, B_1)]\lambda(A_2, B_2, [x : A_1]e[x] \circ app(A_1, B_1, f, x))$$

     If $ALG(\Gamma, A_2, A_1) := c$ and $\Gamma, x : A_2 \vdash B_1(c(x)) = B_2(x)$, then

     $$Alg_2(\Gamma, M_1, M_2) \;:=\; [f : \Pi(A_1, B_1)]\lambda(A_2, B_2 \circ c, app(A_1, B_1, f) \circ c)$$

     If $ALG(\Gamma, A_2, A_1) := c$ and
     $ALG((\Gamma, x : A_2), B_1(c(x)), B_2(x)) := e[x]$, then

$$Alg_2(\Gamma, M_1, M_2) \quad := \quad [f : \Pi(A_1, B_1)]\lambda(A_2, B_2, [x : A_2]e[x](app(A_1, B_1, f, c(x))))$$

Otherwise $Alg_2(\Gamma, M_1, M_2) :=\perp$.

5. If $\Gamma \vdash A_1 = A_2$ and $ALG((\Gamma, x : A_2), B_1(x), B_2(x)) := e[x]$, then

$$Alg_2(\Gamma, M_1, M_2) \quad := \quad [x : \Sigma(A_1, B_1)]pair(A_2, B_2, \pi_1(A_1, B_1, x), e[\pi_1(A_1, B_1, x)](\pi_2(A_1, B_1, x)))$$

If $Alg_2(\Gamma, A_1, A_2) := c$ and $\Gamma, x : A_1 \vdash B_1(x) = B_2(c(x))$, then

$$Alg_2(\Gamma, M_1, M_2) \quad := \quad [x : \Sigma(A_1, B_1)]pair(A_2, B_2, c(\pi_1(A_1, B_1, x)), \pi_2(A_1, B_1, x))$$

If $Alg_2(\Gamma, A_1, A_2) := c$ and
$ALG((\Gamma, x : A_1), B_1(x), B_2(c(x))) := e[x]$, then

$$Alg_2(\Gamma, M_1, M_2) \quad := \quad [x : \Sigma(A_1, B_1)]pair(A_2, B_2, c(\pi_1(A_1, B_1, x)), e[\pi_1(A_1, B_1, x)](\pi_2(A_1, B_1, x)))$$

Otherwise $Alg_2(\Gamma, M_1, M_2) :=\perp$.

6.6.2. Soundness and Completeness

In order to prove the soundness and completeness of the above algorithm, we first need to prove some lemmas.

Lemma 6.6.1

- *If* $\Gamma \vdash M_1 \prec_d M_2 : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$*, then* M_1 *is computationally equal to a* Σ*-type.*
- *If* $\Gamma \vdash \Sigma(A, B) \prec_d A' : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$*, then one of the following judgements is derivable in* $T[\mathcal{R}\pi_1]_0$*:*
  - · $\Gamma \vdash A = A' : Type$*; or*
  - · $\Gamma \vdash A <_c A' : Type$ *for some* c*; or*
  - · $\Gamma \vdash A \prec_c A' : Type$ *for some* c*.*

Proof. By induction on derivations. □

Lemma 6.6.2 *If* $\Gamma \vdash M_1 <_d M_2 : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$*, then one of the following holds:*

- $\Gamma \vdash M_1 <_d M_2 : Type \in \mathcal{C}$*; or*
- *Both* M_1 *and* M_2 *are computationally equal to* Π*-types; or*
- *Both* M_1 *and* M_2 *are computationally equal to* Σ*-types.*

Proof. By induction on derivations. □

Lemma 6.6.3

- *If* $\Gamma \vdash \Pi(A, B) <_d \Pi(A', B') : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$*, then one of the following holds in* $T[\mathcal{R}\pi_1]_0$*:*
  - · $\Gamma \vdash A = A' : Type$ *and* $\Gamma, x : A \vdash B(x) \propto_{e[x]} B'(x) : Type$ *for some* e*; or*
  - · $\Gamma \vdash A' \propto_c A : Type$ *for some* c *and* $\Gamma, x : A' \vdash B(c(x)) = B'(x) : Type$*; or*
  - · $\Gamma \vdash A' \propto_c A : Type$ *for some* c *and* $\Gamma, x : A' \vdash B(c(x)) \propto_{e[x]} B'(x) : Type$ *for some* e*.*
- *If* $\Gamma \vdash \Sigma(A, B) <_d \Sigma(A', B') : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$*, then one of the following holds in* $T[\mathcal{R}\pi_1]_0$*:*
  - · $\Gamma \vdash A = A' : Type$ *and* $\Gamma, x : A \vdash B(x) \propto_{e[x]} B'(x) : Type$ *for some* e*; or*
  - · $\Gamma \vdash A <_c A' : Type$ *for some* c *and* $\Gamma, x : A \vdash B(x) = B'(c(x)) : Type$*; or*
  - · $\Gamma \vdash A <_c A' : Type$ *for some* c *and* $\Gamma, x : A \vdash B(x) \propto_{e[x]} B'(c(x)) : Type$ *for some* e*.*

Proof. By induction on derivations. □

Theorem 6.6.4 ***(Soundness)*** *If* $ALG(\Gamma, M_1, M_2) :=\perp$ *then neither* $\Gamma \vdash M_1 <_d M_2 : Type$ *nor* $\Gamma \vdash M_1 \prec_d M_2 : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$ *for any* d*. If* $ALG(\Gamma, M_1, M_2) := d$ *then either* $\Gamma \vdash M_1 <_d M_2 : Type$ *or* $\Gamma \vdash M_1 \prec_d M_2 : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$*.*

Proof. By Lemma 6.6.2 and 6.6.3. □

Theorem 6.6.5 ***(Completeness)*** *If* $\Gamma \vdash M_1 <_d M_2 : Type$ *or* $\Gamma \vdash M_1 \prec_d M_2 : Type$ *is derivable in* $T[\mathcal{R}\pi_1]_0$*, then there is a* d' *such that* $ALG(\Gamma, M_1, M_2) := d'$ *and* $\Gamma \vdash d = d' : (M_1)M_2$*.*

Proof. By Lemma 6.6.2 and 6.6.3. □

6.6.3. Decidability of the Coercion Search in $T[\mathcal{R}\pi_1]_0$

Theorem 6.6.6 *If the coercion search is decidable in $\mathcal{C}$, it is also decidable in $T[\mathcal{R}\pi_1]_0$,* i.e. *it is decidable whether there is a judgement $\Gamma \vdash M_1 <_d M_2 : Type$ or $\Gamma \vdash M_1 \prec_d M_2 : Type$ is derivable in $T[\mathcal{R}\pi_1]_0$ for arbitrary Γ, M_1 and M_2.*

Proof. By Theorem 6.6.4 and Theorem 6.6.5. □

6.7. Discussions

6.7.1. Side conditions

In order to block the unwanted derivations, one may still try to keep the rule $\pi_1 rule$ in section 6.1 and use side conditions for the First Component rule, without introducing any new subtyping relation. For instance, one of such side conditions for the First Component rule is the following.

$$\frac{\Gamma \vdash A <_c A' : Type \quad \Gamma \vdash B : (A')Type}{\Gamma \vdash \Sigma(A, B \circ c) <_{d_1} \Sigma(A', B) : Type} \quad (size(A) = size(A'))$$

or

$$\frac{\Gamma \vdash A <_c A' : Type \quad \Gamma \vdash B : (A')Type}{\Gamma \vdash \Sigma(A, B \circ c) <_{d_1} \Sigma(A', B) : Type} \quad (size(A) \not> size(A'))$$

In $T[\mathcal{R}\pi_1]_0$, *size* is well-defined. Similarly, *size* can be defined in $T[\mathcal{R}]_0$ and one can prove its well-definedness (see Section 3.2 for more details of $T[\mathcal{R}]_0$ and $T[\mathcal{R}]$. Here, $\mathcal{R}$ includes one of the above rules). It is obvious that $T[\mathcal{R}\pi_1]_0$ and $T[\mathcal{R}]_0$ are equivalent in terms of the following lemma.

Lemma 6.7.1 *If $\Gamma \vdash A \propto_c B : Type$ is derivable in $T[\mathcal{R}\pi_1]_0$ then $\Gamma \vdash A <_c B : Type$ is derivable in $T[\mathcal{R}]_0$ and vice versa.*

However, since the system $T[\mathcal{R}]$ includes the coercive definition rule and the coercive application rules in Figure 3.2.1, A and A' in the side-condition may not be well-typed in the original type theory any more. The way to compute such terms is to insert coercions first and then do the usual computation in the original type theory. So the property that inserting coercion is decidable in $T[\mathcal{R}]$ must be proved first in order to argue the well-definedness of *size*. There is a circularity, that is, a property of $T[\mathcal{R}]$ is needed in order to present $T[\mathcal{R}]$ itself.

6.7.2. New computation rules

In Section 6.3, we have given new subtyping rules for Σ-types and Π-types which have only one constructor. Since the coercions can be defined by using the function operators π_1 and π_2 and *app*, we are able to prove the admissibility of transitivity. However, we need to be careful to introduce the subtyping rules for other inductive types. For example, if we want to introduce new subtyping rules for lists as follows.

$$\frac{\Gamma \vdash A \propto_c B : Type}{\Gamma \vdash List(A) <_d List(B) : Type}$$

where $d = map_{List}(A, B, c)$, we also need to add the new computation rule for lists into the system $T[\mathcal{R}\pi_1]_0$ (see the new computation rule in Section 5.8.1). The reason for doing this is the same as studied in Section 5.1 and Section 5.8. After adding new computation rules, we are able to combine the natural subtyping rules for the parameterised inductive data types, Σ-types and lists, and the normal transitivity rule for subtyping is admissible.

6.7.3. Combining incoherent coercions in general

We have studied in this chapter a special case of incoherent coercions. However, when we consider combining incoherent coercions in general, we must be sensible, that is, we don't try to combine any arbitrary incoherent coercions. For example, suppose that there are two different coercions c_1 and c_2 from type A to B. A sensible thing to do is to use only one at a time; if we want to use c_1 as a coercion, then c_2 must be switched off, and vice versa.

CHAPTER 7

Conclusion

7.1. Summary

This thesis is the first study of the issue of coherence and transitivity at type level in coercive subtyping. We focus on the coercions between parameterised inductive data types. A number of examples are given in this thesis to identify the serious problems with these coercions concerning coherence and transitivity. The thesis provides not only the proofs but also clearer understanding of the subtyping rules for parameterised inductive data types.

We choose two examples, Σ-types and Π-types, as representatives of the parameterised inductive data types that have only one constructor (*i.e. ST-form*) to demonstrate that coercions for such types can be defined by using their special function operators. Since coercions are defined in this way, we proved the coherence and the admissibility of the normal transitivity rule.

Through a close examination of some key examples we get a better understanding of the coercions between parameterised inductive data types in general. For many parameterised inductive data types such as lists, coercions have to be defined inductively and the normal transitivity rule is not admissible. However, a large class of inductive data types with their subtyping rules is suitable for weak transitivity. In every such subtyping rule, there is no coercion dependency that may occur; that is, the coercion in one premise doesn't appear in another premise. We also prove that the meta-level equality requirement is satisfied. If $A <_{c_1} B$, $B <_{c_2} C$ and $A <_{c_3} C$ then c_3 is extensionally equal to $c_2 \circ c_1$.

A counter example shows that the component-wise subtyping rules for Σ-types and the subtyping rule of its first projection are incoherent if they are put together directly. We introduce a new subtyping relation and give a new formulation of coercive subtyping. In particular,

coherence and transitivity are redefined. This new formulation satisfies the new coherence requirements and enjoys the admissibility of the new transitivity rules.

7.2. Implementation

As we mentioned in Section 1.8, coercion mechanisms of non-dependent coercions with certain restrictions have been implemented in both the proof development systems Lego and Coq, by Bailey and Saïbi, respectively. A mixture of simple coercions, parameterised coercions, coercion rules and dependent coercions has been implemented in Plastic by Callaghan.

I also implemented *logical framework* and *inductive data types.* As mentioned in Remark 2.3.8, the elimination operators and computation rules are implemented differently. For logical framework, *Terms* and *Kinds* are represented by mutually recursive data types so that as many as possible ill-typed terms are not representable. In Haskell, they look like the following.

$$\begin{array}{lll} data \quad Term & = & Var \;\; String \\ & \mid & Lam \;\; String \;\; Kind \;\; Term \\ & \mid & App \;\; Term \;\; Term \end{array}$$

$$\begin{array}{lll} date \quad Kind & = & Type \\ & \mid & El \;\; Term \\ & \mid & Prod \;\; String \;\; Kind \;\; Kind \end{array}$$

Chapter 6 is the first study on how to combine the component-wise subtyping rules for Σ-types and the subtyping rule of its first projection. A sound and complete algorithm for the coercion search is also given in Section 6.6. Based on my implementation of logical framework and inductive data types, I also implemented *coercive subtyping,* especially the component-wise subtyping rules for Σ-types and the subtyping rule of its first projection. These two sets of subtyping rules can be used in a single system. The algorithm is on page 125.

7.3. Future work

As we discussed in Section 5.8, there are problems concerning coherence and transitivity for the subtyping rules of parameterised inductive data types. In particular, the problem regards the combination of these subtyping rules. As we showed in Example 5.2.3, neither the strong (or normal) transitivity rule nor the weak transitivity rule can be admissible when we combine the subtyping rules for Σ-types and lists. This leads us to fundamental future work on the extension of type theory. By adding new computation rules for parameterised inductive data types so the natural subtyping rules for all the parameterised inductive data types can be uniformly used together.

The meta-properties of these new computation rules such as *Strong Normalisation* and *Church-Rosser* need further study. Although such meta-properties should intuitively be true, proving them is not easy and likely to be a huge task as proving them in UTT [Gog94]. Even the weak normalisation (*i.e.* There is a finite computation sequence for every well-typed term) is hard to prove, mainly because new redexes may be created after applying the new computation rules. In the following, I give an example to illustrate the increasing of redexes.

Example 7.3.1 *Consider the terms* $d_2 \circ d_1 \equiv [l_0 : List(List(N))]d_2(d_1(l_0))$ *and* $d_3 \circ d_2 \equiv [l_0 : List(List(N))]d_3(d_2(l_0))$ *where*

$$d_1 \equiv map_{List}(List(N), List(N), c_1)$$

$$c_1 \equiv [l : List(N)]nil(N)$$

$$d_2 \equiv map_{List}(List(N), List(N), c_2)$$

$$c_2 \equiv map_{List}(N, N, [n : N]n)$$

$$d_3 \equiv map_{List}(List(N), List(N), c_3)$$

$$c_3 \equiv map_{List}(N, N, [n : N]O)$$

and $List(N)$ *(the type of the lists of natural numbers) and* map_{List} *can be found on page 32. Note that* d_1, d_2 *and* d_3 *are normal forms.*

Now, by the new computation rule for lists on page 104, we compute $d_2 \circ d_1$ *and* $d_3 \circ d_2$ *as follows:*

$$d_2 \circ d_1 \Rightarrow map_{List}(List(N), List(N), c_2 \circ c_1)$$

$$d_3 \circ d_2 \Rightarrow map_{List}(List(N), List(N), c_3 \circ c_2)$$

where

$$c_2 \circ c_1 = [l : List(N)]map_{List}(N, N, [n : N]n, nil(N))$$
$$c_3 \circ c_2 = map_{List}(N, N, [n : N]O) \circ map_{List}(N, N, [n : N]n)$$

According to the first computation rule for lists, $c_2 \circ c_1$ *has a new redex and can be reduced to* $[l : List(N)]nil(N)$ *and; according to the new reduction rule for lists,* $c_3 \circ c_2$ *has a new redex and can be reduced to* $map_{List}(N, N, [n : N]O)$.

So, new redexes may be created after applying the new computation rules for parameterised inductive data types.

Another interesting area for future work is to consider coercive subtyping in the framework of extensional type theories. Although type checking in extensional type theories is undecidable, studying coercive subtyping and its related issues in an extensional framework may provide further theoretical insights. Some fundamental difficulties in extensional type theories need to be overcome first in order to study coercive subtyping. For example, in an extensional type theory, can we prove that $List(A) = List(B)$ implies $A = B$? One promising suggestion is to consider elimination rule for universes. Yet how such direction affects the formulation of coercive subtyping is still open.

Bibliography

[B$^+$00] B. Barras et al. *The Coq Proof Assistant Reference Manual (Version 6.3.1)*. INRIA-Rocquencourt, 2000.

[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-88-52, Dept. of Computer Science, University of Edinburgh, 1988.

[Bai98] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory*. PhD thesis, University of Manchester, 1998.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, 1992.

[BCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. *Information and Computation*, 93, 1991.

[Bee85] M. J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[BF99] G. Barthe and M. J. Frade. Constructor subtyping. *Proceedings of ESOP'99, LNCS 1576*, 1999.

[BM92] R. Burstall and J. McKinna. Deliverables: a categorical approach to program development in type theory. LFCS report ECS-LFCS-92-242, Dept of Computer Science, University of Edinburgh, 1992.

[BT98] G. Betarte and A. Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, pages 21–39. Oxford Science Publications, 1998.

[Bur93] R. Burstall. Extended Calculus of Constructions as a specification language. In R. Bird and C. Morgan, editors, *Mathematics of Program Construction*, 1993. Invited talk.

[BvR00] G. Barthe and F. van Raamsdonk. Constructor subtyping in the calculus of inductive constructions. *Proceedings of FOSSACS'00, LNCS 1784*, 2000.

[C+86] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Pretice-Hall, 1986.

[Cal99] P. C. Callaghan. An implementation of typed LF with coercions. The Annual Meeting of TYPES, 1999.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland Publishing Company, 1958.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency–Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[Che98] G. Chen. *Subtyping, Type Conversion and Transitivity Elimination.* PhD thesis, University of Paris VII, 1998.

[Chu40] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(1), 1940.

[CL99] P. Callaghan and Z. Luo. Implementation techniques for inductive types. *Proceedings of TYPES'99*, 1999.

[CL01] P. Callaghan and Z. Luo. An implementation of LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

[CLP01] P. C. Callaghan, Z. Luo, and J. Pang. Object languages in a type-theoretic meta-framework. *Workshop of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, 2001.

[CMMS91] Luca Cardelli, John C. Mitchell, Simone Martini, and Andre Scedrov. An extension of system F with subtyping. In Takayasu Ito and Albert R. Meyer, editors, *Proc. of 1st Int. Symp. on Theor. Aspects of Computer Software, TACS'91, Sendai, Japan, 24–27 Sept 1991*, volume 526, pages 750–770. Springer-Verlag, Berlin, 1991.

[Coq92] Th. Coquand. Pattern matching with dependent types. Talk given at the BRA workshop on Proofs and Types, Bastad, 1992.

[CPM90] Th. Coquand and Ch. Paulin-Mohring. Inductively defined types. *Lecture Notes in Computer Science*, 417, 1990.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 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. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift*, 39, 1935.

[GM93] M. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher-order logic.* Cambridge University Press, 1993.

[Gog94] H. Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, 1994.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Proc. 2nd Ann. Symp. on Logic in Computer Science. IEEE*, 1987.

[HHP92] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of ACM*, 40(1):143–184, 1992.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic.* Academic Press, 1980.

[JLS98] A. Jones, Z. Luo, and S. Soloviev. Some proof-theoretic and algorithmic aspects of coercive subtyping. *Types for proofs and programs (eds, E. Gimenez and C. Paulin-Mohring), Proc. of the Inter. Conf. TYPES'96, LNCS 1512*, 1998.

[LL01] Y. Luo and Z. Luo. Coherence and transitivity in coercive subtyping. In R. Nieuwenhuis and A. Voronkov, editors, *8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2250 of *Lecture*

Notes in Artificial Intelligence, pages 249–265. Springer-Verlag, 2001.

[LL04a] Y. Luo and Z. Luo. Combining incoherent coercions for Σ-types. In *Proceedings of TYPES 2003*, volume 3085. Lecture Notes in Computer Science, 2004.

[LL04b] Z. Luo and Y. Luo. Transitivity in coercive subtyping. *Journal of Information and Computation*, 2004. To appear.

[LLS02] Y. Luo, Z. Luo, and S. Soloviev. Weak transitivity in coercive subtyping. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 220–239. Springer-Verlag, 2002.

[LMS95] G. Longo, K. Milsted, and S. Soloviev. A logic of subtyping. In *Proc. of LICS'95*, 1995.

[LMS00] G. Longo, K. Milsted, and S. Soloviev. Coherence and transitivity of subtyping as entailment. *Journal of Logic and Computation*, 10:493–526, 2000.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[LPT89] Z. Luo, R. Pollack, and P. Taylor. *How to Use LEGO: a preliminary user's manual.* LFCS Technical Notes LFCS-TN-27, Dept. of Computer Science, Edinburgh University, 1989.

[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.

[Luo90] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990. Also as Report CST-65-90/ECS-LFCS-90-118, Department of Computer Science, University of Edinburgh.

[Luo93] Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3(3), 1993.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[Luo99] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[Luo03] Z. Luo. PAL$^+$: a lambda-free logical framework. *Journal of Functional Programming*, 13(2):317–338, 2003.

[Mit91] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(2):245–285, 1991.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: a prototype verification system. In Deepar Kapur, editor, *11th International Conference on Automted Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[Pau93] L. Paulson. Introduction to Isabelle. Technical Report 280, Computer Laboratory, Cambridge University, 1993.

[PM93] C. Paulin-Mohring. Inductive definitions in the system Coq: rules and properties. *Proceedings of the Inter. Conf. on Typed Lambda Calculi and Applications (TLCA'93), LNCS 664*, 1993.

[Pol94] R. Pollack. *The Theory of LEGO: a proof checker for the Extended Calculus of Constructions.* PhD thesis, Edinburgh University, 1994.

[Pol97] Erik Poll. Subtyping and inheritance for inductive types (extended abstract). In *Electronic Proceedings TYPES WG Workshop on Subtyping, Inheritance and Modular Development of Proofs, Durham, UK, 30 Aug – 1 Sept 1997.* Dept. of Computer Science, Univ. of Durham, 1997.

[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. On the idea of a general proof theory. *Synthese*, 27, 1974.

[Reh96] Jakob Rehof. Strong normalization for non-structural subtyping via saturated sets. *Information Processing Letters*, 58(4):157–162, 1996.

[Sai97] A. Saibi. Typing algorithm in type theory with inheritance. *Proc of POPL'97*, 1997.

[Sch97] Thomas Schreiber. Auxiliary variables and recursive procedures. *TAPSOFT'97: Theory and Practice of Software Development, LNCS 1214*, 1997.

[Set93] A. Setzer. *Proof theoretical strength of Martin-Löf's type theory with W-type and one universe.* PhD thesis, Universität München, 1993.

[Set04] Anton Setzer. Proof theory of Martin-Löf type theory – an overview. *Mathematiques et Sciences Humaines*, 42 année, n°165:59 – 99, 2004.

[SL02] S. Soloviev and Z. Luo. Coercion completion and conservativity in coercive subtyping. Annals of Pure and Applied Logic, 2002.

[Smi84] Jan M. Smith. An interpretation of Martin-Löf's type theory in a type-free theory of proposition. *Journal of Symbolic Logic*, 49, 1984.

[Tas97] A. Tasistro. *Substitution, record types and subtyping in type theory.* PhD thesis, Chalmers University of Technology, 1997.

[Tho91] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[Tho99] Simon Thompson. *Haskell : the craft of functional programming.* Harlow : Addison Wesley, 2nd edition, 1999.

[YL97] S. Yu and Z. Luo. Implementing a model checker for Lego. *Proc. of the 4th Inter Symp. of Formal Methods Europe, FME'97: Industrial Applications and Strengthened Foundations of Formal Methods, Graz, Austria. LNCS 1313*, 1997.

Index

<, 45
App, 27
Eq, 28
Prop, 27
$T[\mathcal{R}\pi_1]$, 112
$T[\mathcal{R}\pi_1]_0$, 109
$T[\mathcal{R}]$, 47
$T[\mathcal{R}]_0$, 46
π_1, 39
π_2, 39
$\prec$, 109
app, 38
map_{List}, 33
minus, 32
plus, 32
pred, 32
times, 32

addition, 32
Admissibility of substitution, 61, 99, 118
Admissibility of transitivity, 61, 118
Admissibility of weak transitivity, 99
Algorithm for the coercion search, 67, 125

Church-Rosser, 41
Coercion dependency, 77
Coercive definition rule, 45, 109
Coercive subtyping, 44
Coherence, 47, 57, 96, 111
Computational equality, 26
Congruence rule, 46
Consistency, 43
Context equality, 25
Covariance and Contravariance, 84

Decidability, 69, 129
Definitional equality, 26
Depth, 120
depth, 62

Extended WT-schema, 102
Extensional equality, 42
Extensional equality requirement, 101
Extensional type theory, 42

Formal verification, 8
Function subtyping, 17

Inductive data types, 29
  Maybe types, 81
  Π-types, 37
  Σ-types, 38

Binary trees, 34
Disjoint union *Either*, 81
Function types, 33
Lists, 32
Natural numbers, 25, 32
type of dependent function spaces, 37
type of dependent pairs, 38
type of non-dependent trio, 39
Vectors, 34
Inductive schemata, 29
Inference rules in $T[\mathcal{R}]$, 47
Inference rules of LF, 24

Judgements in LF, 22

Kinds, 23

Logical Framework, 22
Logical operators
&, 28
$\exists$, 28
$\neg$, 28
$\supset$, 28
$\vee$, 28
false, 28
true, 28

Meta-level equality requirement, 75
Model-checking, 8
multiplication, 32

New computation rules, 102
for lists, 104
New substitution rules, 118
New subtyping relation, 109
New subtyping rules
for Π-types, 112
for Σ-types, 112
for lists, 130
for the first projection, 110
New transitivity rules, 118

Parameterised inductive data types, 31
predecessor, 32
Product subtyping, 17
Projection operators, 39
Proof, 29
Proof of coherence, 59, 97, 116
Proof of transitivity, 64, 122
Proof of Weak transitivity, 99
Propositional equality, 28
Provable, 29

Record subtyping, 17

size, 115
Small kinds, 23
SOL, 27
Soundness and Completeness, 69, 127
Specify type theories, 25
ST-form, 35
Strictly positive operator, 29
Strong Normalisation, 41
Substitution rule, 46
subsumption rule, 17
subtraction, 32
Subtyping rules
for Π-types, 51
for Σ-types, 51
for *Either* types, 83
for *Maybe* types, 82

for Binary trees, 83
for function types, 84
for lists, 74, 82
for ST-form, 69
for vectors, 101
for WT-schemata, 82
WDC rule, 51
Syntactical equality, 27

Transitivity rule, 46
Types, 23

Universes, 42
UTT, 22

WDC, 50
WDC rule, 51
Weak transitivity, 73, 75
Weak transitivity rule ($WTrans$), 75
Weak transitivity schemata, 79
Weakening, 62, 119
Well-defined coercions, 50
WT small kind, 80
WT strictly positive operator, 80
WT-schema, 80