Transitivity in Coercive Subtyping ⋆

Zhaohui Luo and Yong Luo

Department of Computer Science
Royal Holloway, University of London
Egham, Surrey TW20 0EX, U.K.
E-mail: {Zhaohui.Luo,Yong.Luo}@rhul.ac.uk

Abstract

Coercive subtyping is a general approach to abbreviation and subtyping in dependent type theories with inductive types. Coherence and admissibility of transitivity are important both for understanding of the framework and for its correct implementation. In this paper, we study the issue of transitivity in the context of subtyping for parameterised inductive types. In particular, we propose and study the notion of weak transitivity and show that, for a large class of parameterised inductive types, the natural subtyping rules are coherent and weak transitivity is admissible in an intensional type theory. A possible extension of type theory with certain extensional computation rules is also discussed for achieving admissibility of transitivity in general.

1 Introduction

Coercive subtyping has been studied as a promising general approach to abbreviation and subtyping in dependent type theories with inductive types (see, for example, [16,17]). It has been implemented in several proof assistants such as Coq [3], Lego [19] and Plastic [7] and used effectively in proof development (e.g., [2]).

In coercive subtyping, the subtyping relation between any two types is associated with a coercion between them. A is a subtype of B if there is a (unique) coercion c from A to B, where c is a functional operation from A to B in the type theory. Therefore, any object of type A may be regarded as (an abbreviation of) an object of type B via coercion c. Note that this is different from

⋆ This work is partially supported by the UK EPSRC grants GR/R84092 and GR/R72259 and the EU TYPES grant 21900.

the more traditional understanding of subtyping via subsumption. (See the next section for more on this together with a brief introduction to coercive subtyping.)

Coherence and admissibility of transitivity are crucial properties of any coercive subtyping system. Coherence essentially says that coercions between any two types are unique, while admissibility of transitivity (or transitivity elimination) is obviously important as for any subtyping system. Besides ensuring the logical correctness of a coercive subtyping system, these properties are also the basis for a correct implementation.

This paper studies transitivity and coherence for parameterised inductive types and the associated subtyping rules. We propose and study the notion of weak transitivity, to be explained below, and show that, for a large class of parameterised inductive types, the natural subtyping rules are coherent and weak transitivity is admissible in an intensional type theory.

A problem with transitivity In the presentation of coercive subtyping in [17], the following transitivity rule is included:

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

Intuitively, it says that the composition of two coercions is also a coercion, the coercion corresponding to transitivity. In [13], it has been proved that the transitivity rule is admissible for certain subtyping rules, such as those for Π-types and Σ-types.

However, the above transitivity rule is sometimes too strong (in intensional type theories). For some parameterised inductive data types together with natural subtyping rules, especially when the inductive type has more than one constructor, the above rule fails to be admissible or eliminatable. For instance, for the inductive type of lists $List(A)$ parameterised by its element type A, if we introduce the following subtyping rule, where $map(A, B, c)$ is the application of the usual map function to the coercion function c such that $map(A, B, c)(nil(A)) = nil(B)$ and $map(A, B, c)(cons(A, a, l)) = cons(B, c(a), map(A, B, c)(l))$:

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

then the transitivity rule $(Trans)$ fails to be admissible. If we add $(Trans)$ together with the above rule to the system, coherence fails. To see this, suppose that we have types F, E and N such that $F <_{c_1} E$ and $E <_{c_2} N$. Then, by

$(Trans)$, $F <_{c_2 \circ c_1} N$. By the above subtyping rule for lists, we have

$$List(F) <_{map(F,E,c_1)} List(E),$$

$$List(E) <_{map(E,N,c_2)} List(N),$$
$$List(F) <_{map(F,N,c_2 \circ c_1)} List(N).$$

By the transitivity rule $(Trans)$, we also have

$$List(F) <_{map(E,N,c_2) \circ map(F,E,c_1)} List(N).$$

Now, the problem is that $map(F, N, c_2 \circ c_1)$ is not computationally equal to $map(E, N, c_2) \circ map(F, E, c_1)$ in an intensional type theory, although we know that they are extensionally equal. In other words, we have two coercions between $List(F)$ and $List(N)$ which are not computationally equal and hence coherence fails.

Remark 1.1 *The problem shown in the above example arises when we consider subtyping rules for parameterised inductive types. This itself is a difficult issue, but these subtyping rules are powerful and useful.*

Weak transitivity – a proposed solution Rather than the above (strong) transitivity rule $(Trans)$, we introduce a weaker notion of transitivity – *weak transitivity,* which can informally be represented by the following rule:

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

It says that, if $A <_c B$ and $B <_{c'} C$, then $A <_{c''} C$ for some coercion c''. Furthermore, we require that c'' be extensionally equal to $c' \circ c$ (see Section 4.2 for the treatment of the equality requirement). The essential difference compared with the strong transitivity rule $(Trans)$ is that we are only concerned with the existence of a coercion that is extensionally equal to the composition. Weak transitivity expresses the adequate requirement enough for the application of coercive subtyping. For many natural subtyping rules for parameterised inductive types, weak transitivity is admissible, as shown in this paper, but the strong transitivity rule $(Trans)$ is not.

WT-schemata Through our investigation, we have also found out that weak transitivity does not hold for some parameterised inductive types such as dependent Σ-types which involve certain form of dependency between parameters. We show how such dependency can be made precise – we consider, in Section 3, a restricted form of inductive schemata – WT-schemata, which forbid such dependency and the inductive types generated from which enjoy weak transitivity. (Note that the harmful dependency is some form of dependency between parameters. Forbidding such dependency does not mean that all of the dependent types are excluded. See more details in Section 3 and Section 6.) For such inductive schemata, we develop a general method, which is also useful for implementation, to give the subtyping rules and the definition of coercions for a large class of parameterised inductive types. Then, in Section 4.1, we prove the coherence and the admissibility of weak transitivity (and substitution) of the coercive subtyping system with these rules. Section 4.2 discusses the equality requirement for weak transitivity and shows that the general coercions we define satisfy the extensional equality requirement.

Extensional computation rules As seen in the above example of lists and later, the problem arises from the fact that certain extensional equalities do not hold computationally in intensional type theories. For example, it is not the case that $map(F, N, c_2 \circ c_1)$ and $map(E, N, c_2) \circ map(F, E, c_1)$ are computationally equal. It is shown that, if we consider such a restricted form of extensional equality rules as computational, inductive types generated by any inductive schema, including those excluded by the WT-schemata, can be associated with natural subtyping rules which are proved to be coherent and satisfying transitivity elimination. Discussions are given in Section 5 whether such equalities can or should be regarded as computational.

2 Coercive subtyping

We give in this section a brief introduction to coercive subtyping and explain some background notations to be used in latter sections.

2.1 Coercive subtyping: the basic idea and overview

Two views about types and subtyping In the literature, there have been two different views of types and consequently, two related but different views about subtyping. The more traditional view, as often found in the study of programming languages, is that of type assignment. Under this view, types are assigned to objects, which already exist. Considered in this way, it is natural to think that an object may have more than one type and a type A is a subtype of type B if all objects of A are also objects of B. This is the typical view to understand the notion of subtyping by means of the so-called subsumption rule.

The other view considers the relationship between types and objects in a different way; we call this view as that of canonical objects, which is well accepted in the community of dependent type theories (c.f., Martin-Löf's type theory

[21]). Under this view, types are considered as consisting of their canonical objects and the objects and their types depend on each other and do not exist independently. For example, the type N of natural numbers consists of the canonical numbers 0 and $succ(n)$ and the natural numbers only exist because they are objects of N. This view of canonical objects is the basis to consider various inductive types in type theory, where there are associated reasoning principles (elimination rules) expressing that, in order to prove a property for all objects of an inductive type, one only has to prove it for all of its canonical objects.

If types are considered as consisting of their canonical objects, it is difficult to see how subtyping could be understood or introduced by means of subsumption. To do so, one would have to ask questions like: Would the canonical objects of a subtype also be canonical objects of a supertype? How would reasoning principles be formulated to take care of the objects introduced by subtyping relations? Obviously, such thinking leads to difficulties.

Fortunately, the notion of subtyping can be understood in a different way, by means of the existence of coercions: A is a subtype of B if there is a (unique) coercion from A to B, where a coercion is a special function from A to B. Coercive subtyping is based on the view of canonical objects and employs coercions in establishing subtyping relations.

Remark 2.1 *The notion of coercion has been studied in the literature for simple type systems and for the simpler type systems, one can show that subsumption and coercion are equivalent (see, for example, [23,22]). However, when more sophisticated types (e.g., various inductive types) are considered, such an equivalence does not hold anymore and it is difficult to see how the view of type assignment and the notion of subsumption can be used to introduce subtyping.*

Coercive subtyping In coercive subtyping, a type is a subtype of another type if there is a unique coercion between them. A coercion plays the role of abbreviation. More precisely, if c is a coercion from K_0 to K, then a functional operation f from K to K' can be applied to any object k_0 of K_0 and the application $f(k_0)$ is definitionally equal to $f(c(k_0))$. Intuitively, we can view f as a context which requires an object of K; then the argument k_0 in the context f stands for its image of the coercion, $c(k_0)$. Therefore, one can use $f(k_0)$ as an abbreviation of $f(c(k_0))$.

Remark 2.2 *Note that this is different from the traditional view of subtyping where, as explained above, subsumption is the central idea by which supertypes contain also those objects of their subtypes. In coercive subtyping, types do not obtain more objects through subtyping. Although $f(k_0)$ is now a well-typed* *object of K', it abbreviates $f(c(k_0))$ which is already an object of K'. Another way to look at this issue is that subsumption is based on "overloading terms", that is, a term (typically a λ-term) resides in its type and the supertypes of its type. In coercive subtyping, we do not have such overloading – a term such as $f(k_0)$ may inhabit K' only because that it abbreviates the object $f(c(k_0))$, an object of K'.*

The above simple idea becomes very powerful when formulated in the logical framework. Various useful mechanisms of coercion can be represented [2] and they are very useful in the practice of proof development. The framework of coercive subtyping covers a variety of subtyping relations including those represented by parameterised coercions and coercions between parameterised inductive types. For example, see [17,2,7,18] for details of some of these development and applications of coercive subtyping.

Some important meta-theoretic aspects of coercive subtyping have been studied. In particular, the results on conservativity and on transitivity elimination for subkinding have been proved in [27]. 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 [27] is essentially that coherence of basic subtyping rules does imply conservativity. These results not only justify the adequacy of the theory from the proof-theoretic consideration, but also provide the proof-theoretic basis for implementation of coercive subtyping.

Coercion mechanisms with certain restrictions have been implemented both in the proof development system Lego [19] and Coq [3], by Bailey [2] and Saibi [26], respectively. Callaghan of the Computer Assisted Reasoning Group at Durham has implemented Plastic [7], a proof assistant that supports logical framework and coercive subtyping with a mixture of simple coercions, parameterised coercions, coercion rules for parameterised inductive types, and dependent coercions [20].

Related work The early development of the framework of coercive subtyping is closely related to Aczel's idea in type-checking overloading methods for classes [1] and the work on giving coercion semantics to lambda calculi with subtyping by Breazu-Tannen et al [6]. Bailey, Saibi, and Callaghan's respective implementations of coercions in the proof systems Lego, Coq and Plastic are important contributions [2,26,7]. Barthe and his colleagues have studied constructor subtyping and its possible applications in proof systems [4,5]. A logical approach to the study of subtyping in system F can be found in [11] and Chen has studied the issue of transitivity elimination in that framework

[8]. One of Chen's proof methods was used in one of our earlier papers [13] to prove the admissibility of transitivity in the framework of coercive subtyping. This paper is a further development of [14].

2.2 Coercive subtyping: a formal presentation

Coercive subtyping [17] is formally formulated as an extension of (type theories specified in) the logical framework LF [15], whose rules are given in Appendix A. (The LF here is different from the Edinburgh Logical Framework [10].) Types in LF are called kinds. The kind $Type$ represents the conceptual universe of types and a kind of the form $(x : K)K'$ represents the dependent product with functional operations f as objects (*e.g.*, abstraction $[x : K]k'$) which can be applied to objects of kind K to form application $f(k)$. For every type (an object of kind $Type$), $El(A)$ is the kind of objects of A. A kind is small if it does not contain $Type$. LF can be used to specify type theories, such as Martin-Löf's type theory [24] and UTT [15].

Notation 2.3 *We shall use the following notations:*

- *We often write $(K)K'$ for $(x : K)K'$ when x does not occur free in K', and A for $El(A)$ and hence $(A)B$ for $(El(A))El(B)$ when no confusion may occur.*
- *Substitution: 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.*
- *Functional composition: for $f{:}\,(K_1)K_2$ and $g{:}\,(K_2)K_3$, define $g \circ f =_{df} [x : K_1]g(f(x)){:}\,(K_1)K_3$, where x does not occur free in f or g.*

A system with coercive subtyping, $T[R]$, is an extension of any type theory T specified in LF. It can be presented in two stages: first we consider the system $T[R]_0$, which is an extension of T, with subtyping judgements of the form $\Gamma \vdash A <_c B{:}\,Type$; then the system $T[R]$, which is an extension of $T[R]_0$, with subkinding judgements of the form $\Gamma \vdash K <_c K'$. The rules for subkinding include, for example, the basic subkinding rule, that lifts subtyping to subkinding:

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

and the coercive definition rule:

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

As we are mainly concerned with the subtyping rules and their transitivity and coherence (in $T[R]_0$), we shall omit the details of the kind level in this paper. (See, for example, [17] for more details.)

$T[R]_0$ is an extension of T with the following rules:

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

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

In the presentation of coercive subtyping in [17], $T[R]_0$ also has the following substitution and transitivity rules:

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

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

Since we consider in this paper the admissibility of transitivity and substitution, we do not include the above two rules as basic rules.

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

Definition 2.4 (coherence condition) *We say that the subtyping rules are coherent if $T[R]_0$ 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 2.5 *This is a weaker notion of coherence as compared with that given in [17], since there the rules $(Subst)(Trans)$ are included in $T[R]_0$. 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.*

Well-defined coercions After new subtyping rules are added into R, we need to prove that the system 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, and 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* (WDC) [13].

Definition 2.6 (well-defined coercions) *If C is a set of subtyping judgements of the form $\Gamma \vdash M <_d M' : Type$ which satisfies the following conditions, we say that C is a* well-defined set of judgements for coercions, *or briefly called* Well-Defined Coercions (WDC).

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

Remark 2.7 *One may change the third condition (transitivity) to weak transitivity,* i.e., $\Gamma \vdash A <_{c_1} B : Type \in C$ *and* $\Gamma \vdash B <_{c_2} A' : Type \in C$ *imply* $\Gamma \vdash A <_{c_3} A' : Type \in C$ *for some* c_3 *such that* c_3 *and* $c_2 \circ c_1$ *are extensionally equal. This weaker condition is also sufficient for the following development in this paper, except that some lemmas and proofs require minor changes.*

In this paper, we consider the system of coercive subtyping in which the set R of the subtyping rules contains the following rule, where C is a WDC:

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

2.3 Inductive schemata and parameterised inductive types

Inductive types are generated by inductive schemata, as studied, for example, in [9,25]. In this subsection, we lay down some notations of inductive schemata, to be used in the next section (see Chapter 9 of [15] for more details). We shall first give some formal definitions and then some examples to explain.

Definition 2.8 (inductive schemata)

- *A strictly positive operator Φ, with respect to a type variable X and a valid context Γ, is of one of the following forms:*

(1) $\Phi \equiv X$ *, or*

(2) $\Phi \equiv (x : K)\Phi_0$*, where K is a small kind and Φ_0 is a strictly positive operator.*

- *An inductive schema Θ, with respect to a type variable X and a valid context Γ, is of one of the following forms:*

(1) $\Theta \equiv X$ *, or*

(2) $\Theta \equiv (x : K)\Theta_0$ *, where K is a small kind and Θ_0 is an inductive schema, or*

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

Any finite sequence of inductive schemata $\overline{\Theta} \equiv < \Theta_1, ..., \Theta_m > (m \in \omega)$ generates an inductive type $M[\overline{\Theta}]$, with its introduction, elimination and computation rules. In this paper, we shall consider the following form of parameterised inductive types:

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

where $Y_1, ..., Y_n$ are parameters (λ-abstracted bound variables) and $P_1, ..., P_n$ are kinds. One can specify such parameterised types in LF by declaring the following constant expressions, where $\overline{Y} = Y_1, ..., Y_n$:

$$\begin{aligned}
T &: (Y_1 : P_1)...(Y_n : P_n)Type \\
l_j &: (Y_1 : P_1)...(Y_n : P_n)\Theta_j[T(\overline{Y})] \quad (j = 1, ..., m) \\
E_T &: (Y_1 : P_1)...(Y_n : P_n)(C : (T(\overline{Y}))Type) \\
&\quad (f_1 : \Theta_1^\circ[T(\overline{Y}), C, l_1(\overline{Y})])... \\
&\quad (f_m : \Theta_m^\circ[T(\overline{Y}), C, l_m(\overline{Y})]) \\
&\quad (z : T(\overline{Y}))C(z)
\end{aligned}$$

and asserting the following computation rules: $(j = 1, ..., m)$

$$E_T(\overline{Y}, C, \overline{f}, (l_j(\overline{Y}, \Theta_j^v)) = f_j(\Theta_j^\sharp) : C(l_j(\overline{Y}, \Theta_j^v))$$

where Θ°, Θ^v and $\Theta^\sharp$ are formally introduced in the following definition and will also be used in latter sections.

Definition 2.9 *Let Φ be a strictly positive operator and Θ an inductive schema. For $A\!:\!Type$, $C\!:\!(A)Type$, $f\!:\!(x : A)C(x)$, $y\!:\!\Phi[A]$ and $z\!:\!\Theta[A]$,*

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

$$(X)^*[C,y] = C(y)$$
$$((x:K)\Phi_0)^*[C,y] = (x:K)\Phi_0^*[C,y(x)]$$

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

$$(X)^\circ[A,C,z] = C(z)$$
$$((x:K)\Theta_0)^\circ[A,C,z] = (x:K)\Theta_0^\circ[A,C,z(x)]$$
$$((x:\Phi)\Theta_0)^\circ[A,C,z] = (x:\Phi[A])(x':\Phi^*[C,x])\Theta_0^\circ[A,C,z(x)]$$

- *define* $\Phi^\natural[f,y]$ *as follows:*

$$(X)^\natural[f,y] = f(y)$$
$$((x:K)\Phi_0)^\natural[f,y] = [x:K]\Phi_0^\natural[f,y(x)]$$

- *Assume that* Θ *be of the form* $(x_1:M_1)...(x_s:M_s)X$ *and* $x_1,...,x_s$ *are fresh variables. Then* $\Theta^v = <x_1,...,x_s>$ *and* $\Theta^\sharp$ *is defined as the following sequence of arguments:*

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

Example 2.10 *We give three examples of parameterised inductive types.*

(1) Lists: List $=_{df} [A:Type]M[X,(A)(X)X]$*. This is equivalent to declaring the following constants:*

$$\begin{aligned} List &: (A)Type \\ nil &: (A:Type)List(A) \\ cons &: (A:Type)(a:A)(l:List(A))List(A) \\ E_{List} &: (A:Type)(C:(List(A))Type)(C(nil(A))) \\ &\quad ((a:A)(l:List(A))(C(l))C(cons(A,a,l))) \\ &\quad (z:List(A))C(z) \end{aligned}$$

with computation rules:

$$\begin{aligned} E_{List}(A,C,c,f,nil(A)) &= c : C(nil(A)) \\ E_{List}(A,C,c,f,cons(A,a,l)) &= f(a,l,E_{List}(A,C,c,f,l)) \\ &\quad : C(cons(A,a,l)) \end{aligned}$$

(2) Function types: $(\rightarrow) =_{df} [A:Type][B:Type]M[((A)B)X]$*.*

$$\begin{aligned} (\rightarrow) &: (A:Type)(B:Type)Type \\ \lambda &: (A:Type)(B:Type)((A)B)(A\rightarrow B) \\ E_{(\rightarrow)} &: (A:Type)(B:Type)(C:(A\rightarrow B)Type) \\ &\quad ((g:(A)B)C(\lambda(A,B,g)))(z:A\rightarrow B)C(z) \end{aligned}$$

with computation rule:

$$E_{(\to)}(A,B,C,f,\lambda(A,B,g)) = f(g){:}\,C(\lambda(A,B,g))$$

(3) *Either types (disjoint union): Either* $=_{df}$ $[A : Type][B : Type]M[(A)X,(B)X]$.

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

with computation rules:

$$\begin{aligned}
E_{Either}(A,B,C,f_1,f_2,left(A,B,a)) &= f_1(a){:}\,C(left(A,B,a)) \\
E_{Either}(A,B,C,f_1,f_2,right(A,B,b)) &= f_2(b){:}\,C(right(A,B,b))
\end{aligned}$$

3 WT-schemata and general subtyping rules

In this section, we define the WT-schemata and the general subtyping rules for those (parameterised) inductive types generated by the WT-schemata. The WT-schemata are those that generate (parameterised) inductive types whose subtyping rules satisfy the weak transitivity requirements.

3.1 WT-schemata

As briefly mentioned in the introduction, although it is suitable for subtyping rules of a large class of inductive types, weak transitivity is not admissible for some parameterised inductive types whose generation involves certain form of dependency between parameters. We start with this problem and explain that this is captured by the notion of WT-schemata.

A problem with weak transitivity Weak transitivity does not hold for the subtyping rules for every parameterised inductive type. For example, its admissibility fails for the subtyping rules for Σ-types and Π-types. An important observation is that the admissibility of weak transitivity fails for such types because they involve certain form of dependency between parameters. For example, $\Sigma =_{df} [A : Type][B : (A)Type]M[(x : A)(B(x))X]$ where $B(x)$ is dependent on the objects of parameter A. There are three subtyping rules for Σ-types, two of which are:

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

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

We can see that the coercion c in the first premise occurs in the second premise. And hence a proof of the admissibility of weak transitivity cannot go through. 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 above subtyping rule $(*)$ for Σ-types:

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

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

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 $x : A_1 \vdash c_1(x) : A_2$ we have $x : A_1 \vdash B_2(c_1(x)) = B_3(c_2(c_1(x)))$ and hence $x : A_1 \vdash B_1(x) = B_3(c_2(c_1(x)))$. But $x : A_1 \vdash B_1(x) = B_3(c_3(x))$ 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 example shows that weak transitivity is not admissible when we combine the subtyping rules for Σ-types and types of lists (the rule given in Section 1), *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.

Example 3.1 *Assume that we have some type constants A_1, A_2 and A_3, a constant B_3 of kind $(List(A_3))Type$, a WDC C generated by the congruence rule (Cong), and three coercions $A_1 <_{c_1} A_2$, $A_2 <_{c_2} A_3$ and $A_1 <_{c_2 \circ c_1} A_3$. By the subtyping rule for lists, we have $List(A_1) <_{d_1} List(A_2)$, $List(A_2) <_{d_2} List(A_3)$ and $List(A_1) <_{d_3} List(A_3)$, where d_1, d_2 and d_3 are defined as in Section 1. Note that d_3 and $d_2 \circ d_1$ are not computationally equal.*

Since $B_3 \circ d_2 : (List(A_2))Type$, by the above subtyping rule $(*)$, we have

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

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

Here, we omit the definition of e_1 and e_2.

Now, is the judgement $\Sigma(List(A_1), B_3 \circ d_2 \circ d_1) <_{e_3} \Sigma(List(A_3), B_3)$ derivable for some e_3? The answer is NO. We prove this by contradiction. If it is derivable, it can only be derived from the above subtyping rule (∗) (except several uses of the congruence rule). By coherence, which can be proved by the same method as in Section 4.1 and in [13], and the Church-Rosser property of the original type theory, we would have $d_3 = d_2 \circ d_1$, i.e., *they are computationally equal – a contradiction.*

Weak transitivity schemata The fact that the admissibility of weak transitivity fails for some parameterised inductive types has led us to introduce a restricted form of schemata, WT-schemata, which disallow that a coercion in one premise occurs in a type of another premise.

Definition 3.2 (WT-schemata) *Let Y be a set of parameters and Θ an inductive schema. Then Θ is a WT-schema w.r.t. Y if the following is the case:*

- *if $(x : K)M$ is a subterm of Θ and x occurs free in M, then K does not contain any of the parameters in Y.*

Remark 3.3 *Obviously, WT-schemata can be defined inductively as done for inductive schemata, but the above definition captures directly the dependency to be excluded.*

A parameterised inductive type T is generated by WT-schemata if

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

and each of the schemata in $\overline{\Theta}$ is a WT-schema w.r.t. $Y = \{Y_1, \ ..., Y_n\ \}$. The above notion of WT-schema covers a large class of parameterised inductive types such as lists, $Maybe$ types, $Either$ types (disjoint union), function types, product types, types of branching trees, etc. What it excludes are those parameterised types such as Σ-types and Π-types.

3.2 Subtyping rules and coercions

Now, we consider how to define subtyping rules and the associated coercions for the parameterised types of the form

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

generated by WT-schemata $\overline{\Theta}$ wrt the parameters $Y_1, ..., Y_n$. The general form of the subtyping rules of T is

$$(**) \qquad \frac{premises}{\Gamma \vdash T(\overline{A}) <_{d_T} T(\overline{B}) : Type}$$

where $\overline{A} = A_1, ..., A_n$ and $\overline{B} = B_1, ..., B_n$ are fresh and distinct schematic letters. Intuitively, we associate T with subtyping rules whose conclusion is of the form $T(\overline{A}) <_{d_T} T(\overline{B})$ such that the coercion d_T is defined by induction on $T(\overline{A})$ and maps the canonical objects of $T(\overline{A})$ to the corresponding canonical objects of $T(\overline{B})$. For example, for the type of lists $List(Y)$ parameterised by the type parameter Y, the subtyping rule is

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

where the coercion $map(A, B, c)$ is defined as

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

which maps the canonical objects $nil(A)$ to $nil(B)$ and $cons(A, a, l)$ to $cons(B, c(a), map(A, B, c)(l))$. Note that the definition of the coercion $map(A, B, c)$ is dependent on the premise, in particular, the assumed coercion c in the premise.

3.2.1 A formal definition

The premises and the corresponding definition of the coercion d_T of the rules of the form $(**)$ are given below. We first give a generic form of the sequence of premises and the corresponding coercion ($Prem_\Gamma(\overline{\Theta})$ and D_T below), and then specify how to make instantiations to obtain the concrete premises and coercions d_T of the rules.

Notation 3.4 *In the following, we shall write $D[\overline{A}]$ for $[A_1/Y_1, ..., A_n/Y_n]D$ and $D[\overline{B}]$ for $[B_1/Y_1, ..., B_n/Y_n]D$. Also, we 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 3.5 (premise set)

- *For 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 a WT-schema* Θ *in* Γ *w.r.t. the parameters* $Y = \{Y_1, ..., Y_n\}$*, 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* $prem_\Gamma(\Theta) = prem_\Gamma(K) \cup prem_{\Gamma,x:K}(\Theta_0)$*. Note that in this case,* $\overline{Y} \notin FV(K)$*.*

(3) $\Theta \equiv (x : \Phi)\Theta_0$*, then* $prem_\Gamma(\Theta) = \overline{prem_\Gamma(\Phi)} \cup prem_\Gamma(\Theta_0)$*, where the definition of* $\overline{prem_\Gamma(\Phi)}$ *is the same as that above.*

- *For any sequence of WT-schemata* $\overline{\Theta} \equiv <\Theta_1, ..., \Theta_m>$ *in* Γ *w.r.t. the parameters* $Y = \{Y_1, ..., Y_n\}$*, we define*

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

Now, we give an order to the elements of $prem_\Gamma(\overline{\Theta})$:

$$(\Gamma_1, A_1, B_1), ..., (\Gamma_m, A_m, B_m).$$

Then, the sequence of premise forms w.r.t Γ and $\overline{\Theta}$, $Prem_\Gamma(\overline{\Theta})$, is

$$\Gamma_1 \vdash A_1 \leq_{c_1} B_1 : Type,\ ...,\ \Gamma_m \vdash A_m \leq_{c_m} B_m : Type,$$

where the schematic letters c_i $(i = 1, ..., m)$ are fresh and distinct.

Having defined the general forms of the premises, we now define a general form of the corresponding coercion. We first introduce the following notational definition.

Definition 3.6 *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* $Prem_\Gamma(\overline{\Theta})$*, 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* $Func[K_{12}, K_{22}]$ *and* $Func[K_{21}, K_{11}]$ *are defined, then*

$$Func[K_1, K_2] = [g : K_1][y : K_{21}]Func[K_{12}, K_{22}](g(Func[K_{21}, K_{11}](y))).$$

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

Remark 3.7 *In general, when c in $\Gamma \vdash C \leq_c D{:}\,Type$ is of kind $(C)D$, $Func[K_1, K_2]$ is of kind $(K_1)K_2$.*

Notation 3.8 *Let $Y_1, ..., Y_n$ be the parameters and Ψ be either a 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*
- *$\Psi[\overline{B}][T(\overline{B})]$ for $[B_1/Y_1, ..., B_n/Y_n, T(\overline{B})/X]\Psi$.*

Definition 3.9

- *Let Φ be a strictly positive operator. For any $f{:}\,\Phi[\overline{A}][T(\overline{B})]$, define $\Phi^k[f]$ of kind $\Phi[\overline{B}][T(\overline{B})]$ as follows:*

(1) if $\Phi \equiv X$ then $\Phi^k[f] = f$;
(2) if $\Phi \equiv (x : K)\Phi_0$, then

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

- *Let Θ be a WT-schema. For any $g{:}\,\Theta[\overline{B}][T(\overline{B})]$, 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 define

$$\Theta^\lambda(g) = [x : \Phi[\overline{A}][T(\overline{A})]][x' : \Phi[\overline{A}][T(\overline{B})]]\Theta_0^\lambda(g(\Phi^k[x']))$$

Then, we define

$$D_T =_{df} E_T(\overline{A}, C, \Theta_1^\lambda(l_1(\overline{B}))), ..., \Theta_m^\lambda(l_m(\overline{B})))$$

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

Now we are ready to specify the rules of the form $(**)$, the premises and the coercion. Let $Prem_\Gamma(\overline{\Theta})$ be the following sequence of length m:

$$\Gamma_1 \vdash A_1 \leq_{c_1} B_1{:}\,Type, ..., \Gamma_m \vdash A_m \leq_{c_m} B_m{:}\,Type,$$

then there are $2^m - 1$ rules of the form $(**)$ for the parameterised inductive type, each of which has m 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 $\Gamma \vdash A <_c B{:}\,Type$.

For each sequence of premises, the corresponding coercion d_T is obtained as follows. Define

$$d_i =_{df} \begin{cases} id_{A_i} & \textit{if the ith premise is } \Gamma_i \vdash A_i = B_i{:}\,Type \\ c_i & \textit{if the ith premise is } \Gamma_i \vdash A_i <_{c_i} B_i{:}\,Type \end{cases}$$

Then

$$d_T =_{df} [\overline{d}/\overline{c}]D_T,$$

where $\overline{d} = d_1, ..., d_m$ and $\overline{c} = c_1, ..., c_m$.

Remark 3.10 *Some rules have contradictory premises. For example, one of the subtyping rules for inductive type* $T(Y) =_{df} M[((Y)Y)X]$ *parameterised by type variable* Y *is*

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

Since the premises in such rules are contradictory (and never satisfied), they can never be applied.

3.2.2 Justification of the coercion d_T

The fact that the coercion d_T as defined above sends the canonical objects of $T(\overline{A})$ to the corresponding canonical objects in $T(\overline{B})$ is described and proved in the following lemma.

Definition 3.11 *Let* Θ *be a WT-schema and assume that* Θ *be of the form* $(x_1 : M_1)...(x_s : M_s)X$ *and* $x_1, ..., x_s$ *are fresh variables.* $\Theta^u(\overline{A}, \overline{B})$ *is the sequence of arguments defined as follows:*

(1) If $\Theta \equiv X$*, then* $\Theta^u(\overline{A}, \overline{B}) =<>$.
(2) If $\Theta \equiv (x_t : K)\Theta_0$ $(t = 1, ..., s)$*, then*

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

(3) If $\Theta \equiv (x_t : \Phi)\Theta_0$ $(t = 1, ..., s)$ *then*

$$\Theta^u(\overline{A}, \overline{B}) =< \Phi^k[\Phi^\natural[d_T, x_t]], \Theta_0^u(\overline{A}, \overline{B}) > .$$

Lemma 3.12 $d_T(l_j(\overline{A}, \Theta_j^v)) = l_j(\overline{B}, \Theta_j^u(\overline{A}, \overline{B}))$*, where* Θ_j^v *as defined in Section 2.3.*

Proof *By the definition of* d_T *and the computation rules for* T *(see page 10), we have*

$$d_T(l_j(\overline{A}, \Theta_j^v)) = E_T(\overline{A}, C, f_1, ..., f_m, l_j(\overline{A}, \Theta_j^v)) = \Theta_j^\lambda(l_j(\overline{B}))(\Theta_j^\sharp).$$

Now, we need to prove that $\Theta_j^\lambda(l_j(\overline{B}))(\Theta_j^\sharp) = l_j(\overline{B}, \Theta_j^u(\overline{A}, \overline{B}))$*. Rather than proving it directly, we generalise the problem first; for any* $g{:}\,\Theta[\overline{B}][T(\overline{B})]$, $\Theta_j^\lambda(g)(\Theta_j^\sharp) = g(\Theta_j^u(\overline{A}, \overline{B}))$*. This can be proved by induction on the structures of the WT-schemata.*

3.2.3 Examples

For the last two parameterised inductive types in Example 2.10 in Section 2.3, the subtyping rules and associated coercions are as follows (those for lists have been given above).

Example 3.13

(1) The subtyping rules for Either types:

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

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

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

where

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

satisfying

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

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

(2) Subtyping rules for function types:

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

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

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

where

$$d_{(\to)3} =_{df} E_{(\to)}(A, B, [z : A \to B](A' \to B'),$$
$$[g : (A)B]\lambda(A', B', c_2 \circ g \circ c_1))$$

satisfying $d_{(\to)3}(\lambda(A, B, g)) = \lambda(A', B', c_2 \circ g \circ c_1)$. *The definitions of* $d_{(\to)1}$ *and* $d_{(\to)2}$ *are similar to* $d_{(\to)3}$.

4 Coherence, admissibility results and equality requirement

In this section, we show that the subtyping rules defined as above, for the inductive types generated by the WT-schemata, satisfy coherence and the admissibility results as expected, including that of weak transitivity. Furthermore, we show that these subtyping rules satisfy the equality requirement for weak transitivity, that is, if A is a subtype of B via coercion c and B a subtype of C via coercion c', then the coercion c'' from A to C is extensionally equal to the composition of c and c'.

4.1 Coherence and admissibility results

We show that coherence and admissibility of weak transitivity and substitution are satisfied by $T[R]$, where T is the original type theory and the set of subtyping rules R consists of the following:

- The subtyping rules for parameterised inductive types T as defined above. We assume that these inductive types are different, i.e., $T \not\equiv T'$ for any two such inductive types.
- A set of well-defined coercions C. We assume that for any judgement $\Gamma \vdash A <_c B : Type \in C$, neither A nor B is computationally equal to any T-type. Also note that we have the rule (C) in the system.

Furthermore, we 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 C_M the set of the derivable subtyping judgements of the form $\Gamma \vdash M <_d M' : Type$ in $T[R]_0$; that is, $\Gamma \vdash M <_d M' : Type \in C_M$ if and only if $\Gamma \vdash M <_d M' : Type$ is derivable in $T[R]_0$.

It is then not difficult to prove the following lemma by induction on derivations.

Lemma 4.1

(1) *If* $\Gamma \vdash M_1 <_d M_2 \colon Type \in C_M$ *then both* M_1 *and* M_2 *are computationally equal to a T-type or* $\Gamma \vdash M_1 <_d M_2 \colon Type \in C$.
(2) ***(context equality)*** *If* $\Gamma \vdash M_1 <_d M_2 \colon Type \in C_M$ *and* $\vdash \Gamma = \Gamma'$ *then* $\Gamma' \vdash M_1 <_d M_2 \colon Type \in C_M$.
(3) ***(weakening)*** *If* $\Gamma \vdash M_1 <_d M_2 \colon Type \in C_M$, $\Gamma \subseteq \Gamma'$ *and* Γ' *is valid then* $\Gamma' \vdash M_1 <_d M_2 \colon Type \in C_M$.

The following theorem can be proved by induction on derivations and using Lemma 4.1(1).

Theorem 4.2

(1) ***(Coherence)***
  (a) *If* $\Gamma \vdash M_1 <_d M_2 \colon Type \in C_M$ *then* $\Gamma \not\vdash M_1 = M_2 \colon Type$.
  (b) *If* $\Gamma \vdash M_1 <_d M_2 \colon Type \in C_M$, $\Gamma \vdash M_1' <_{d'} M_2' \colon Type \in C_M$, $\Gamma \vdash M_1 = M_1' \colon Type$ *and* $\Gamma \vdash M_2 = M_2' \colon Type$ *then* $\Gamma \vdash d = d' \colon (M_1)M_2$.
(2) ***(Substitution)*** *If* $\Gamma, x : K, \Gamma' \vdash M_1 <_d M_2 \colon Type \in C_M$ *and* $\Gamma \vdash k \colon K$ *then* $\Gamma, [k/x]\Gamma' \vdash [k/x]M_1 <_{[k/x]d} [k/x]M_2 \colon Type \in C_M$.
(3) ***(Weak Transitivity)*** *If* $\Gamma \vdash M_1 <_{d_1} M_2 \colon Type \in C_M$, $\Gamma \vdash M_2' <_{d_2} M_3 \colon Type \in C_M$ *and* $\Gamma \vdash M_2 = M_2' \colon Type$ *then* $\Gamma \vdash M_1 <_{d_3} M_3 \colon Type \in C_M$ *for some* d_3.

Remark 4.3 *In [13], we use the measure of depth, introduced by Chen in his PhD thesis [8], to prove the admissibility of the (strong) transitivity rule. But here, the proof concerning weak transitivity is simply by induction on derivations since WT-schemata disallow the dependency where a coercion in one premise can occur in another premise.*

4.2 Equality requirement for weak transitivity

In the coercive subtyping framework, a subtyping relation between two types means that there is a (unique) coercion between them. However, such a coercion should not be an arbitrary one; in particular, if $\Gamma \vdash A <_c B$ and $\Gamma \vdash B <_{c'} C$, then the coercion from A to C must be in some sense related to $c' \circ c$, the composition of c and c'. As we have mentioned earlier, in such a case, we require that the coercion from A to C be extensionally equal to $c' \circ c$.

There are choices one may make about this notion of extensional equality. First, note that, although we have considered coercive subtyping in intensional type theories, the equality requirement for weak transitivity can be considered to be at the meta-level, and hence can be outside the intensional type theory. One of such choices, that we adopt here, is the notion of equality in extensional type theory [21].

In an extensional type theory, one has the following rule

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

where A is a type, a and b are objects of type A, Eq is the propositional equality (Martin-Löf's equality type or the Leibniz equality), and $=$ is the judgemental equality. One can consider an extension of the intensional type theory (which has Eq-types) by the above rule to obtain the corresponding extensional theory. Note that the above rule makes the resulting type theory undecidable and loses its property of strong normalisation. However, it does capture the notion of extensional equality in a strong sense.

We can now use the above notion of extensional equality to express our equality requirement about weak transitivity.

- **Equality requirement:** If $\Gamma \vdash A <_c B : Type$, $\Gamma \vdash B <_{c'} C : Type$, and $\Gamma \vdash A <_{c''} C : Type$, then $\Gamma \vdash c'' = c' \circ c : (A)C$ in the extentional type theory.

The following theorem says that the equality requirement is satisfied by the general subtyping rules for parameterised inductive types as defined in Section 3.

Theorem 4.4 (equality requirement) *If* $\Gamma \vdash A <_c B : Type$, $\Gamma \vdash B <_{c'} C : Type$ *and* $\Gamma \vdash A <_{c''} C : Type$ *and are all in* C_M, *then* $\Gamma \vdash c'' = c' \circ c : (A)C$ *in the extensional type theory.*

Proof *By induction on derivations and using Lemma 4.1(1) and Lemma 3.12.*

5 Extensional computation rules: a discussion

The problem of transitivity we have considered in this paper arises from the fact that certain extensional equalities do not hold computationally in intensional type theories. For example, in an intensional type theory, the following equality

$$map(E, N, c_2) \circ map(F, E, c_1) = map(F, N, c_2 \circ c_1)$$

does not hold computationally, and hence we have a coherence problem, if we include the strong transitivity rule ($Trans$), as shown in the Introduction section. This has led us to introduce weak transitivity and study the related coherence and admissibility properties.

One might consider a restricted form of extensional equality to be 'computational' and introduce them as computational rules. For example, we might simply stipulate that the above equality is computational. In general, we can stipulate such equalities as computational for all of the parameterised inductive types generated by any inductive schemata, including those excluded by the restriction in WT-schemata. (See [12] for the details.) Then, it can be shown that, with such rules, inductive types can be associated with general subtyping rules as defined systematically in the same way as in the last section and these subtyping rules are coherent and satisfying transitivity elimination even in the presence of the strong transitivity rule ($Trans$).

However, it is not clear at all whether such equality rules should or could be taken as computational. First of all, these rules are not 'computational' in nature. They are rules concerning the commutative features of the composition operator. Intuitively, it is rather difficult to consider them as computational, although when considered so, they do provide smooth treatment of the subtyping rules.

Secondly, not less importantly, it is not clear whether such rules, if considered computational, have any negative impacts on the meta-theory of the resulting type theory. For example, it is unclear whether the resulting type theory would still have the properties of strong normalisation, Church-Rosser and subject reduction. (We do not have either counter-examples or proofs about them.) Such a problem could be rather difficult to settle.

6 Conclusion

In this paper, we have studied transitivity in coercive subtyping. In particular, after explaining a problem with the strong notion of transitivity as originally considered, we have introduced the notion of weak transitivity and shown that the parameterised inductive types generated by the WT-schemata can be associated with natural subtyping rules to be shown coherent and satisfying good properties such as the admissibility of weak transitivity.

The WT-schemata as described above in the paper generate only inductive types. They can be extended to inductive families of types straightforwardly and the above results concerning subtyping extend naturally, too. For example, the inductive family of types of vectors, which is parameterised by the object type A, can be defined as follows:

$$Vec(A) =_{df} M[X(0), (n : N)(A)(X(n))X(S(n))]$$

where X is a place holder of kind $(N)Type$, N the type of natural numbers, and 0 and S are constructors for zero and the successor respectively (see [15] for more details). A common subtyping rule for vectors is the following:

$$\frac{\Gamma \vdash n\colon N \quad \Gamma \vdash A <_c B\colon Type}{\Gamma \vdash Vec(A,n) <_{d(n)} Vec(B,n)\colon Type}$$

where

$$\begin{aligned} d(0, vnil(A)) &= vnil(B) \\ d(S(m), vcons(A,m,a,l)) &= vcons(B,m,c(a),d(m,l)) \end{aligned}$$

and *vnil* and *vcons* are the constructors of vectors introduced as usual. Adding this subtyping rule into R, all the good properties are kept, *i.e.*, R is still coherent, substitution rule is admissible, weak transitivity holds and equality requirement is satisfied. Note that Vec is a dependent family. As mentioned in Section 3, WT-schemata avoid the kind of dependency between parameters such as that for Σ-types to make sure that there is no coercion in one premise that occurs in another premise. The above subtyping rule for vectors does not have such dependency.

In the above section, we have discussed the issue of considering certain restricted form of extensional equality as computational in order to solve the transitivity problem. One naturally thinks that an interesting issue to be studied is how transitivity (and subtyping in general) works in extensional type theories. Although extensional type theories are undecidable and arguably not suitable for implementation or practical use, it may still be worth being studied. However, the topic might not be as easy as it appears to be, partly because that we are aware of some technical difficulties to work in an extensional type theory. On the other hand, to study coercive subtyping and its related issues in an extensional framework may provide further theoretical insights.

References

[1] P. Aczel. Simple overloading for type theories. Draft, 1994.

[2] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998.

[3] B. Barras et al. *The Coq Proof Assistant Reference Manual (Version 6.3.1).* INRIA-Rocquencourt, 2000.

[4] G. Barthe and M.J. Frade. Constructor subtyping. *Proceedings of ESOP'99, LNCS 1576*, 1999.

[5] G. Barthe and F. van Raamsdonk. Constructor subtyping in the calculus of inductive constructions. *Proceedings of FOSSACS'00, LNCS 1784*, 2000.

[6] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. *Information and Computation*, 93, 1991.

[7] P. Callaghan and Z. Luo. An implementation of LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

[8] G. Chen. *Subtyping, Type Conversion and Transitivity Elimination*. PhD thesis, University of Paris VII, 1998.

[9] 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.

[10] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Proc. 2nd Ann. Symp. on Logic in Computer Science. IEEE*, 1987.

[11] G. Longo, K. Milsted, and S. Soloviev. Coherence and transitivity of subtyping as entailment. *Journal of Logic and Computation*, 10(4), 2000.

[12] Y. Luo. Coherence and transitivity in coercive subtyping. Forthcoming PhD thesis, University of Durham., 2004.

[13] 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 *LNAI*, pages 249–265. Springer-Verlag, 2001.

[14] Y. Luo, Z. Luo, and S. Soloviev. Weak transitivity in coercive subtyping. *Types for Proofs and Programs, Proc. of Inter Conf TYPES'02. LNCS 2646.*, 2003.

[15] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[16] Z. Luo. Coercive subtyping in type theory. *Proc. of CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic, Utrecht. LNCS 1258*, 1997.

[17] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[18] Z. Luo and P. Callaghan. Coercive subtyping and lexical semantics (extended abstract). *LACL'98*, 1998.

[19] 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.

[20] 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.

[21] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[22] J. C. Mitchell. Coercion and type inference. In *Proc. of Tenth Annual Symposium on Principles of Programming Languages (POPL)*, 1983.

[23] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(2):245–286, 1991.

[24] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[25] 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.

[26] A. Saibi. Typing algorithm in type theory with inheritance. *Proc of POPL'97*, 1997.

[27] S. Soloviev and Z. Luo. Coercion completion and conservativity in coercive subtyping. Annals of Pure and Applied Logic, 2002.

Appendix A

The following gives the rules of the logical framework LF.

Contexts and assumptions

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

Equality rules

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

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

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

Substitution rules

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

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

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

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

The kind Type

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

Dependent product kinds

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

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

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

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