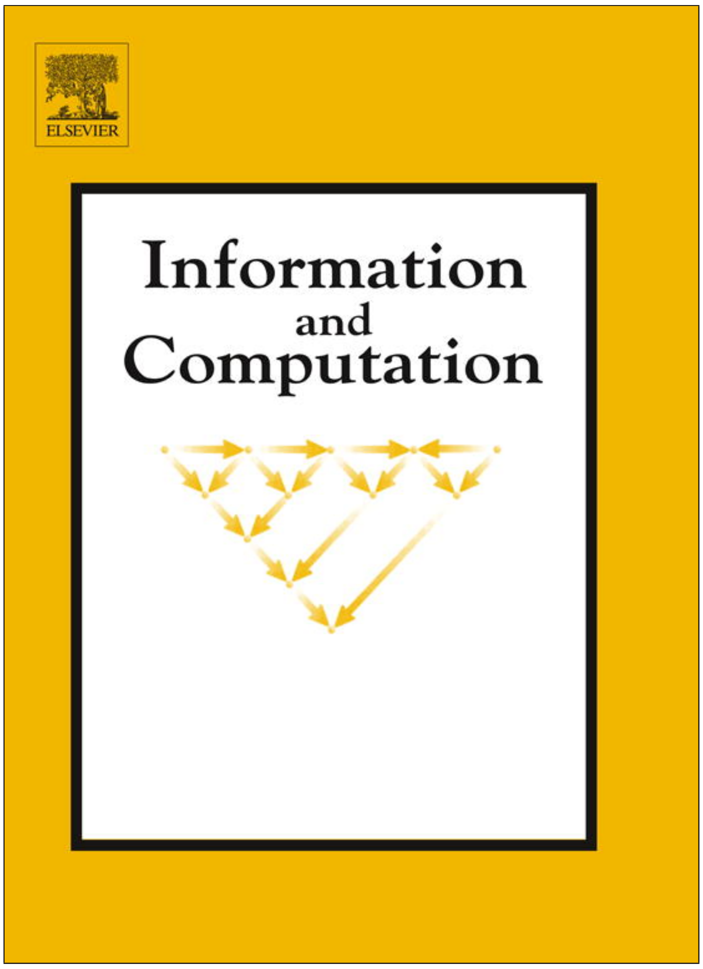

Information and Computation 223 (2013) 18–42

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Coercive subtyping: Theory and implementation

Z. Luo [a,*,1], S. Soloviev [b], T. Xue [a]

[a] *Department of Computer Science, Royal Holloway, University of London, Surrey, UK*
[b] *IRIT, 118 route de Narbonne, 31062 Toulouse cedex 4, France*

A R T I C L E   I N F O

Article history:
Received 15 June 2012
Available online 13 November 2012

Keywords:
Coercive subtyping
Conservativity
Definitional extension
Subsumptive subtyping
Type theory

A B S T R A C T

Coercive subtyping is a useful and powerful framework of subtyping for type theories. The key idea of coercive subtyping is subtyping as abbreviation. In this paper, we give a new and adequate formulation of $T[\mathcal{C}]$, the system that extends a type theory T with coercive subtyping based on a set $\mathcal{C}$ of basic subtyping judgements, and show that coercive subtyping is a conservative extension and, in a more general sense, a definitional extension. We introduce an intermediate system, the star-calculus $T[\mathcal{C}]^*$, in which the positions that require coercion insertions are marked, and show that $T[\mathcal{C}]^*$ is a conservative extension of T and that $T[\mathcal{C}]^*$ is equivalent to $T[\mathcal{C}]$. This makes clear what we mean by coercive subtyping being a conservative extension, on the one hand, and amends a technical problem that has led to a gap in the earlier conservativity proof, on the other. We also compare coercive subtyping with the 'ordinary' notion of subtyping – subsumptive subtyping, and show that the former is adequate for type theories with canonical objects while the latter is not. An improved implementation of coercive subtyping is done in the proof assistant Plastic.

1. Introduction

Coercive subtyping is a useful and powerful framework of subtyping for type theories with canonical objects.[2] It extends such type theories with a suitable notion of subtyping for effective applications such as proof development and dependently-typed programming.[3]

The basic idea of coercive subtyping is that subtyping is modelled as an abbreviation mechanism: A is a subtype of B, if there is a unique coercion c from A to B, written as $A <_c B$. Then, if a hole in a context requires an object of type B, it is legal to supply an object a of type A – it is equivalent to supplying the object $c(a)$. This simple idea, when used with the rich type structures in type theories with inductive types, provides a powerful and general subtyping mechanism. It subsumes injective subtyping (e.g., those between basic types such as $Nat < Int$ and $Man < Human$), projective subtyping (e.g., those between Σ-types or record types induced by their projection operations) and structural subtyping (e.g., those between inductive types induced by the subtyping relations of their parameters).

* Corresponding author.
E-mail addresses: zhaohui.luo@hotmail.co.uk (Z. Luo), soloviev@irit.fr (S. Soloviev), taoxue@cs.rhul.ac.uk (T. Xue).

[1] Partially supported by the Leverhulme research grant F/07-537/AJ.

[2] Type theories with canonical objects are sometimes called 'modern type theories', examples of which include Martin-Löf's type theory [24,28], the Unifying Theory of dependent Types (UTT) [11,6] and the type theory implemented in the Coq proof assistant (pCIC) [5].

[3] For instance, coercions have been implemented and used in several proof assistants such as Coq [5,30], Lego [21,3], Matita [25] and Plastic [4].

0890-5401/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ic.2012.10.020

As an abbreviation mechanism, coercive subtyping extension should be conservative over the original type theory and, in particular, logical consistency of the original type theory should be preserved.[4] However, the coercive subtyping extension has the same syntax of terms with the original type theory (see Section 2.2 for formal details), the notion of conservative extension could not be directly applied to them and, as a consequence, in its earlier treatment [31], the notion of conservativity for coercive subtyping was not clearly spelled out and, in particular, it was not explicitly linked to the traditional studies in logic. In this paper, we give a new and adequate formulation of coercive subtyping and show that it is a conservative extension by introducing an intermediate system, the star-calculus $T[\mathcal{C}]^*$, in which the positions that require coercion insertions are marked by the $*$-symbol, and showing that $T[\mathcal{C}]^*$ is a conservative extension of T and that $T[\mathcal{C}]^*$ is equivalent to the coercive subtyping extension $T[\mathcal{C}]$. This makes it clear what we mean by conservativity of the coercive subtyping extension.

As reported in this paper, the earlier formulation of coercive subtyping by means of the so-called basic subtyping rules [13] has been found to be unnecessarily general in that it does not exclude certain 'bad' subtyping rules which would destroy the much desired property of conservativity. As a consequence it has led to an incomplete proof of conservativity in [31]. The formulation of coercive subtyping given in this paper solves this problem and, furthermore, this new formulation is general enough to encompass the earlier one.

It is also explained that coercive subtyping is not only a conservative extension, but essentially a *definitional extension*, in a more general sense. This is reflected in that, as long as the basic coercions are coherent, unique coercions can always be inserted to obtain the abbreviated expressions correctly. This is closely related to the logical notion of definitional extension, where a first-order theory is a definitional extension of another if it is not only the case that the former is a conservative extension of the latter but also the case that any formula in the former is logically equivalent to its translation in the latter [7]. In the context of coercive subtyping, this second extra requirement is captured by means of a notion of equivalence between derivations in type theory, as to be explained in the paper.

Besides coercive subtyping, there is a traditional 'ordinary' notion of subtyping based on the following subsumption rule:

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

Let us call such systems as systems with *subsumptive subtyping*. As far as we know, in the literature, there are only scattered remarks to compare these two notions of subtyping (see, for example, §2.1 of [20]). In Section 4, we shall analyse them from a particular angle, discussing the idea that subsumptive subtyping is suitable for type assignment systems as employed in functional programming languages, but not suitable for type theories with canonical objects as implemented in proof assistants, and showing that coercive subtyping provides a very general framework for the latter.

The new theory of coercive subtyping has been implemented in the proof assistant Plastic, based on an earlier implementation of coercions in the system by Callaghan [4], and used for some case studies. We shall describe the implementation and illustrate its use with examples in formal semantics of natural languages.

In Section 2, we present the new formulation of coercive subtyping, giving the formal presentations of both $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$, and briefly explains its adequacy both as a conservative extension of the original type theory and as a theory that encompasses the seemingly more general earlier formulation by means of rules. The proof that unique coercions can be correctly inserted, and hence that the coercive subtyping extension is conservative, and definitional in a more general sense, is given in Section 3. In Section 4, we compare coercive subtyping with subsumptive subtyping and explain that the former is adequate for type theories with canonical objects while the latter is not. The implementation of coercive subtyping in Plastic is described in Section 5.

2. Coercive subtyping

Coercive subtyping is based on the idea that subtyping is abbreviation.[5] As an abbreviational extension that allows one to omit certain expressions called coercions, it should intuitively be conservative over the original type theory. However, in the previous treatments of the notion of conservativity for coercive subtyping [31], it was not explicitly linked to that in the traditional studies of logic and, as a consequence, it was not as well understood as it should have been.

In this section, for a type theory T and a set of subtyping judgements $\mathcal{C}$, we present the formal system $T[\mathcal{C}]$ for the coercive subtyping extension of T and, at the same time, describe an intermediate system $T[\mathcal{C}]^*$ in which the positions for coercion insertion are marked. As the proof in the next section shows, $T[\mathcal{C}]^*$ is a conservative extension of T (in the traditional sense) and equivalent to $T[\mathcal{C}]$. In this way, we make it clear that coercive subtyping is a conservative extension.

Coercive subtyping has the property that, as long as the basic coercions are coherent, coercions can always be correctly inserted to obtain the abbreviated expressions (see the next section for its formal proof). This has turned out to be closely related to the notion of *definitional extension*, in a more general sense. In traditional studies of logic, the notion of definitional

[4] A type theory T' is a conservative extension of T if T' extends T in such a way that every T-judgement is derivable in T' if, and only if, it is derivable in T. This definition follows that of the traditional notion, although derivability of judgements is employed instead of provability (or truth) of formulas.

[5] The idea of subtyping as coercions were studied for simpler type systems by many authors including, for example, Mitchell [26,27] for the simply-typed λ-calculus and Longo et al. [9] about the second-order λ-calculus. There have also been studies of subtyping on dependent types including Aspinall and Compagnoni [2] and Longo [8]. In this paper, coercive subtyping is studied for modern type theories.

extension was formulated for first-order logical theories [7]: a first-order theory S' is a definitional extension over S if S' is a conservative extension of S and, in S', any formula in S' is logically equivalent to its translation in S. Intuitively, the extra symbols in a definitional extension can be replaced correctly by their translations in the original theory. In the context of coercive subtyping, such translations correspond to the abbreviated expressions after coercion insertions and, this is captured by a notion of equivalence between derivations.[6]

Our new formulation of coercive subtyping also solves a problem of an earlier one in [13] which has led to a gap in the conservativity proof in [31]. In fact, the new formulation not only solves it but leads to a theory that is general enough to encompass the old one.

2.1. An informal account of a problem

The formulation of coercive subtyping in [13] is not quite adequate: it is unnecessarily general and, in particular, does not exclude certain 'bad' coercion rules that destroy the much desired property of conservativity. As a consequence it has led to an incomplete proof of conservativity in [31]. In this subsection, this is explained. Let us first start with the introduction of several basic notions.

Basic idea of coercive subtyping. The basic idea of coercive subtyping is that subtyping is modelled as an abbreviation mechanism: A is a subtype of B, if there is a unique coercion c from A to B, written as $A <_c B$. Then, if a hole in a context requires an object of B, it is legal to supply an object a of A – it is equivalent to supplying the object $c(a)$.

The theory of coercive subtyping was first studied in [12] and extended later to a more general setting in [13]. In a type theory specified in a logical framework such as LF (see Appendix A for formal description), a context with a hole that requires an object of B can be represented as a functional operation with domain B and the above basic idea of coercive subtyping can be captured formally by means of the following rules:

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

They state that, if $A <_c B$, then f with domain B may be applied to an object a of A, which actually stands for its image $c(a)$ under the coercion c, to form $f(a)$ and the result is equal to $f(c(a))$. If one considers f as a context with a 'hole' of B, then it can be supplied with an object a of A, standing for $c(a)$. These rules are formally called *coercive application/definition rules*.

Coherence and conservativity. Intuitively, a system with coercions is coherent if all coercions between any two types are the same. In other words, $c = c'$ if $A <_c B$ and $A <_{c'} B$. However, this notion of coherence cannot be defined formally for the whole system with coercive subtyping and, in particular, it cannot be defined if the above coercive definition rule $(**)$ is present, for otherwise, any two coercions between the same types can be proved to be equal.[7] Therefore, it has to be defined by means of a subsystem that does not contain the rule $(*)$ or $(**)$.

Such a formulation was given in [13] where, given a set $\mathcal{R}$ of basic subtyping rules, the coercive subtyping extension $T[\mathcal{R}]$ of a type theory T is defined as an extension of the subsystem $T[\mathcal{R}]_0$, which does not contain the above rule $(*)$ or $(**)$. The notion of coherence is defined by means of $T[\mathcal{R}]_0$: $\mathcal{R}$ is coherent if, for any two coercions c and c' from A to B, we can prove in $T[\mathcal{R}]_0$ that $c = c'$. Based on this formulation, the conservativity theorem (i.e., coherence implies conservativity of $T[\mathcal{R}]$ over T) was studied in [31]. Unfortunately, as mentioned above, this notion of conservativity is not well explained and, furthermore, it has become known later that there was a gap in the proof.[8] In fact, the formulation of coercive subtyping in [13] was too general for the result to hold, as explained in more details below.

Problem with the basic subtyping rules. In the formulation of the coercive subtyping extension $T[\mathcal{R}]$ in [13], it is only required that $\mathcal{R}$ be a set of rules whose conclusions are subtyping judgements of the form $\Gamma \vdash A <_c B : Type$. This is not exclusive enough: it does not exclude some 'bad' rules, for example the rules which are non-applicable in the subsystem $T[\mathcal{R}]_0$ but become applicable in the whole system $T[\mathcal{R}]$. Here is an example that involves such a rule.

Example 2.1. Let A and B be types (e.g., $A \equiv Nat$ is the type of natural numbers and $B \equiv Bool$ the type of booleans) and c_1 and c_2 functional operations from A to B such that we cannot prove that they are equal in the original type theory T. Consider the set $\mathcal{R}$ that consists of the following two rules:

$$(\#)\quad \frac{\Gamma \vdash valid}{\Gamma \vdash A <_{c_1} B : Type} \qquad (\#\#)\quad \frac{\Gamma \vdash a:A \quad \Gamma \vdash f:(B)B \quad \Gamma \vdash f(a):B}{\Gamma \vdash A <_{c_2} B : Type}$$

[6] As logical systems, type theories are more complicated than first-order theories. For instance, the well-formedness of expressions is not given syntactically, but governed by judgemental derivability: whether an expression A is a type (which in traditional logics corresponds to the well-formedness of a formula) is given by the derivability of a judgement of the form $A : Type$ and whether a is of type A by the derivability of $a : A$.

[7] Formally, if $A <_c B$ and $A <_{c'} B$, then using the rules $(**)$ and the $(\beta\eta\xi)$-equality rules in the logical framework (see Appendix A), one can prove that $c = c' : (A)B$.

[8] We (the first two authors) realised the problem when discussing a question raised by Robin Adams in 2007.

Note that, without the coercive application rule $(*)$, the third premise of the above rule (##) can never be derivable; in other words, in $T[\mathcal{R}]_0$, the rule (##) can never be applied – it is non-applicable in $T[\mathcal{R}]_0$. Therefore, $\mathcal{R}$ is coherent with only one coercion c_1 from A to B.

However, in the whole system $T[\mathcal{R}]$, where the coercive application rule $(*)$ is present, the rule (##) does become applicable – its third premise now becomes derivable since $A <_{c_1} B$ by (#). Therefore, there are two coercions from A to B: both $A <_{c_1} B$ and $A <_{c_2} B$ are derivable in $T[\mathcal{R}]$. As a consequence, $T[\mathcal{R}]$ is not a conservative extension of T: the T-judgement $c_1 = c_2 : (A)B$ is derivable in $T[\mathcal{R}]$, but not in T. □

The above example shows that the formulation in [13] does not exclude the 'bad' rules such as (##) above as basic subtyping rules which, if present, would destroy the key property of conservativity and lead to unreasonable behaviours of the coercive subtyping extension.

2.2. Coercive subtyping: an adequate formulation

As explained above, there are two problems with the previous treatments of coercive subtyping. One is that the notion of conservativity is not linked to the traditional notion of conservative extension and the other that the notion of 'basic subtyping rule' is too liberal that has failed to exclude the problematic rules.

To deal with the first problem, we shall introduce below, besides the coercive subtyping calculus $T[\mathcal{C}]$, the star-calculus $T[\mathcal{C}]^*$ that will help make the notion of conservativity clear.

To solve the second problem, it suffices to realise that, if we consider only subtyping *judgements*, rather than *rules*, the problem as demonstrated in Example 2.1 does not occur anymore. In other words, we consider the original formulation of coercive subtyping in [12] or that in Y. Luo's PhD thesis [20,10] where the original type theory is extended with a set $\mathcal{C}$ of subtyping judgements to form the coercive subtyping extension $T[\mathcal{C}]$. In such a formulation, the above problem does not occur and, as we show below, the proof method of [31] can be used (and further improved) to prove that the coercive subtyping extension is indeed conservative.

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

Let's now proceed with this new formulation. Assume T be a type theory specified in the logical framework LF[9] and $\mathcal{C}$ be a (possibly infinite) set of subtyping judgements of the form $\Gamma \vdash A <_c B : Type$. The coercive subtyping extension $T[\mathcal{C}]$ and the corresponding star-calculus $T[\mathcal{C}]^*$ are formally introduced as follows.

The system $T[\mathcal{C}]_0$ and coherence. The intermediate system $T[\mathcal{C}]_0$ is an extension of the original type theory T.

- Syntax: The syntax of $T[\mathcal{C}]_0$ is the same as that of T.
- Judgements: $T[\mathcal{C}]_0$ is extended with the new judgement form $\Gamma \vdash A <_c B : Type$.
- Rules: $T[\mathcal{C}]_0$ is extended with the following rule

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

  and the structural subtyping rules in Fig. 1 which state that the subtyping relation is congruent, transitive, and closed under substitution, and satisfies the rules of weakening and contextual equality.

It is straightforward to see that $T[\mathcal{C}]_0$ is a conservative extension of T because in this system the subtyping judgements do not contribute to any derivation of a judgement of any other form.

Remark. Compared with the earlier definition of $T[\mathcal{R}]_0$ in [13], we have added the weakening and contextual equality rules. □

The notion of coherence is to guarantee that the judgements in $\mathcal{C}$ are well-behaved and, in particular, the coercions in the coercive subtyping extension are unique between any two types. It can now be defined formally as follows.

Definition 2.2 *(Coherence).* The set $\mathcal{C}$ of subtyping judgements is coherent if the following hold for $T[\mathcal{C}]_0$:

1. If $\Gamma \vdash A <_c B : Type$, then $\Gamma \vdash A : Type$, $\Gamma \vdash B : Type$ and $\Gamma \vdash c : (A)B$.
2. $\Gamma \nvdash A <_c A : Type$, for any Γ, A and c.
3. If $\Gamma \vdash A <_c B : Type$ and $\Gamma \vdash A <_{c'} B : Type$, then $\Gamma \vdash c = c' : (A)B$.

[9] See Appendix A for a brief description of LF [11], the typed version of Martin-Löf's logical framework [28]. Type theories specified in LF may contain logical propositions and inductive types, some examples of which are given in Appendix B.

Congruence

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

Transitivity

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

Substitution

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

Weakening

$$\frac{\Gamma, \Gamma' \vdash A <_c B : \mathit{Type} \quad \Gamma, \Gamma'' \vdash \mathit{valid}}{\Gamma, \Gamma'', \Gamma' \vdash A <_c B : \mathit{Type}}$$

Contextual equality

$$\frac{\Gamma, x : K, \Gamma' \vdash A <_c B : \mathit{Type} \quad \Gamma \vdash K = K'}{\Gamma, x : K', \Gamma' \vdash A <_c B : \mathit{Type}}$$

Fig. 1. The structural subtyping rules of $T[\mathcal{C}]_0$.

Basic subkinding rule

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

Subkinding for dependent product kinds

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

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

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

Structural subkinding rules

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

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

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

$$\frac{\Gamma, \Gamma' \vdash K_1 <_c K_2 \quad \Gamma, \Gamma'' \vdash \mathit{valid}}{\Gamma, \Gamma'', \Gamma' \vdash K_1 <_c K_2}$$

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

Fig. 2. The subkinding rules in $T[\mathcal{C}]/T[\mathcal{C}]^*$.

The coercive subtyping extension $T[\mathcal{C}]$ and the star-calculus $T[\mathcal{C}]^*$. Let $\mathcal{C}$ be any set of subtyping judgements.

- $T[\mathcal{C}]$, the extension of T with coercive subtyping with respect to $\mathcal{C}$, is the system obtained from $T[\mathcal{C}]_0$ by adding the subkinding judgements of the new form $\Gamma \vdash K <_c K'$ and the rules in Figs. 2 and 3.
- $T[\mathcal{C}]^*$, the star-calculus with respect to $\mathcal{C}$, is the system obtained from $T[\mathcal{C}]_0$ by extending its syntax with terms of the form $M * N$ (and, hence, the sets of contexts and judgements are extended accordingly) and adding the subkinding judgements of the new form $\Gamma \vdash K <_c K'$ and the rules in Figs. 2 and 4.

The rules in Fig. 2 are those concerning the subkinding judgements:

Coercive application rules

$$(CA_1) \qquad \frac{\Gamma \vdash f \ : \ (x{:}K)K' \quad \Gamma \vdash k_0 \ : \ K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) \ : \ [c(k_0)/x]K'}$$

$$(CA_2) \qquad \frac{\Gamma \vdash f = f' \ : \ (x{:}K)K' \quad \Gamma \vdash k_0 = k_0' \ : \ K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) = f'(k_0') \ : \ [c(k_0)/x]K'}$$

Coercive definition rule

$$(CA_2) \qquad \frac{\Gamma \vdash f \ : \ (x{:}K)K' \quad \Gamma \vdash k_0 \ : \ K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) = f(c(k_0)) \ : \ [c(k_0)/x]K'}$$

Fig. 3. The coercive application/definition rules in $T[\mathcal{C}]$.

Coercive application rules in the star-calculus

$$(CA_1^*) \qquad \frac{\Gamma \vdash f \ : \ (x{:}K)K' \quad \Gamma \vdash k_0 \ : \ K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f * k_0 \ : \ [c(k_0)/x]K'}$$

$$(CA_2^*) \qquad \frac{\Gamma \vdash f = f' \ : \ (x{:}K)K' \quad \Gamma \vdash k_0 = k_0' \ : \ K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash f * k_0 = f' * k_0' \ : \ [c(k_0)/x]K'}$$

Coercive definition rule in the star-calculus

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

Fig. 4. The coercive application/definition rules in $T[\mathcal{C}]^*$.

- The basic subkinding rule lifts subtyping relations to subkinding.
- The three subkinding rules for dependent product kinds propagate the subkinding relations through the dependent product kinds.
- The structural subkinding rules are the counterpart of those in Fig. 1 for the subtyping relation, stating that the subkinding relation is congruent, transitive, and closed under substitution, and satisfies the rules of weakening and contextual equality.

The rules in Fig. 3 and 4 are the key rules of coercive application and coercive definition, as explained informally in Section 2.1, for $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$, respectively.

Note that, in the star-calculus, coercive applications are of the form $f * a$, with the positions for coercion insertion marked by the $*$-symbol.[10] This syntactic difference makes it possible for us to link the coercive subtyping extension to the traditional notion of conservativity: $T[\mathcal{C}]^*$ is a conservative extension of T, as to be summarised in the following subsection and formally shown in the next section.

Remark. It may be worth remarking that $T[\mathcal{C}]$ is the 'official' calculus for coercive subtyping, rather than its equivalent $T[\mathcal{C}]^*$, although the latter is a conservative extension in the traditional sense. This becomes apparent when we point out that *implicit coercions* are supported by the possible omission of coercions in $T[\mathcal{C}]$, but not completely by $T[\mathcal{C}]^*$. In $T[\mathcal{C}]$, the same term $f(a)$ may become well-typed although it is not well-typed in T. This directly reflects the idea of subtyping as (implicit) coercions. □

2.3. *Adequacy of the formulation*

The above formulation of coercive subtyping is adequate in the sense that it has the following three properties:

1. Coercive subtyping is a conservative extension.
2. Coercive subtyping is also a definitional extension, in a more general sense.
3. The formulation is general enough to encompass the old one by coercion rules.

We shall now give further explanations of each of the above.

[10] An explanation of syntax may be necessary. We sometimes use $M(N)$ to stand for the 'usual' notation MN for application. $M * N$ obeys the same conventions as the usual terms for application. For example, $M * N * P$ abbreviates $(M * N) * P$ and, if we want to apply M to $N * P$ in the star-calculus, we write $M * (N * P)$.

Coercive subtyping: a conservative extension. The above formulation of coercive subtyping in Section 2.2 does not suffer from the problem as explicated in Section 2.1. Intuitively, this is because we only allow the basic coercion *judgements* in $\mathcal{C}$, rather than coercion rules. This has excluded the 'bad' rules such as (##) in Example 2.1.

Assuming that T be the type theory UTT or Martin-Löf's type theory and $\mathcal{C}$ a set of subtyping judgements, we can show that the coercive subtyping extension is conservative. More precisely, the following hold if $\mathcal{C}$ is coherent:

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

This will be proved as Theorem 3.9(2) and Theorem 3.10(3) in Section 3.

Coercive subtyping: a definitional extension. In coercive subtyping, for any T-judgement that is $T[\mathcal{C}]$-derivable, coercions can always be correctly inserted to obtain an equivalent T-derivable judgement. This intuitively means that coercive subtyping is not only a conservative extension, but a definitional extension, in a more general sense.[11]

To make this precise, we need to introduce the intermediate system $T[\mathcal{C}]_{0K}$:

- $T[\mathcal{C}]_{0K}$ is the system obtained from extending $T[\mathcal{C}]_0$ by the subkinding judgement form $\Gamma \vdash K <_c K'$ and the inference rules in Fig. 2.

  Alternatively, $T[\mathcal{C}]_{0K}$ is the system obtained from $T[\mathcal{C}]$ by removing the rules in Fig. 3.

It is obvious that $T[\mathcal{C}]_{0K}$ is a conservative extension of T and $T[\mathcal{C}]_0$, since the subkinding judgements do not contribute to the derivation of a judgement of any other form in absence of coercive rules (this is the same reason that $T[\mathcal{C}]_0$ is a conservative extensions of T).

Now, assuming T be UTT or Martin-Löf's type theory and $\mathcal{C}$ a set of subtyping judgements, the coercive subtyping extension $T[\mathcal{C}]$ is definitional in the following sense: if $\mathcal{C}$ is coherent,

- $T[\mathcal{C}]^*$ is a conservative extension over $T[\mathcal{C}]_{0K}$;
- every $T[\mathcal{C}]^*$-derivation can be transformed into an 'equivalent' $T[\mathcal{C}]_{0K}$-derivation; and
- $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ are equivalent.

This will be proved as Theorem 3.9(1), Theorem 3.7(3) and Theorem 3.10(3) in Section 3, where the necessary transformations of derivations (called Θ, Θ^*, θ^* and θ) and the notion of equivalence between derivations are formally defined.

Generality of the formulation. The above formulation of $T[\mathcal{C}]$ is general enough to encompass the old one with coercion rules. In practice, the set $\mathcal{C}$ of coercion judgements can often be described by means of a set of coercion rules. For instance, besides other basic subtyping judgements in $\mathcal{C}$, one may declare that $\mathcal{C}$ is closed under some rules:

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

where $map(c)$ is the usual mapping function that lifts c to the lists. Could such descriptions be accommodated? Remember that we only allow coercion judgements as basic subtyping axioms, rather than subtyping rules. Is this a limitation? In fact, it is not: the above formulation $T[\mathcal{C}]$ is very general and does lead to an adequate formulation by means of rules, as shown by the following definition and the remarks below.

Definition 2.3. Let $\mathcal{R}$ be a set of subtyping rules (i.e., their conclusions are subtyping judgements).

- $\mathcal{C}_{\mathcal{R}}$ is defined to be the set of subtyping judgements that can be derived in the system consisting of the rules in T and $\mathcal{R}$ and those in Fig. 1.
- $T[\mathcal{R}]$ is defined to be $T[\mathcal{C}_{\mathcal{R}}]$ (and $T[\mathcal{R}]^*$ to be $T[\mathcal{C}_{\mathcal{R}}]^*$).

Note that the non-applicable rules in $\mathcal{R}$ do not make any contributions to the formation of $\mathcal{C}_{\mathcal{R}}$. For example, if the rule (##) in Example 2.1 is in $\mathcal{R}$, it does not contribute to $\mathcal{C}_{\mathcal{R}}$ since it is not applicable when $\mathcal{C}_{\mathcal{R}}$ is formed. It is straightforward to conclude that, if $\mathcal{R}$ is a set of subtyping rules that is coherent in the sense of [13], $\mathcal{C}_{\mathcal{R}}$ is coherent and $T[\mathcal{R}]$, as defined to be $T[\mathcal{C}_{\mathcal{R}}]$, has the nice properties as described above.

[11] The notion of definitional extension has been studied for first-order theories: a first-order theory is a definitional extension of another if the former is a conservative extension of the latter and any formula in the former is logically equivalent to its translation in the latter [7]. The notion of definitional extension for more general formal systems is worth being explored. We have made some initial attempts to understand the general concept of definitional extension in the context of type theories [17] and it is clear that the current work has laid a promising basis in this direction.

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

Fig. 5. Relationships between $T[\mathcal{C}]$, $T[\mathcal{C}]^*$ and $T[\mathcal{C}]_{0K}$.

3. Properties of coercive subtyping: formal proofs

In this section, we give an outline of our proof that the coercive subtyping extension is a conservative extension, and a definitional extension in a more general sense, of the original type theory under the assumption that the set of basic coercion judgements be coherent. The proof method is based on (and further improved from) that in [31]. Since the proof is rather sophisticated, we explain only its key elements. (See [32] for further details.)

3.1. Key ideas

In this section we study the transformations of derivations between the systems $T[\mathcal{C}]$, $T[\mathcal{C}]^*$ (as defined in Section 2.2) and $T[\mathcal{C}]_{0K}$ (as defined in Section 2.3) (cf., Fig. 5). The leading idea is to use the properties of these transformations, together with the fact that $T[\mathcal{C}]_{0K}$ is a conservative extension over T, to obtain our final conservativity result.

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

$$f : T_1 \to T_2$$

if f is a function from the T_1-derivations to T_2-derivations. We describe four *algorithms*, to be defined in Section 3.2, which are such functions.

- $\Theta : T[\mathcal{C}] \to T[\mathcal{C}]_{0K}$ and $\Theta^* : T[\mathcal{C}]^* \to T[\mathcal{C}]_{0K}$.

The algorithm Θ (Θ^*) replaces the derivations of $\Gamma \vdash K_0 <_c K$ in the premises of the coercive rules (CA_1), (CA_2) and (CD) in Fig. 3 ((CA_1^*), (CA_2^*) and (CD^*) in Fig. 4) by derivations of $\Gamma \vdash c : (K_0)K$ and replaces the coercive applications in their conclusions by several ordinary applications. (Other rules are not modified.)

More precisely, suppose for example that the derivation in $T[\mathcal{C}]$ ends with the coercive application rule (CA_1) in Fig. 3:

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

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

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

The other coercive application rule (CA_2) and the coercive definition rule (CD) can be dealt with in the same way.

The intended result is the insertion of appropriate coercions into the 'gaps' so that, if d is a $T[\mathcal{C}]$-derivation ($T[\mathcal{C}]^*$-derivation) of J, then $\Theta(d)$ ($\Theta^*(d)$) is an 'equivalent' $T[\mathcal{C}]_{0K}$-derivation (and actually a T-derivation if J is not a subtyping or subkinding judgement).

Now the question is how to extend the transformation to the whole derivation. It is natural to start from the leaves of the derivation tree, and move to the root (conclusion). When a coercive application or definition rule is met, the subkinding judgements are to be replaced by the derivation of the coercion terms and the rules modified as above. To make this plan work, certain difficulties must be solved. Most importantly, one needs to make sure that, after the insertion transformations, the premises of the rules that are applied after the transformed judgements will be matching (see below for further explanations).

- $\theta : T[\mathcal{C}]^* \to T[\mathcal{C}]$ and $\theta^* : T[\mathcal{C}] \to T[\mathcal{C}]^*$.

The algorithm θ replaces coercive applications of the form $f * a$ in $T[\mathcal{C}]^*$ by coercive applications $f\ a$ in $T[\mathcal{C}]$, simply by erasing all occurrences of the $*$-symbol. The algorithm θ^* replaces coercive applications in $T[\mathcal{C}]$-derivations by coercive applications in $T[\mathcal{C}]^*$ by inserting $*$ into appropriate places.

Note that, although transforming a $T[\mathcal{C}]^*$-derivation into a $T[\mathcal{C}]$-derivation is straightforward – we obtain $\theta(d)$ simply by erasing all of the $*$-symbol in d, the other direction (in defining θ^*) to insert the $*$-symbol is not: nothing in the syntax of $T[\mathcal{C}]$ permits us to distinguish directly coercive and ordinary applications and to guarantee that in different premises of every inference inside a $T[\mathcal{C}]$-derivation the $*$-symbol would be inserted at the same places (and hence the premises would be matching after insertion).

The existence of θ and θ^* and the fact that they are inverse functions show that $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ are equivalent (see Theorem 3.10 below).

One would expect also Θ to be the composition of θ^* and Θ^* but to assure this both θ^* and Θ^* have to be defined. The difficulty that must be taken into account is that the properties of θ^* and Θ^* cannot be established independently of those of Θ.

Major difficulties and solutions. The "conceptual core" of all difficulties is that the insertion depends on derivations. If an expression appears in two places, the above process might map it to two expressions of T that are not identical. For example, consider the rule

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

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

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

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

Coherence, as defined in Definition 2.2, is the key to solving *all* these problems. Suppose a gap in the same expression is filled with a coercion c at one point in a derivation, and a coercion d at another point. Then c and d may not be identical, but coherence is used to ensure that they are judgementally *equal* in T and that after filling the gaps the expressions are equal in T. Of course, to implement this, a carefully planned induction is needed.

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

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

An important part of the inductive proof includes the lemmas about *presupposed judgements*.[12] The reason is that the "common part" of the premises of a rule is a presupposed judgement of both. We prove that if Θ (or Θ^* or θ^*) is defined for certain derivation d of $\Gamma \vdash J$ then it is defined for the derivations of presupposed judgements of $\Gamma \vdash J$ even if these derivations may be longer. (The same for Θ^* and θ^*.) The proof of this uses an auxiliary induction on the number of steps of an extraction algorithm that permits to obtain a derivation of a presupposed judgement.

Another algorithm and lemma take care of the rules for substitution, weakening and contextual equality which make direct inductive proofs difficult. We define a canonical form of derivations, in which these rules only occur after an introduction of a subtyping judgement (from $\Gamma \vdash A <_c B \in \mathcal{C}$), and an algorithm $\mathbf{E}$ that moves these rules up to the "permitted" position. Then we show that if Θ is defined for d, it is also defined for the derivation $\mathbf{E}(d)$ in canonical form (similarly for Θ^* and θ^*). The proof uses induction on the number of elimination steps.

In this paper, when proving the properties of the coercive subtyping extension $T[\mathcal{C}]$, we have restricted T to be the known type theories such as UTT and Martin-Löf's type theory. The reason for this is to guarantee that the rule forms are

[12] The meaningfulness of a judgement (not its correctness or derivability) may presuppose the derivability of some other judgements. For example, the judgemental statement $a : A$ presupposes that the judgement $A : Type$ is derivable. As another example, $\Gamma \vdash k = k' : K$ presupposes the derivability of $\Gamma \vdash k : K$ and $\Gamma \vdash k' : K$.

preserved by coercion insertions, an aspect that was not sufficiently investigated in [31]. For example, let the derivation d in $T[\mathcal{C}]$ end by some rule r of T (r is not one of coercive rules), $d \equiv \dfrac{\dfrac{d_1}{J_1} \quad \cdots \quad \dfrac{d_n}{J_n}}{r(J_1,\ldots,J_n)}$, and $\Theta(d_1),\ldots\Theta(d_n)$ be defined. If $\Theta(d)$ is defined, it is supposed to end with an application of the same rule r. This assumes that, if J_i is a premise of an instance of r, then the conclusion of $\Theta(d_i)$ may be used as a premise of an instance of r. More precisely, some "adjustment" using the provable equalities of T may be permitted before r is applied, as we have seen in the previous subsection, but it has to be proven (by analysis of r) that the form of the conclusion of $\Theta(d_i)$ is appropriate.

Example 3.1. Consider, e.g., the elimination operator for an inductive type in UTT. Let it be, for simplicity, $\mathbf{E}_{Nat}$ for the type of natural numbers Nat (see Appendix B, repeated here):

$$\begin{aligned}\mathbf{E}_{Nat} \quad : \quad & \big(C{:}(Nat)Type\big)\\ & \big(c{:}C(0)\big)\big(f{:}(x{:}Nat)\big(C(x)\big)C\big(succ(x)\big)\big)\\ & (z{:}Nat)C(z)\end{aligned}$$

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

To take care of this and similar cases, the coherence at the kind level (i.e., non-derivability of the statements of the form $\Gamma \vdash K <_c K$) is used. The lemma with appropriate clause (Lemma 3.21) is used in the proof of the main theorem.[13] This is an illustration of the fact that the proofs *are* interconnected because this lemma uses the assumption that Θ is defined (to prove the absence of the judgements $\Gamma \vdash K <_c K$ in subderivations).

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

3.2. Main algorithms: definitions

In this subsection, the algorithms Θ, Θ^*, θ and θ^*, as discussed in Section 3.1, are formally defined (cf., Fig. 5). Before doing so, we shall first introduce some preparatory concepts – the notion of equality between judgements and that of presupposed judgement.

Notation. Let $\Gamma_1 \equiv x_1{:}K_1,\ldots,x_n{:}K_n$ and $\Gamma_2 \equiv x_1{:}K_1',\ldots,x_n{:}K_n'$. Then,

$$\Gamma \vdash \Gamma_1 = \Gamma_2$$

stands for the following list of n judgements:

$$\begin{aligned}&\Gamma \vdash K_1 = K_1'\\ &\Gamma,\ x_1{:}K_1 \vdash K_2 = K_2'\\ &\ldots\ \ldots\\ &\Gamma, x_1{:}K_1\ldots,x_{n-1}{:}K_{n-1} \vdash K_n = K_n'\end{aligned}$$

Definition 3.2 *(Equality between judgements).* Let S be a type theory. The notion of equality between judgements of the same form in S, notation $J_1 = J_2$ (with S omitted), is inductively defined as follows:

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

[13] This clause was first explicitly formulated by Marie-Magdeleine [23].

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

Remark. We note that the following inference rule is derivable:

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

Definition 3.3. Let $conc(d)$ denote the conclusion of derivation d. Given two derivations d_1 and d_2, we write $d_1 \sim_S d_2$ iff $conc(d_1) = conc(d_2)$ in S. (We often omit the index S when no confusions may occur.)

Lemma 3.4. *Relation $\sim_S$ is an equivalence relation.*

In the following sections, we shall consider only the systems T, $T[\mathcal{C}]_0$, $T[\mathcal{C}]_{0K}$, $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ as S in Definition 3.2. Notice that, because of the conservativity of $T[\mathcal{C}]_0$ and $T[\mathcal{C}]_{0K}$ over T, the relation $\sim$ in these two systems is defined via the equality in T.

Definition 3.5 *(Presupposed judgement).* The notion of *presupposed judgement* is inductively defined as follows.

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

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

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

$$\Theta(d) \equiv \frac{\dfrac{\Theta(d_1)}{(conc(\Theta(d_1)))}}{R(conc(\Theta(d_1)))}.$$

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

$$\Theta(d) \equiv \frac{\dfrac{\dfrac{\dfrac{\Theta(d_1)}{conc(\Theta(d_1))} \;\cdots\; \dfrac{\Theta d_k}{conc(\Theta(d_k))} \quad \dfrac{\textbf{?T}[\mathcal{C}]_{\mathbf{0K}}\textbf{-derivations}}{\textbf{Equalities}}}{= \textbf{-transitivity and context replacement}}}{J_1' \;\cdots\;\cdots\; J_k'}}{R(J_1' \;\cdots\;\cdots\; J_k')}$$

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

5. Suppose $d \equiv \dfrac{\overset{d_1}{\Gamma \vdash f:(x:M)N} \quad \overset{d_2}{\Gamma \vdash k:K} \quad \overset{d_3}{\Gamma \vdash K <_c M}}{\Gamma \vdash f(k):[c(k)/x]N}$.
Applying Θ to the derivations d_1, d_2, d_3, we get derivations

$$\Gamma_1 \overset{\Theta(d_1)}{\vdash} f_1:(x:M_1)N_1, \qquad \Gamma_2 \overset{\Theta(d_2)}{\vdash} k_2:K_2 \quad \text{and} \quad \Gamma_3 \overset{\Theta(d_3)}{\vdash} K_3 <_{c_3} M_3$$

Then $\Theta(d)$ is the derivation:

$$\frac{\begin{array}{c}\Theta(d_1)\\ \Gamma_1\vdash f_1:(x:M_1)N_1\end{array}\quad \dfrac{\dfrac{\dfrac{\begin{array}{c}co(\Theta(d_3))\\ \Gamma_3\vdash c_3:(K_3)M_3\end{array}\quad \dfrac{?_1}{\vdash\Gamma_1=\Gamma_3}}{\Gamma_1\vdash c_3:(K_3)M_3}\quad \dfrac{\dfrac{\begin{array}{c}\Theta(d_2)\\ \Gamma_2\vdash k_2:K_2\end{array}\quad \dfrac{?_2}{\vdash\Gamma_2=\Gamma_1}}{\Gamma_1\vdash k_2:K_2}\quad \dfrac{?_3}{\Gamma_1\vdash K_2=K_3}}{\Gamma_1\vdash k_2:K_3}}{\Gamma_1\vdash c_3(k_2):M_3}\quad \dfrac{?_4}{\Gamma_1\vdash M_1=M_3}}{\Gamma_1\vdash c_3(k_2):M_1}}{\Gamma_1\vdash f_1(c_3(k_2)):[c_3(k_2)/x]N_1}$$

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

6. The case where d ends in an instance of coercive application rule for equality and coercive definition rule is handled similarly.
7. It can be useful to do the "adjustment" of the premises using equalities in a deterministic way, for example, to make the expressions that occur in the premises more to the right equal to the leftmost occurrence (like Γ_2 and Γ_3 were made equal to Γ_1 above). We shall assume, for certainty, that this convention is applied to both Θ and Θ^* in the same way.

Definitions of $\theta : T[\mathcal{C}]^* \to T[\mathcal{C}]$ and $\theta^* : T[\mathcal{C}] \to T[\mathcal{C}]^*$.
The transformation θ erases the $*$-symbol from a derivation in $T[\mathcal{C}]^*$. Its well-definedness (and hence totality) can be trivially verified.

The transformation θ^* that inserts the $*$-symbol into derivations in $T[\mathcal{C}]$ has the same cases as the definition of Θ, except the following differences: coercion application rules in $T[\mathcal{C}]$ are not replaced by ordinary applications but by coercive applications of the form $f * a$ in $T[\mathcal{C}]^*$. No contextual equality rules, $=$-transitivity etc. are inserted.

3.3. Main results

In the following, we summarise the main results about the algorithms and their consequences concerning conservativity. The key points (the proof structures and key proof methods) have been highlighted in the earlier subsections and their detailed proofs can be found in [32].

The following two theorems summarise the properties of Θ and Θ^*, in whose proofs the systems $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ are treated in similar ways. The 'backbone' of each of the theorems is in their first two statements (1) and (2), which are the core parts of the main inductive hypothesis, as explained in the outline of the proof at the end of this section.

Theorem 3.6 *(Properties of Θ).*

1. *$\Theta : T[\mathcal{C}] \to T[\mathcal{C}]_{0K}$ is a total function.*
2. *If $conc(d) \equiv conc(d')$ in $T[\mathcal{C}]$ then $\Theta(d) \sim_T \Theta(d')$.*
3. *For any $T[\mathcal{C}]$-derivation d, $\Theta(d) \sim_{T[\mathcal{C}]} d$.*

Theorem 3.7 *(Properties of Θ^*).*

1. *$\Theta^* : T[\mathcal{C}]^* \to T[\mathcal{C}]_{0K}$ is a total function.*
2. *If $conc(d) \equiv conc(d')$ in $T[\mathcal{C}]^*$ then $\Theta^*(d) \sim_T \Theta^*(d')$.*
3. *For any $T[\mathcal{C}]^*$-derivation d, $\Theta^*(d) \sim_{T[\mathcal{C}]^*} d$.*
4. *If the judgement J is derivable in $T[\mathcal{C}]^*$ and it does not contain $*$ then it is not changed by Θ^*, $conc(\Theta^*(J)) \equiv J$.*

Note that the third statements of the above two theorems, Theorem 3.6(3) and Theorem 3.7(3), make clear what we mean by coercive subtyping being a definitional extension (this was discussed in Section 2.3, where the transformational algorithms Θ and Θ^* and the $\sim$-equivalence between derivations were left to be vague). This is reinforced by the following corollary which shows that Θ and Θ^* respect the equivalence relation $\sim$.

Corollary 3.8.

1. *If $d \sim_{T[\mathcal{C}]} d'$ then $\Theta(d) \sim_T \Theta(d')$.*
2. *If $d \sim_{T[\mathcal{C}]^*} d'$ then $\Theta^*(d) \sim_T \Theta^*(d')$.*

The conservativity of the star-calculus is given in the following Theorem 3.9. Note that Theorem 3.7(4) above is necessary for the proof of conservativity and that Theorem 3.9(2) follows from Theorem 3.9(1) because $T[\mathcal{C}]_{0K}$ is a conservative extension of T.

Theorem 3.9 *(Conservativity).*

1. *$T[\mathcal{C}]^*$ is a conservative extension of $T[\mathcal{C}]_{0K}$.*
2. *$T[\mathcal{C}]^*$ is a conservative extension of T.*

The following theorem shows that $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ are equivalent and its corollary that Θ is indeed the composition of Θ^* and θ^* (cf., the picture in Fig. 5).

Theorem 3.10 *(Equivalence between $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$).*

1. *$\theta^* : T[\mathcal{C}] \to T[\mathcal{C}]^*$ is a well-defined total function.*
2. *$\theta : T[\mathcal{C}]^* \to T[\mathcal{C}]$ is the inverse of θ^* (with respect to the identity of derivations).*
3. *The type theories $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ are equivalent.*

Corollary 3.11. *The composition $\Theta^* \circ \theta^*$ is defined and equal to Θ (with respect to the identity of derivations).*

3.4. Auxiliary algorithms/lemmas and outline of the main inductive proof

The proofs of the above results proceed by induction on the structure of derivations. The main induction uses several auxiliary algorithms and lemmas that we present below. The connections between the assertions of the main theorem are explained within the outline of the proof given at the end of this subsection.

We need to introduce certain canonical form of derivations in $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$ by removing, or restricting the uses of, certain structural rules.

Definition 3.12. We shall say that the derivation d of $T[\mathcal{C}]$ or $T[\mathcal{C}]^*$ is in canonical form, if the rules (*Weakening*), (*Contextual equality*) and (*Substitution*) in Fig. 1 and the substitution rules of LF Appendix A may occur only immediately after the introduction of the subtyping judgement:

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

This definition applies without modification to the derivations in the subsystems of $T[\mathcal{C}]$ and $T[\mathcal{C}]^*$, such as $T[\mathcal{C}]_{0K}$ and T.

The auxiliary algorithms/lemmas. Conceptually, to prepare the main induction, we need, on the one hand, to define the algorithms that will simplify derivations, either reducing them to canonical form (algorithm **E**) or extracting derivations with simpler final judgement (presupposition algorithm *pre*), and establish their properties, and on the other hand to extend the coherence property from subtyping to subkinding.

Algorithm 3.13. There is an algorithm **E** that transforms every derivation d in the calculus $T[\mathcal{C}]$ ($T[\mathcal{C}]^*$) into a canonical derivation of the same judgement. If d is in $T[\mathcal{C}]_{0K}$ ($T[\mathcal{C}]_0, T$) then $\mathbf{E}(d)$ is in $T[\mathcal{C}]_{0K}$ ($T[\mathcal{C}]_0, T$).

The algorithm **E** works by “moving” the rules in question up along the branches of the derivation tree. The only obstacle is the subtyping introduction rule because its conclusion corresponds to the predefined elements of $\mathcal{C}$ and should not contain coercive applications.

Complete definition and proof of the correctness of this algorithm involves many technical subtleties that we cannot discuss in detail here. For example, we use certain equivalent formulations of the rules to be moved (with extra premises). We need to proceed in certain order: we begin always at the topmost rule; weakening has to be moved first, then substitution rules, and contextual equality the last.

Now we will consider the extraction of derivations of presupposed judgements. The input data of the extraction algorithm consist of a derivation in canonical form, (i.e., after application of **E**), and the presupposed judgement whose derivation is to be obtained. Admitting some redundancy, one may take a “source” judgement, its derivation, and a presupposed judgement of the former as the arguments of this algorithm.

Algorithm 3.14 *(Pre, presupposition algorithm).* There exists an algorithm that permits to obtain from canonical derivations of a judgement J the canonical derivations of all its presupposed judgements described in Definition 3.5. We shall write $pre(J, d)$ for the derivation obtained by the algorithm for the presupposed judgement J of d.

The proof of the existence of the presupposition algorithm proceeds by induction on derivations of J (the extraction algorithm is defined simultaneously by recursion). For example, consider

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

and let

$$J_1 \equiv \Gamma \vdash (x{:}K_1)K_2 \ kind, \qquad J_2 \equiv \Gamma, x{:}K_1 \vdash K_2 \ kind$$
$$J_1' \equiv \Gamma \vdash \big(x{:}K_1'\big)K_2' \ kind, \qquad J_2' \equiv \Gamma, x{:}K_1 \vdash K_2' \ kind$$

Then, we have

$$pre(J_1, d) \equiv \frac{\Gamma, x{:}K_1 \overset{pre(J_2,d_1)}{\vdash} K_2 \ kind}{\Gamma \vdash (x{:}K_1)K_2 \ kind}$$

and

$$pre\big(J_1', d\big) \equiv \frac{\mathbf{E}\big(\frac{\Gamma, x{:}K_1 \overset{pre(J_2',d_1)}{\vdash} K_2' \ kind \qquad \Gamma \overset{d_2}{\vdash} K_1 = K_1'}{\Gamma, x{:}K_1' \vdash K_2' \ kind}\big)}{\Gamma \vdash (x{:}K_1')K_2' \ kind}.$$

Lemma 3.15 *(Elimination of transitivity of subkinding in $T[\mathcal{C}]_{0K}$). There is an algorithm, transforming every canonical derivation of the judgement $\Gamma \vdash K <_c K'$ in $T[\mathcal{C}]_{0K}$ into a canonical derivation of the judgement $\Gamma \vdash K <_{c'} K'$ in the same calculus not containing structural rules for subkinding (in particular, transitivity) such that $\Gamma \vdash c = c' : (K)K'$ in T.*

To prove this lemma, a rank of dependent product kinds is defined. The proof goes by induction on the rank. (The algorithm **E** is needed within.)

Corollary 3.16. *If $\mathcal{C}$ is coherent, then coherence holds in $T[\mathcal{C}]_{0K}$ in the sense that*:

1. *if $\Gamma \vdash K = K'$, then $\Gamma \vdash K <_c K'$ is not derivable*;
2. *if $\Gamma \vdash K <_c K'$ and $\Gamma \vdash K <_{c'} K'$ then $\Gamma \vdash c = c' : (K)K'$.*

In all lemmas and theorems below we assume that $\mathcal{C}$ is coherent. The formulations are given only for transformation Θ, but in fact similar lemmas are proved for Θ^* and θ^* (the calculi have to be modified accordingly).

The following lemma is proved by structural induction on derivations.

Lemma 3.17. *Let dbe a derivation, and suppose $\Theta(d)$ is defined. Then*:

1. *$\Theta(d)$ ends with the same rule with d and the last judgement of $\Theta(d)$ and d are of the same form.*
2. *If d_0 is a sub-derivation of d, then $\Theta(d_0)$ is defined.*
3. *Let d_0 be a sub-derivation of d, d_0' be another derivation of conc(d_0), $\Theta(d_0')$ be defined and $\Theta(d_0) \sim \Theta(d_0')$. If d' is obtained from d by replacing d_0 with d_0' then $\Theta(d')$ is defined and $\Theta(d) \sim_T \Theta(d')$.*

The proof of two lemmas below use structural induction and that on the number of steps of the corresponding algorithm on a given input. The proof of the second lemma uses the first.

Lemma 3.18. *Let d be a derivation in $T[\mathcal{C}]$ and* **E** *the algorithm defined in Algorithm 3.13. If $\Theta(d)$ is defined, then $\Theta(\mathbf{E}(d))$ is, and $\Theta(\mathbf{E}(d)) \sim_T \Theta(d)$.*

Lemma 3.19 *(Presupposition lemma). Let d be a derivation in canonical form, and suppose $\Theta(d)$ is defined. Let the algorithm pre be applied to d with presupposed judgement J of conc(d) as second argument. Then $\Theta(pre(d, J))$ is defined and conc($\Theta(pre(d, J))$) $\sim_T$ conc($pre(\Theta(d), \Theta_d(J))$).*

The following Lemma 3.20, which is purely technical and whose proof is done by structural induction on derivations in canonical form, is mentioned because of its use in the proof of Lemma 3.21 below that plays a crucial role in the main inductive proof.

Lemma 3.20 *(Product equality). In system $T[\mathcal{C}]$, the following hold*:

1. *If $\Gamma \vdash (x{:}K_1)K_2 = M$, then there are terms N_1, N_2, such that $M \equiv (x{:}N_1)N_2$.*
2. *If $\Gamma \vdash (x{:}K_1)K_2 = (x{:}N_1)N_2$, then $\Gamma \vdash K_1 = N_1$ and $\Gamma, x{:}K_1 \vdash N_1 = N_2$.*

Lemma 3.21. *Suppose d_1 and d_2 are two derivations in canonical form, and $\Theta(d_1)$ and $\Theta(d_2)$ are defined.*

1. *If d_1 and d_2 are both derivations of $\Gamma \vdash$ valid, then*
$$\Theta(d_1) \sim_T \Theta(d_2).$$
2. *If d_1 and d_2 are both derivations of $\Gamma \vdash K$ kind, then*
$$\Theta(d_1) \sim_T \Theta(d_2).$$
3. *If d_1 is a derivation of $\Gamma \vdash k{:}K_1$, d_2 is a derivation of $\Gamma \vdash k{:}K_2$, then*
$$\Theta(d_1) \sim_T \Theta(d_2).$$
4. *If d is a derivation of $\Gamma \vdash K_1 <_c K_2$ and $\Theta(d)$ is defined, then the judgement $\Theta_d(\Gamma) \vdash \Theta_d(K_1) = \Theta_d(K_2)$ is not derivable in T, where $\Theta_d(\Gamma)$, $\Theta_d(K_1)$, $\Theta_d(K_2)$ denote the corresponding parts of the judgement conc$(\Theta(d))$.*
5. *If d_1 is a derivation of $\Gamma \vdash K_1 <_{c_1} K_2$ and d_2 is a derivation of $\Gamma \vdash K_1 <_{c'} K_2$ in $T[\mathcal{C}]$ and $\Theta(d_1)$, $\Theta(d_2)$ are defined then the equality $\Theta_{d_i}(\Gamma) \vdash \Theta_{d_1}(c) = \Theta_{d_2}(c'){:}(\Theta_{d_i}(K_1))\Theta_{d_i}(K_2)$ is derivable in T. Here Θ_{d_i} ($i = 1, 2$) are defined as Θ_d above.*

The proof goes by induction on the *sum* of the sizes of two derivations. Notice especially the last two statements. To obtain their proof, the assumption that Θ (or Θ^*) is defined is used together with coherence of $T[\mathcal{C}]_{0K}$ established in Corollary 3.16.

The outline of the main inductive proof. The proof of Theorem 3.6(1,2), that Θ is total and that the final judgement of derivations obtained by its application to the derivations of the same judgement is defined up to equality of judgements in T, proceeds by induction on the size of derivations. The third clause, Theorem 3.6(3), may be included as part of the inductive hypothesis, but it is not necessary for the proof of the first two clauses Theorem 3.6(1,2).

The proof of Theorem 3.7 concerning Θ^* is similar and may be done "in parallel" (the proofs do not depend on each other). In particular, the clauses (3) and (4) may be included as parts of the inductive hypothesis.

The main idea used in the inductive step is to combine the presupposition Lemma 3.19 and Lemma 3.21. We observe, that matching of the premises (more precisely, certain parts of the premises) after the application of Θ (and similarly Θ^*) may be viewed as matching of certain presupposed judgements. For example, in case of premises $\Gamma \vdash K = K'$ and $\Gamma \vdash K' = K''$ (premises of transitivity of equality rule with derivations d and d') there is the common presupposed judgement $\Gamma \vdash K'$ *kind*. It may be changed differently by Θ applied to d and d' but Lemma 3.21 assures that the results are equal in T.

In general the proof of the inductive step (i.e., the proof that if the derivation d ends by some rule r with $d_1, \ldots, d_k$ being subderivations of the premises then the assumption that the $\Theta(d_1), \ldots, \Theta(d_k)$ are defined implies that $\Theta(d)$ is defined) may be reduced to the question of T-equality between certain presupposed judgements of the premises modified by Θ.

The clauses (4) and (5) of the Lemma 3.21 play the following roles:

- (4) is necessary to assure that there is no coercion insertions that destroy the structure of the premises in a "non-adjustable" way, as discussed in Example 3.1;
- (5) is necessary to assure that the "adjustment" using T-equalities is possible.

The proofs of Theorem 3.9 to Corollary 3.11 are sketched as follows.

- Theorem 3.9, conservativity of $T[\mathcal{C}]^*$ over $T[\mathcal{C}]_{0K}$ and T, is an easy consequence of Theorem 3.7(4). Indeed, let us take an arbitrary judgement without $*$. If d is its derivation in $T[\mathcal{C}]^*$ then by this clause $\Theta^*(d)$ is the derivation of the same judgement in $T[\mathcal{C}]_{0K}$ or in T (depending only on its form). Also, the derivability of J in $T[\mathcal{C}]_{0K}$ or in T implies its derivability in $T[\mathcal{C}]^*$.
- The proof of Theorem 3.10 follows roughly the same "template" as in Theorems 3.6 and 3.7, with the following differences to be noted:
  - The proof uses Theorem 3.6(1, 2) and Lemma 3.21(4) to verify that the premises will remain matching after the insertion of $*$.
  - Because of this, the insertion of equality rules to "adjust" the premises is not necessary.
- The proof of Theorem 3.10(2, 3) is obtained now from Theorem 3.10(1) and the obvious properties of θ.
- The proof of Corollary 3.11 is obtained by a separate structural induction using all three theorems.

4. Subtyping for type theories with canonical objects

Coercive subtyping is not the first notion one would consider when introducing subtyping into a type theory. Instead, one usually first considers *subsumptive subtyping*, the notion of subtyping that corresponds to the notion of inclusion in set theory and is represented by the subsumption rule. In this section, we analyse these two notions of subtyping and show that, for the type theories with canonical objects, coercive subtyping provides us an adequate framework, while subsumptive subtyping does not. In doing this, we shall also illustrate that coercive subtyping is rather general with many interesting applications.

4.1. Two views on typing and subtyping

What is a type? What is a subtype? In the literature, there are two different views of types which in turn give rise to two notions of subtyping. One of the views of types, as often found in the study of programming languages, is that of *type assignment* (cf., the type assignment systems employed in functional programming languages such as ML and Haskell). Under this view, objects and types exist independently and types are assigned to objects. 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 of the objects of A are also objects of B. In other words, the view of type assignment is typically associated with the notion of subtyping characterised by means of the so-called subsumption rule:

$$(\mathrm{SUB}) \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \leqslant B}{\Gamma \vdash a : B}$$

Let us call such systems as systems with *subsumptive subtyping*.

Another view of types 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. Under this view, most of the types are considered as inductive, consisting of their canonical objects, and the objects and their types depend on each other and do not exist independently. For example, the type *Nat* of natural numbers consists of the canonical numbers 0 and $succ(n)$ and the natural numbers only exist because they are objects of *Nat*. This view of canonical objects is the basis to consider inductive types in dependent type theories, each of which is equipped with an induction principle (elimination rule) expressing that, in order to prove a property for all objects of the inductive type, one only has to prove it for all of its canonical objects. We may call such type theories as *type theories with canonical objects* (cf., footnote 2 in Introduction for examples of such type theories), which have the following property:

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

This property of canonicity justifies the induction principles for inductive types in a type theory with canonical objects.

Is it possible to employ subsumptive subtyping for a type theory with canonical objects? Unfortunately, the answer is no. If a type is considered as consisting of its canonical objects, it is difficult to see how subtyping could be understood or introduced by means of the subsumption rule (SUB). To do so, one would have to answer the questions like:

1. Would the canonical objects of a subtype be canonical objects of a supertype?
2. How would induction principles be formulated to take care of the objects introduced by subtyping?

Such considerations lead to difficulties: subsumptive subtyping is incompatible with the idea of canonical object in the sense that it cannot be employed for a type theory with canonical objects without destroying canonicity.[14]

4.2. Limitation of subsumptive subtyping

The subsumption rule (SUB) says that, if A is a subtype of B, then every object of type A is also of type B. However, if T is a type theory with canonical objects (e.g., Martin-Löf's type theory), extending it with the subsumption rule would be problematic – canonicity would be lost. This is a very unpleasant situation, as illustrated by the following two examples.

Structural subtyping for inductive types. Structural subtyping is a natural subtyping relation for an inductive type and has been studied for arbitrary inductive types in the framework of coercive subtyping [20,19]. However, it is incompatible with subsumptive subtyping.

[14] In type theory, canonicity is in general expressed by means of the elimination rules for inductive types – see the elimination operators $\mathbf{E}_I$ for several inductive types I in Appendix B. However, one might take different forms of elimination operators for some types. For example, in ECC [11], Π-types employ application as their elimination operator and Σ-types the projection operators (and, similarly for the Π-types in Coq). In such cases, the resulting system with full cumulativity has expected good properties such as canonicity. However, this is in general not the case, as discussed here.

Consider the example of lists (see Appendix B for the type constructor $List(A)$). The structural subtyping relationship for lists is that, if A is a subtype of B, then $List(A)$ is a subtype of $List(B)$. This would be expressed by means of the following rule:

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

In a framework of subsumptive subtyping, canonicity would be lost with this rule. To see this, let A and B be closed types such that $A < B$. Then $nil(A) : List(A)$ would be of type $List(B)$. But $nil(A)$ is not definitionally equal to any canonical object of $List(B)$, $nil(B)$ or $cons(B, b, l)$.

This has many unpleasant consequences. For example, for a propositional equality $Eq(T)$ over any type T, we can prove the following proposition by means of the elimination rule for $List(B)$:

$$Eq\big(List(B), x, nil(B)\big) \ \vee\ \exists b{:}B\exists l{:}List(B).\ Eq\big(List(B), x, cons(B, b, l)\big),$$

where x is an arbitrary object of type $List(B)$. Taking x to be $nil(A)$, we get

$$Eq\big(List(B), nil(A), nil(B)\big) \ \vee\ \exists b{:}B\exists l{:}List(B).\ Eq\big(List(B), nil(A), cons(B, b, l)\big).$$

But neither of the disjuncts is the case definitionally. Therefore, the property of equality reflection does not hold: there is a mismatch between the definitional and propositional equalities (even for closed terms).

Projective subtyping. Projective subtyping is based on ‘projections’, from a type of pairs (or a record type) to a component type. For example, let $P : (Nat)Prop$ be a predicate over Nat. The Σ-type $\Sigma(Nat, P)$ is sometimes used to represent the subtype of Nat of those natural numbers n such that $P(n)$ holds. (See Appendix B for formal specifications of Nat and Σ-types.) Now, it would be natural to consider the following subtyping relation:

$$\Sigma(Nat, P) < Nat.$$

However, considered as subsumptive subtyping, this destroys canonicity. For example, consider the following term:

$$t \equiv pair(Nat, P, 0, p_0) : \Sigma(Nat, P).$$

t is not definitionally equal to either 0 or $succ(n)$ for any n. Furthermore, we can prove from the elimination rule for Nat that

$$Eq(Nat, t, 0) \ \vee\ \exists x{:}Nat.\ Eq\big(Nat, t, succ(x)\big),$$

but neither of the disjuncts is true definitionally (as in the above example).

Here is the general situation, summarising the above two examples. Let I and J be closed inductive types which are different (i.e., I is not definitionally equal to J) and $I < J$. Assume that i be a closed object of type I. By the subsumption rule, $i : J$. However, i is not definitionally equal to any of the canonical objects of J! In other words, we would have a closed object of an ‘inductive’ type which is not equal to any of the canonical objects of the type – canonicity fails to hold.

4.3. *Coercive subtyping: adequacy and generality*

Coercive subtyping is a suitable subtyping framework for type theories with canonical objects. As compared with subsumptive subtyping, coercive subtyping does not introduce new objects into a type. In the framework of coercive subtyping, $A < B$ means that there is a (unique) coercion $c : (A)B$ that maps any object of A to an object of B. This is consistent with the idea of canonical object – if B is an inductive type, we do not need to change its elimination rule, since B will still have the same objects even if $A < B$.

Coercive subtyping has been studied for introducing various useful notions of subtyping, including structural subtyping and projective subtyping, as studied in Section 4.2. For instance, for *structural subtyping* of lists, we have the following coercive subtyping rule:

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

where $map(c)$ is the ‘usual’ function that intuitively maps a list $[a_1, \ldots, a_n]$ to $[c(a_1), \ldots, c(a_n)]$. In general, structural subtyping can be introduced for all of the inductive types covered by the general inductive schemata (all inductive types as in Martin-Löf's type theory or as implemented in the proof assistants such as Agda and Coq) and, furthermore, the property of coherence and that of transitivity elimination hold [20,19]. For *projective subtyping* for Σ-types (or types of pairs in the non-dependent case), one may use the first projection π_1 or the second projection π_2 (but not both, in order to guarantee coherence [10]) as coercions. For instance, we may have

$$(\Sigma_{\pi_1}) \qquad \frac{A : Type \quad B : (A)Type}{\Sigma(A, B) <_{\pi_1(A,B)} A}$$

This coercive subtyping relation was heavily used in proof development (see, for example, [3]).

Projective subtyping is an example of *non-structural subtyping* relations that the framework of coercive subtyping can accommodate. There are many other interesting ways to introduce non-structural coercions in applications. For instance, we may introduce the following subtyping relation:

$$(\xi) \qquad \frac{\Gamma \vdash A : Type \quad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{1}(A, a) <_{\xi_{A,a}} A : Type}$$

where $\mathbf{1}(A, a)$ is the unit type that has only one object $*(A, a)$ and $\xi_{A,a}(x) = a$ for any $x : \mathbf{1}(A, a)$. Such a subtyping relation may look rather strange, but it is very useful. For example, with the help of the above rule (ξ), we can express manifest fields in a type of modules (a Σ-type or a dependent record type) without the need to employ any extensional features in type theory [14].

4.4. Coercions in a logical framework

We have studied coercive subtyping in the logical framework LF. Although some proof assistants such as Plastic [4] implement logical frameworks, most of the proof assistants implement type theories directly, without implementing a logical framework. For example, in Coq, the Π-types directly implemented: a Π-type `(x:A)B` in Coq corresponds to $\Pi(A, [x:A]B)$ in LF. Because of this difference, the coercion mechanism we have studied (and implemented in Plastic – see the next section) provides a more general tool than those based on a direct syntax (cf., implementations of coercions in Coq [5,30] and Lego [21,3]). In particular, several forms of coercions useful in practice, as studied by Bailey in his PhD thesis [3], can be captured by our coercion mechanism.

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

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

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

$$app\big(A, [x{:}A]B, f, a\big) = Elim_{\Pi}\big(A, [x{:}A]B, \big[G{:}\Pi\big(A, [x{:}A]B\big)\big]B[a], \big[g{:}(x{:}A)B\big]g(a), f\big).$$

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

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

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

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

for all groups G and for all elements of G, $\ldots\ \ldots$

Formally, this is represented as

$$\Pi G : Group\ \Pi x{:}G.\ \ldots\ \ldots$$

But this is ill-typed: G is not a type! For such applications, Bailey [3] has considered the so-called type coercions that convert non-types (e.g., G) to types (e.g., G's carrier type $El(G)$). In a direct syntax, this has to be introduced separately from the argument coercion mechanism. However, when we consider coercions based on the logical framework, the above term is the following in LF:

$$\Pi\big(Group,\ [G{:}Group]\Pi(G,\ \ldots)\big),$$

which is equal to (by coercive subtyping)

$$\Pi\big(Group,\ [G{:}Group]\Pi\big(El(G),\ \ldots\big)\big),$$

where $Group <_{El} U$.[15] Therefore, the type coercions are just special cases of argument coercions in the logical framework.

[15] U is a type universe. We omit some of the technical details here.

Function coercions. This is another kind of coercions that convert from functions with a type-mismatch to type-matching ones or even from non-functions into functions. For example, if the application $f\ a$ is not well-typed and there is no argument coercion that can be inserted to get it well-typed, we may coerce f into a function whose domain is the type of a, to get the well-typed term $(c\ f)\ a$. In an LF-based syntax, this is to coerce

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

into

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

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

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

5. Implementation of coercive subtyping

Coercive subtyping has been implemented in various degrees in several proof assistants such as Coq [5,30], Lego [21,3], Matita [25] and Plastic [4]. As mentioned above, in a proof assistant like Coq that implements the direct syntax of a type theory, only argument coercions are supported, while in a proof assistant like Plastic that implements a logical framework, the coercion mechanism supports other forms of coercion insertion besides argument coercions (see Section 4.4).

Based on the implementation of coercive subtyping in Plastic, the third author has implemented the newly improved formulation, as described in Section 2.2, and improved the original implementation. This will be briefly reported below.[16]

Coercive subtyping has been used in many applications, including its use for notational abbreviation in proof development. Recently, the first author has developed the type-theoretical semantics with coercive subtyping [15,16,18]. Based on the above implementation of coercive subtyping, dot-types have also been implemented [33] and we shall use that to give examples to illustrate the implementation.

5.1. *Implementation of coercive subtyping in Plastic*

Plastic implements the logical framework LF and, in particular, the type theory UTT as presented in Chapter 9 of [11] and coercive subtyping. In Plastic, type-checking a term of the application form $f(a)$, where f is of kind $(x{:}K_1)K_2$ and a of kind K_1', the following cases are considered:

- if K_1' is convertible (i.e., computationally equal) to K_1, then $f(a)$ is well-typed and of kind $[a/x]K_1$;
- if K_1' is not convertible to K_1 and there is a coercion c such that, in the current context, $K' <_c K$ and K' and K are convertible to K_1' and K_1, respectively, then $f(a)$ is well-typed, of kind $[c(a)/x]K_1$, and equal to $f(c(a))$;
- otherwise, $f(a)$ is not well-typed.

Remark. Note that the implementation of coercions in Plastic is different from the Coq system in that convertibility is automatically considered when deciding whether there exists a coercion between two types. □

5.1.1. *Coercion declarations*

The syntax of declaring a coercion in Plastic is as follows:

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

where `term` is for the term of the coercion to be specified with type `type`, which is optional. `Parameters` and `Prerequisites` are also optional and can be used to define parameterised coercions and coercion rules (see examples below). The latter one requires some other coercion assumptions as prerequisites. Once a user defines a coercion, the system will give it a name with `cx` as a prefix. The names will be `cx1`, `cx2`, etc.

[16] For the details of the implementation of coercive subtyping and dot-types, see the forthcoming thesis by Xue [32].

In Plastic, users can define coercions to handle any combination of the following cases:

- plain coercions of kind $(A)B$ (i.e., $A \to B$, in Plastic's notation);
- dependent coercions of kind $(x{:}A)B$[17];
- parameterised coercions: such coercions are families of coercions parameterised by some variables; and
- coercion rules stating that, if some subtyping premises hold, so does the conclusive subtyping relation.

The following gives an example for each of the above cases.

Example 5.1.

1. Plain coercions. If we want to define a plain coercion c from Man to Human, we could simply write:

```
> [c : Man -> Human ];
> Coercion = c : Man -> Human;
```
or simply

```
> Coercion = c;
```

2. Dependent coercions. We can define a function lv from $List(Nat)$ to $Vec(Nat, n)$ (n is the length of the vector) as follows:

$$lv\big(nil(Nat)\big) = vnil(Nat)$$
$$lv\big(cons(Nat, x, l)\big) = vcons\big(Nat, len(Nat, l), x, lv(l)\big)$$

where $len : (A{:}Type)(l{:}(List(A)))Nat$ gives the length of a list of type $List(A)$:

$$len\big(A, nil(A)\big) = zero$$
$$len\big(A, cons(A, x, l)\big) = succ\big(len(A, l)\big)$$

We can define lv as a coercion:

```
> Coercion = lv;
```

3. Parameterised coercions. We can define a function vl from $Vec(Nat, n)$ to $List(Nat)$:

$$vl\big(vnil(Nat)\big) = nil(Nat)$$
$$vl\big(vconst(Nat, n, x, v)\big) = cons\big(Nat, x, vl(v)\big)$$

as a parameterised coercion, with parameter $n{:}Nat$.

```
> Coercion
> Parameters [n:\mathit{Nat}]
> = vl n;
```

4. Coercion rules. For example, the rule (L_c) in Section 4.3, saying that, if there is a coercion c from A to B, then we can have a coercion $map(c)$ from $List(A)$ to $List(B)$, where map is the function from $(A)B$ to $(List(A))List(B)$ defined as follows:

$$map\big(A, B, nil(A)\big) = nil(B)$$
$$map\big(A, B, cons(A, x, l)\big) = cons\big(B, f(x), map(A, B, l)\big)$$

This coercion can be defined as follows:

```
> Coercion
> Parameters [A,B:\mathit{Type}][f:A->B]
> Prerequisites f
> = map A B f : List A -> List B;
```

[17] Dependent coercions are coercions whose kinds are dependent product kinds. See [22] for further details.

One may consider how the specified coercions would work together, as the following example shows.

Example 5.2. We assume that we have specified the coercion `c` from *Man* to *Human* and the coercion rule for lists, as in 1 and 4 in the above example. Plastic will generate internal names for them, say `cx1` for the coercion `c` and `cx2` for the coercion rule.

With any given types A and B and coercion k from A to B, $\mathtt{cx2}(A, B, k)$ is a coercion from $List(A)$ to $List(B)$. So $\mathtt{cx2}(Man, Human, \mathtt{cx1})$ is a coercion from $List(Man)$ to $List(Human)$. Hence, for example, if $l_m : List(Man)$ is a list of men, then $len(Human, l_m)$ is still well-typed and equal to $len(Human, \mathtt{cx2}(Man, Human, \mathtt{cx1})(l_m))$ – the coercion is inserted as expected. □

5.1.2. Remarks on coherence

In Plastic, as in other systems like Coq, coherence is checked for plain coercions: when a new coercion c from A to B is introduced, where c is either newly specified or generated by transitivity because of the introduction of other coercions, it checks whether there is already a coercion from A' to B' which are convertible to A and B, respectively. If no such a coercion exists, we accept the new coercion. If there exists such a coercion c', we check whether the two coercion terms c and c' are convertible: if they are, we will do nothing with it[18] and, if they are not, the coercion c is rejected.

Remark. If the newly introduced coercion is generated by transitivity, when it is rejected, we will reject the term causing this transitivity as well. For example, if we have $A <_{c_1} B$, $A <_{c_2} C$ and we want to introduce $B <_{c_3} C$. By transitivity, we could generate a new coercion $[x{:}A]c_3(c_1 x)$ from A to C, but c_2 is already a coercion from A to C and they are not convertible. We will not only reject $[x{:}A]c_3(c_1 x)$, but also reject coercion c_3. □

When parameterised coercions are specified or when coercion rules are used to introduce coercions, coherence checking is undecidable in general. Therefore, we need to show that, for example, certain coercion rules are coherent and hence can be used in practice. Examples include those discussed in Section 4.3 and those about dot-types as discussed below.

5.2. Examples with dot-types

Inductive data types can be specified in Plastic and coercions can be introduced between them. For example, we can introduce the first projection for Σ-types as a parameterised coercion: $\Sigma(A, B) <_{\pi_1(A,B)} A$. In this subsection, rather than showing how to introduce coercions for these inductive data types, we will informally show our implementation for a new kind of data type, dot-types, based on the implementation of coercive subtyping.

Dot-types are proposed by Pustejovsky in his Generative Lexicon Theory [29]. It has been found difficult to formalise the notion of dot-types (see, for example, [1]). A type-theoretic treatment is given in [15] where coercive subtyping is essentially employed to capture the notion of dot-type. In this subsection, we shall briefly explain how dot-types are implemented in Plastic based on the implementation of coercive subtyping.

Informally, a dot-type $A \bullet B$ is a type of pairs with the following two attributes:

– $A \bullet B$ can be formed if A and B do not share *components*, and
– $A \bullet B$ is subtype of *both* A and B.

According to the first, two types can only form a dot-type if they do not share components. Informally, the notion of component of a type A is defined as follows: (1) if A is not equal to a dot-type, its components are its supertypes (including itself), and (2) if A is equal to $A_1 \bullet A_2$, then a component of A is either a component of A_1 or a component of A_2. For example, $A \bullet A$ is not a dot-type, because its constituent types are the same type. $A \bullet (A \bullet B)$ is not a dot-type, because its constituent types A and $A \bullet B$ share the component A.

The objects of a dot-type $A \bullet B$ are of the form $\langle a, b\rangle$, where $a : A$ and $b : B$. There are associated projection operators p_1 and p_2 so that $p_1\langle a, b\rangle = a$ and $p_2\langle a, b\rangle = b$. This makes the dot-type $A \bullet B$ very much like the product type $A \times B$, but with the difference that A and B cannot share components. An important feature of dot-types (the second attribute in the above) is that both projections p_1 and p_2 are coercions; that is $A \bullet B <_{p_1} A$ and $A \bullet B <_{p_2} B$. Because that the constituent types of a dot-type do not share components, taking both projections as coercions is coherent.[19] The formal details, including the definition of component and the inference rules for dot-types, can be found in [15].

Dot-types are not ordinary inductive data types. As we have explained above, for $A \bullet B$ to be a dot-type, the constituent types A and B should not share components. In an implementation of dot-types, this special condition of type formation must be checked and adhered to. In order to make sure of this, we have to implement the dot-types as special data types,

[18] That is, c' will be taken as the representative coercion. In other words, in Plastic, only the first coercion will be kept and the latter convertible ones will simply be ignored.

[19] This makes the dot-types different from the product types from another angle: if we took both projections π_1 and π_2 for product types $A \times B$ as coercions, the resulting system would be incoherent, as shown by Y. Luo in his PhD thesis [10].

different from ordinary inductive types. In Plastic, we use `A*B` for dot-type $A \bullet B$ and `dot<a,b>` for a dot term $\langle a, b\rangle$. Here are some simple examples of using dot-types in Plastic. Further details of the implementation could be found in [33].

Example 5.3. We can define a dot-type or a dot-term simply in the following way:

1. If we have two types A, B which do not share components, we could simply define a type M to be $A \bullet B$ like this:

```
> [M = A*B];
```

   The system will generate two coercions from $A \bullet B$ to A and B.
2. We can also define a dot term. If we have two terms $a{:}A$ and $b{:}B$, we can define a dot term $\langle a, b\rangle$ like this:

```
> [m = dot<a,b>];
```

   Now if A and B do not share components (otherwise, an error occurs), m is defined to be a dot term $\langle a, b\rangle$ which is of type $A \bullet B$. The system will also generate two coercions from $A \bullet B$ to A and B.

Example 5.4. In the following examples, the types share components in different ways and, therefore, none of them could be defined as a dot-type or dot term: they fail and warnings will be shown in all the following cases.

1. The two constituents are the same:

```
> [M = A*A];
```

2. $A \bullet C$ and $A \bullet B$ have the same component A:

```
> [M = (A*C)*(A*B)];
```

3. A is a subtype of B, by definition, they share component A:

```
> [c:A->B];
> Coercion = c;
> [M = A*B];
```

Appendix A. Judgement forms and inference rules of LF

Kinds in LF. The types in LF are called *kinds*, including

1. *Type* – the kind representing the universe of types;
2. $El(A)$ – the kind of objects of type A; and
3. $(x{:}K)K'$ (or simply $(K)K'$ when $x \notin FV(K')$) – the kind of dependent functional operations such as the abstraction $[x{:}K]k'$.

Judgement forms of LF-specified type theories. Any type theory specified in LF has the following five forms of judgements:

$$\Gamma \vdash valid, \qquad \Gamma \vdash K\ kind, \qquad \Gamma \vdash K = K', \qquad \Gamma \vdash k{:}K, \qquad \Gamma \vdash k = k'{:}K$$

Inference rules of LF. The following are the inference rules of LF. Any type theory specified in LF contains these rules.

Contexts, assumptions and weakening

$$\frac{}{\langle\rangle \vdash valid} \qquad \frac{\Gamma \vdash K\ \ kind\ \ x \notin FV(\Gamma)}{\Gamma, x{:}K \vdash valid} \qquad \frac{\Gamma, x{:}K, \Gamma' \vdash valid}{\Gamma, x{:}K, \Gamma' \vdash x : K}$$

$$\frac{\Gamma, \Gamma' \vdash J \quad \Gamma \vdash K\ kind \quad \Gamma, \Gamma'' \vdash valid}{\Gamma, \Gamma'', \Gamma' \vdash J}$$

where J is of the form $valid$, $K'\ kind$, $K_1 = K_2$, $k : K'$ or $k_1 = k_2 : K'$. *General equality rules*

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

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

Equality typing rules

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

where J is of the form *valid*, K' *kind*, $K_1 = K_2$, $k : K'$ or $k_1 = k_2 : K'$. *Substitution rules*

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

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

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

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

The kind Type

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

Dependent product kinds

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

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

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

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

Appendix B. LF-specifications of some inductive types

The specifications of the inductive types *Nat*, *List*(A), $\Pi(A, B)$ and $\Sigma(A, B)$ in the logical framework LF are given below.

Nat **– *the type of natural numbers.***

$$\begin{array}{rcl} Nat & : & Type \\ 0 & : & Nat \\ succ & : & (Nat)Nat \\ \mathbf{E}_{Nat} & : & \big(C{:}(Nat)Type\big) \\ & & \big(c{:}C(0)\big)\big(f{:}(x{:}Nat)\big(C(x)\big)C\big(succ(x)\big)\big) \\ & & (z{:}Nat)C(z) \end{array}$$

$$\mathbf{E}_{Nat}(C, c, f, 0) = c\ :\ C(0)$$
$$\mathbf{E}_{Nat}\big(C, c, f, succ(n)\big) = f\big(n, \mathbf{E}_{Nat}(C, c, f, n)\big)\ :\ C\big(succ(n)\big)$$

***List*(*A*) – the types of lists.**

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

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

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

***Π*-types of dependent functions.**

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

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

***Σ*-types of dependent pairs.**

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

$$\mathbf{E}_{\Sigma}\big(A,B,C,f,pair(A,B,a,b)\big)=f(a,b)\ :\ C\big(pair(A,B,a,b)\big)$$

References

[1] N. Asher, A type driven theory of predication with complex types, Fundamenta Informaticae 84 (2) (2008).
[2] D. Aspinall, A. Compagnoni, Subtyping dependent types, Theoretical Computer Science 266 (2001).
[3] A. Bailey, The machine-checked literate formalisation of algebra in type theory, PhD thesis, University of Manchester, 1999.
[4] P.C. Callaghan, Z. Luo, An implementation of typed LF with coercive subtyping and universes, Journal of Automated Reasoning 27 (1) (2001) 3–27.
[5] Coq, The Coq Proof Assistant Reference Manual (Version 8.3), INRIA. The Coq Development Team, 2010.
[6] H. Goguen, A typed operational semantics for type theory, PhD thesis, University of Edinburgh, 1994.
[7] S. Kleene, Introduction to Metamathematics, North-Holland, 1952.
[8] G. Longo, Subtyping parametric and dependent types, Invited lecture at school on type theory and term rewriting, September 1996, based on work by G. Chen and G. Longo.
[9] G. Longo, K. Milsted, S. Soloviev, Coherence and transitivity of subtyping as entailment, Journal of Logic and Computation 10 (4) (2000) 493–526.
[10] Y. Luo, Coherence and transitivity in coercive subtyping, PhD thesis, University of Durham, 2005.
[11] Z. Luo, Computation and Reasoning: A Type Theory for Computer Science, Oxford University Press, 1994.
[12] Z. Luo, Coercive subtyping in type theory, in: CSL'96, in: LNCS, vol. 1258, 1997.
[13] Z. Luo, Coercive subtyping, Journal of Logic and Computation 9 (1) (1999) 105–130.
[14] Z. Luo, Manifest fields and module mechanisms in intensional type theory, in: Types for Proofs and Programs, TYPES'08, in: LNCS, vol. 5497, 2009.
[15] Z. Luo, Type-theoretical semantics with coercive subtyping, Semantics and Linguistic Theory 20 (SALT20), Vancouver, 2010.
[16] Z. Luo, Contextual analysis of word meanings in type-theoretical semantics, in: Logical Aspects of Computational Linguistics (LACL'2011), in: LNAI, vol. 6736, 2011.
[17] Z. Luo, D-conservativity, Notes, January 2012.
[18] Z. Luo, Formal semantics in modern type theories with coercive subtyping, Linguistics & Philosophy, 2012, in press.
[19] Z. Luo, R. Adams, Structural subtyping for inductive types with functorial equality rules, Mathematical Structures in Computer Science 18 (5) (2008).
[20] Z. Luo, Y. Luo, Transitivity in coercive subtyping, Information and Computation 197 (2005) 122–144.
[21] Z. Luo, R. Pollack, LEGO Proof Development System: User's Manual, LFCS Report ECS-LFCS-92-211, Dept. of Comp. Sci., University of Edinburgh, 1992.
[22] Z. Luo, S. Soloviev, Dependent coercions, in: The 8th Inter. Conf. on Category Theory and Computer Science (CTCS'99), Edinburgh, Scotland, in: Electronic Notes in Theoretical Computer Science, vol. 29, 1999.

[23] L. Marie-Magdeleine, Sous-typage coercitif en présence de réductions non-standards dans un système aux types dépendants, PhD thesis, Université de Toulouse, 2009.
[24] P. Martin-Löf, Intuitionistic Type Theory, Bibliopolis, 1984.
[25] Matita, The Matita proof assistant, , http://matita.cs.unibo.it/, 2008.
[26] J.C. Mitchell, Coercion and type inference, in: Proc. of Tenth Annual Symposium on Principles of Programming Languages (POPL), 1983.
[27] J.C. Mitchell, Type inference with simple subtypes, Journal of Functional Programming 1 (2) (1991) 245–286.
[28] B. Nordström, K. Petersson, J. Smith, Programming in Martin–Löf's Type Theory: An Introduction, Oxford University Press, 1990.
[29] J. Pustejovsky, The Generative Lexicon, MIT, 1995.
[30] A. Saibi, Typing algorithm in type theory with inheritance, in: POPL'97, 1997.
[31] S. Soloviev, Z. Luo, Coercion completion and conservativity in coercive subtyping, Annals of Pure and Applied Logic 113 (1–3) (2002) 297–322.
[32] T. Xue, Conservativity of coercive subtyping, PhD thesis, Royal Holloway, University of London, 2012, forthcoming.
[33] T. Xue, Z. Luo, Dot-types and their implementation, in: Logical Aspects of Computational Linguistics (LACL 2012), in: LNCS, vol. 7351, 2012.