Dependent Coercions*

Zhaohui Luo
Computer Science Department
University of Durham
South Road, Durham DH1 3LE, U.K.
E-mail: Zhaohui.Luo@durham.ac.uk

Sergei Soloviev
IRIT, Université Paul Sabatier
118 route de Narbonne, 31062
Toulouse, France
E-mail: Sergei.Soloviev@irit.fr

July 30, 1999

Abstract

A notion of dependent coercion is introduced and studied in the context of dependent type theories. It extends our earlier work on coercive subtyping into a uniform framework which increases the expressive power with new applications.

A dependent coercion introduces a subtyping relation between a type and a family of types in that an object of the type is mapped into one of the types in the family. We present the formal framework, discuss its meta-theory, and consider applications such as its use in functional programming with dependent types.

1 Introduction

Coercive subtyping, as studied in [Luo97, Luo99, JLS98], represents a new general approach to subtyping and inheritance in type theory. In particular, it provides a framework in which subtyping, inheritance, and abbreviation can be understood in dependent type theories where types are understood as consisting of canonical objects.

In this paper, we extend the framework of coercive subtyping to introduce a notion of *dependent coercion*. A dependent coercion introduces a subtyping relation between a type A and a family of types $B(x)$ that are indexed by objects x of type A. For example, the type of lists may be regarded as a subtype of the family of vector types via a coercion that maps a list into its 'corresponding' vector. This extends our earlier work on coercive subtyping and provides a uniform framework in which simple coercions (between two types), parameterised coercions (between two families of types), and dependent coercions (between a type and a family of types) can all be studied. Applications of dependent coercions include its use in functional programming with dependent types, large proof development, and formalisation of certain mathematical concepts.

In the following section, we first give an overview of coercive subtyping and a summary of some of our earlier work on this. Then, in Section 3, we introduce the framework of

*In Proc. of the 8th Inter. Conf. on Category Theory in Computer Science (CTCS'99), Edinburgh, 1999. This work is partly supported by the UK EPSRC grant on 'Subtyping, Inheritance and Reuse' (GR/K79130).

dependent coercion. In Section 4, we show that every dependent coercion can be represented in a 'canonical' form. Section 5 discusses the potential applications, its implementation in Callaghan's system Plastic, and the related issues such as coherence checking.

2 Coercive subtyping: an overview of work so far

Motivation and basic ideas

Data types in dependent type theories such as Martin-Löf's type theory [NPS90] and the type theory UTT [Luo94], can in general be considered as inductive in the sense that they consist of their canonical objects. This is rather different from the traditional views when one studies type systems of programming languages and most of the work about subtyping, where objects constitute a pre-given universe, while types are assigned to the objects and a subtyping relation is obtained by overloading object terms (eg. λ-terms). It is not clear (if possible) how the traditional approach to subtyping can be applied to type theory with inductive types in accordance with the view that types consist of canonical objects.

Coercive subtyping represents a new approach to subtyping and inheritance in type theory. The basic idea is that A is a subtype of B if there is a (unique) coercion c from A to B, and therefore, any object of type A may be regarded as an object of type B via c, where c is a functional operation from A to B in the type theory. In the theoretical framework of coercive subtyping, this is represented by the coercive definition rule (see Figure 2), which says that, if f is a functional operation with domain K, k_0 is an object of K_0, and c is a coercion from K_0 to K, then $f(k_0)$ is definitionally equal to $f(c(k_0))$. Intuitively, we can view f as a context which requires an object of K; then the argument k_0 in the 'context f stands for its image of the coercion, $c(k_0)$. Therefore, one can use $f(k_0)$ as an abbreviation of $f(c(k_0))$.

Power of the framework

The above simple idea, when formulated in a typed logical framework [Luo94], becomes very powerful. In our early work [Luo97, Luo99], we have developed the framework that covers subtyping relations represented by the following kinds of coercions:

- *Simple coercions*: representing subtyping between two types. For example, coercions between basic inductive types: $Even$ is a subtype of Nat.

- *Parameterised coercions*: representing (point-wise) subtyping (or subfamily relation) between two families of types indexed by objects of the same type. A coercion can be parameterised over free variables occurring in it and (possibly) its domain or range types. As a special case, for example, each vector type $Vec(A, n)$ can be taken as a subtype of that of lists $List(A)$, parameterised by the index n, where the coercion would map the vector $< a_1, ..., a_n >$ to the list $[a_1, ..., a_n]$.

- *Coercions between parameterised inductive types*: we have general schematic rules that represent natural propagation of the basic coercions to other structured (or parameterised) inductive types. For example, $\Sigma(A, B)$ is a subtype of $\Sigma(A', B')$ if A is a subtype of A and B is a subfamily of B'.

Coercive subtyping has applications in many areas such as large proof development, inductive reasoning, representing implicit syntax, etc.

Conservativity and meta-theoretic results

We have studied some important meta-theoretic aspects of coercive subtyping (for non-dependent coercions) [JLS98, SL98]. In particular, we have proved results on transitivity elimination for kinds and on conservativity.

The conservativity result says, intuitively, that every judgement that is derivable in the theory with coercive subtyping and that does not contain coercive applications is derivable in the original type theory. Furthermore, for every derivation in the theory with coercive subtyping, one can always insert coercions correctly to obtain a derivation in the original type theory.

The main result of [SL98] was that coherence of basic subtyping rules does imply conservativity, under certain conditions (these conditions are satisfied, for example, for the type theory UTT or Martin-Löf's type theory.) The proof of the conservativity theorem consists of the following three major parts:

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

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

Implementations

Coercion mechanisms of non-dependent coercions with certain restrictions have been implemented both in the proof development systems Lego [LP92] and Coq [Coq96], by Bailey [Bai96, Bai98] and Saibi [Sai97], respectively. Callaghan of the Computer Assisted Reasoning Group at Durham has recently implemented Plastic, a proof assistant that supports logical framework and coercive subtyping (see Section 5 for more information.)

Related work

Subtyping in various type systems is actively studied since mid-eighties (cf, [CW85]). The more traditional approach to subtyping considers usually a subtyping relation over lambda-terms and its properties (eg, the existence of principal or minimal typing). The notion of coercion was introduced later as an explicit representation of the transformation of (the elements of) the subtype into (the elements of) the supertype. The subtyping relation was interpreted by the existence of a certain definable term $c{:}A \to B$ when $A < B$, with motivation of giving semantics to calculi with subtyping and inheritance (see, e.g., [BCGS91], where no equational theory was studied for the calculus with coercions). Others have also considered coercions in different frameworks of subtyping. See, for example, [LMS95, Che98].

The framework on coercive subtyping takes a different approach – taking coercions seriously and directly at the proof-theoretic level (they extend type theories directly with coercive definition rules) and providing a coherent view on how subtyping and inheritance can be studied in a type theory with inductive data types [CPM90, Dyb91, Luo94]. The work has been influenced by Peter Aczel and Anthony Bailey via their project on classes and coercions [Bai98], and by Randy Pollack via his idea of type-checking terms with implicit coercions (private communication). The current work extends this framework to dependent coercions.

3 Dependent coercions

We first give an informal explanation of what a dependent coercion is. Then, the formal framework of coercive subtyping (with dependent coercions) is presented.

3.1 An informal introduction

With dependent types, it is natural to consider when a type is a subtype of a family of types. For instance, we can consider the type of lists $List(A)$ be a subtype of the family of types of vectors, $Vec(A,n)$.[1] A natural coercion between them is the functional operation c that maps list $[a_1,...,a_n]$ to the vector $< a_1,...,a_n >$. More precisely, we can define

$$\begin{array}{rcl} c(nil(A)) & = & nil_V(A) : Vec(A,0) \\ c(cons(A,a,l)) & = & cons_V(A,|l|,a,c(l)) : Vec(A,|l|+1) \end{array}$$

This coercion c is of kind dependent product $(l{:}List(A))Vec(A,|l|)$, where $|l|$ is the length of l.

The framework of coercive subtyping studied before, eg, in [Luo99], does not allow such coercions with dependent types. This is an example of dependent coercion; we can declare that c is a dependent coercion from the type of lists to the family of types of vectors. In notation, we write this as:

$$l{:}List(A) \xrightarrow{c} Vec(A,|l|),$$

where $Vec(A,|l|)$ depends on the bound variable l. More generally and more formally, we may declare the following basic subtyping rule to introduce such a coercion:

$$\frac{A \xrightarrow{c_{AB}} B : \mathbf{Type}}{l{:}List(A) \xrightarrow{c} Vec(B,|l|) : \mathbf{Type}}$$

where c is defined as above except that in the second clause of the above definition, we have $c(a{::}l) = cons_V(B,|l|,c_{AB}(a),c(l))$.

Note that a dependent coercion is *different* from a parameterised coercion of the form $A \xrightarrow{c'} B(x)\ [x{:}P]$. The parameterised coercion c' says that the type A is regarded as a subtype of each type in family B, while a dependent coercion of the form $x{:}A \xrightarrow{c} B(x)$ is in fact saying that, informally, A is a subtype of the 'union' of the types in the family B.

[1] $List(A)$, parameterised over type A, is introduced as the inductive type with constructors $nil(A) : List(A)$ and $cons(A) : (a{:}A)(l{:}List(A))List(A)$. The inductive family of types $Vec(A) : (n{:}Nat)\mathbf{Type}$, indexed over natural numbers and parameterised over type A, is introduced with constructors $nil_V(A) : Vec(A,0)$ and $cons_V(A) : (n{:}Nat)(a{:}A)(v{:}Vec(A,n))Vec(A,n+1)$.

3.2 Dependent coercions: a formal presentation

We consider how to extend any type theory specified in the logical framework LF with dependent coercions as well as other coercions.

3.2.1 Logical framework and notations

The logical framework LF [Luo92, Luo94] is a *typed* version of Martin-Löf's logical framework (see Chapter 19 of [NPS90] for a presentation of the latter). The rules of LF are given in Appendix A and, for how to specify type theories in LF, we refer to Chapter 9 of [Luo94] or [Luo99] for more detailed discussions. Examples of type theories that can be specified with LF include Martin-Löf's intensional type theory [NPS90], UTT [Luo94], and many others.

Paul Callaghan of the Computer Assisted Reasoning Group at Durham has implemented LF in the form of a proof assistant for the language, called Plastic. In Plastic one can specify type theories such as UTT; it provides mechanisms for inductive types and universes, and has a library providing logical reasoning and many standard data types. Plastic also implements coercive subtyping. See Section 5.2 for more information.

Notations The following basic notational conventions will be used in this paper.

- Substitution: as usual, $[N/x]M$ stands for the expression obtained from M by substituting N for the free occurrences of variable x in M, defined as usual with possible changes of bound variables; informally, we sometimes use $M[x]$ to indicate that variable x may occur free in M and subsequently write $M[N]$ for $[N/x]M$, when no confusion may occur.

- We shall often omit El-operator in LF to write A for $El(A)$ when no confusion may occur and may write $(K)K'$ for $(x{:}K)K'$ when x does not occur free in K'.

- Identity function: $id_M \equiv [x{:}M]x$.

- Functional composition: for $f : (K_1)K_2$ and $g : (y{:}K_2)K_3[y]$, define $g \circ f =_{\mathrm{df}} [x{:}K_1]g(f(x)) : (x{:}K_1)K_3[f(x)]$, where x does not occur free in f or g.

3.2.2 Judgement forms

Besides the judgement forms in LF, we consider two new forms of judgement, which assert that c is a coercion (possibly dependent coercion) from kind K to kind K' and from type A to type B, respectively:

$$\Gamma \vdash x{:}K \xrightarrow{c} K' \quad and \quad \Gamma \vdash x{:}A \xrightarrow{c} B : \mathbf{Type},$$

where $x{:}K$ and $x{:}A$ bind variable x in K' and B, respectively, but they do not bind x in c. We also say that K is a subkind of K' (and A is a subtype of B) via coercion c.

Notation When x does not occur free in K' (B), we write

$$\Gamma \vdash K \xrightarrow{c} K' \quad and \quad \Gamma \vdash A \xrightarrow{c} B : \mathbf{Type}$$

for the above two judgements, respectively. Note that, when K and K' (or B and B') are not computationally equal, $\Gamma \vdash K \xrightarrow{c} K'$ and $\Gamma \vdash A \xrightarrow{c} B$: **Type** correspond to the judgement forms $\Gamma \vdash K <_c K'$ and $\Gamma \vdash A <_c B$: **Type** we have used in, eg, [Luo99].

Let T be any type theory specified in **LF**. We shall present the system T[$\mathcal{R}$], the extension of T with coercive subtyping (with dependent coercions), whose subtyping relation is given by the basic subtyping rules $\mathcal{R}$, which satisfy certain coherence conditions. In order to state the coherence conditions for the basic subtyping rules, we first consider an intermediate system T[$\mathcal{R}$]$_0$.

3.2.3 T[$\mathcal{R}$]$_0$ and coherence conditions

T[$\mathcal{R}$]$_0$ extends T (only) with the new judgement form of subtyping, $\Gamma \vdash x{:}A \xrightarrow{c} B$: **Type**, and the following new rules:

- A set $\mathcal{R}$ of basic subtyping rules whose conclusions are subtyping judgements of the form $\Gamma \vdash x{:}A \xrightarrow{c} B$: **Type**.
- The general subtyping rules in Figure 1. .

Identity coercion

$$\frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash A \xrightarrow{id_A} A : \mathbf{Type}}$$

Congruence

$$\frac{\Gamma \vdash x{:}A \xrightarrow{c} B : \mathbf{Type} \quad \Gamma \vdash A = A' : \mathbf{Type} \quad \Gamma, x{:}A \vdash B = B' : \mathbf{Type} \quad \Gamma \vdash c = c' : (x{:}A)B}{\Gamma \vdash x{:}A' \xrightarrow{c'} B' : \mathbf{Type}}$$

Transitivity

$$\frac{\Gamma \vdash x{:}A \xrightarrow{c} B : \mathbf{Type} \quad \Gamma, x{:}A \vdash y{:}B \xrightarrow{c'} C[x,y] : \mathbf{Type}}{\Gamma \vdash x{:}A \xrightarrow{[x:A]c'(c(x))} C[x, c(x)] : \mathbf{Type}}$$

Substitution

$$\frac{\Gamma, x{:}K, \Gamma' \vdash y{:}A \xrightarrow{c} B : \mathbf{Type} \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x](y{:}A \xrightarrow{c} B) : \mathbf{Type}}$$

Figure 1: General type coercion rules in T[$\mathcal{R}$]$_0$ (and T[$\mathcal{R}$]).

Note that in T[$\mathcal{R}$]$_0$, the subtyping judgements do not contribute to any derivation of a judgement of any other form. Therefore, T[$\mathcal{R}$]$_0$ is obviously a conservative extension of T.

Note that we have included the identity function as a coercion. Subtyping relations for the object type theories specified in LF are introduced as (default) basic subtyping rules, which may include subtyping rules for parameterised data types such as Π-types and Σ-types. For most of the applications, these coercions are introduced between *data types* in the type theory. (See [Luo99] for examples.)

The set of basic coercion rules are required to be coherent in the following sense.

Definition 3.1 (coherence condition) *A set R of basic coercion rules is coherent if the following is true in $T[\mathcal{R}]_0$:*

- *If* $\Gamma \vdash x{:}A \xrightarrow{c} B[x] : \mathbf{Type}$*,* $\Gamma \vdash x{:}A \xrightarrow{c'} B'[x] : \mathbf{Type}$*,* $\Gamma \vdash a : A$*, and* $\Gamma \vdash B[a] = B'[a] : \mathbf{Type}$*, then* $\Gamma \vdash c(a) = c'(a) : B[a]$*.*

Remark From the above, we have the following as consequences:

- If $\Gamma \vdash A \xrightarrow{c} A : \mathbf{Type}$, then $\Gamma \vdash c = id_A : (A)A$.
- If x does not occur free in B or B', we have (by $\eta\xi$-equality rules in LF) that $\Gamma \vdash c = c' : (A)B$, if $\Gamma \vdash A \xrightarrow{c} B : \mathbf{Type}$, $\Gamma \vdash A \xrightarrow{c'} B : \mathbf{Type}$, and A is not empty in Γ. This is the coherence condition for non-dependent coercions in, eg, [Luo99], except the requirement of non-emptyness of A.

3.2.4 T[$\mathcal{R}$] and inference rules

Let $\mathcal{R}$ be a set of coherent basic subtyping rules. The system T[$\mathcal{R}$], the extension of T with coercive subtyping (with dependent coercions) with respect to $\mathcal{R}$, is the system obtained from T$[\mathcal{R}]_0$ by adding the new subkinding judgement form $\Gamma \vdash x{:}K \xrightarrow{c} K'$, the coercive application and coercive definition rules in Figure 2, and the general kind coercion rules in Figure 3.

Note that the judgement $\Gamma \vdash k \ : \ K$ means that k is an object with *principal* kind K, while the definable judgement $\Gamma \vdash k \ :: \ K$, which can be introduced by means of the following rules (cf, [Luo99]):

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

would represent typing in general.

The basic subtyping rules $\mathcal{R}$ represent the intended (and possibly user-defined) subtyping relations between data types. Note that the basic relations are between *types*, not between arbitrary kinds. It is not restricted to constant types (such as $Even$ and Nat) but can be between structured types such as Σ-types representing abstract mathematical theories (such as those of rings and groups) possibly with the intended coercions specified by the user of a proof system.

The coherence conditions are the most basic and necessary requirements for the basic subtyping rules. Note that in the paradigm of coercive subtyping, coercions between any two kinds are required to be unique up to computational equality: it is easy to show that, by the coercive definition rule and $\beta\eta\xi$-equality rules, if $x{:}K \xrightarrow{c} K'$ and $x{:}K \xrightarrow{c'} K'$, then we have $c = c' : (x{:}K)K'$. Coherence checking and proofs are not easy when parameterised coercions or dependent coercions are present. (See Section 5.2 for a further discussion.)

Coercive application rules

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

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

Coercive definition rule

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

Figure 2: Coercive application/definition rules in T[$\mathcal{R}$].

3.3 Meta-theoretic results

The meta-theoretic results for non-dependent coercions, as sketched in Section 2, can be lifted for dependent coercions. In particular, the conservativity theorem holds for the framework with dependent coercions as well: every judgement that is derivable in T[$\mathcal{R}$] and that does not contain coercive applications (cf, the coercive application rule in Figure 2) is derivable in the original type theory T. Furthermore, coercion completion is justified: for every derivation/judgement/object in T[$\mathcal{R}$], one can insert coercions correctly to obtain a computationally equal counterpart of the derivation/judgement/term in the original type theory T. We omit the details here and refer the reader to the similar results for non-dependent coercions presented in [SL98].

3.4 Coercion rules for dependent products: a discussion

The coercion rule for dependent product kinds in Figure 3 is worth further discussion. In our rule, the coercions in the premises are restricted to be non-dependent; in other words, dependent coercions are not allowed to be lifted to dependent product kinds in the usual contravariant way.

One may consider more general rules. For example, the following rule allows the second coercion in the premises to be dependent, while restricting the first to be non-dependent:

$$(*) \quad \frac{\Gamma, x{:}K_1 \vdash K_2[x] \ \mathbf{kind} \quad \Gamma \vdash K_1' \xrightarrow{c_1} K_1 \quad \Gamma, x'{:}K_1' \vdash y{:}K_2[c_1(x')] \xrightarrow{c_2} K_2'[x', y]}{\Gamma \vdash f{:}(x{:}K_1)K_2[x] \xrightarrow{[f:(x:K_1)K_2[x]][x':K_1']c_2(f(c_1(x')))} (x'{:}K_1')K_2'[x', f(c_1(x'))]}$$

Or, one could take a even more liberal view to allow both coercions in the premises to be dependent:

$$\frac{\begin{array}{l}\Gamma, x'{:}K_1', x{:}K_1[x'] \vdash K_2[x', x] \ \mathbf{kind} \\ \Gamma \vdash x'{:}K_1' \xrightarrow{c_1} K_1[x'] \quad \Gamma, x'{:}K_1' \vdash y{:}K_2[x', c_1(x')] \xrightarrow{c_2} K_2'[x', y]\end{array}}{\Gamma \vdash f \ : \ (x'{:}K_1')(x{:}K_1[x'])K_2[x', x] \xrightarrow{c} (x'{:}K_1')K_2'[x', f(x', c_1(x'))]}$$

Basic kind coercion rules

$$\frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash \mathbf{Type} \xrightarrow{id_{Type}} \mathbf{Type}} \qquad \frac{\Gamma \vdash x{:}A \xrightarrow{c} B : \mathbf{Type}}{\Gamma \vdash x{:}El(A) \xrightarrow{c} El(B)}$$

Kind coercion for dependent product kinds

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

Congruence rule

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

Transitivity rule

$$\frac{\Gamma \vdash x{:}K \xrightarrow{c} K' \quad \Gamma, x{:}K \vdash y{:}K' \xrightarrow{c'} K''[x,y]}{\Gamma \vdash x{:}K \xrightarrow{[x:K]c'(c(x))} K''[x, c(x)]}$$

Substitution rule

$$\frac{\Gamma, x{:}K, \Gamma' \vdash y{:}K_1 \xrightarrow{c} K_2 \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x](y{:}K_1 \xrightarrow{c} K_2)}$$

Figure 3: Kind coercion rules in T[$\mathcal{R}$].

where $c(f, x') = c_2(f(x', c_1(x')))$.

It is obvious that having more general rules would lift more dependent coercions from the type level to dependent product kinds. For instance, the $(*)$ rule above would have the effect that, for example, there is a coercion

$$f{:}(A{:}\mathbf{Type})List(A) \xrightarrow{d} (A{:}\mathbf{Type})Vec(A, |f(A)|),$$

where $d(f, A) = c(f(A))$, if we assume that we have the dependent coercion from lists to vectors as discussed before. This coercion would not be derivable using our simple rule.

It requires further investigation to understand how these dependent coercions lifted to the dependent product kinds can be used in practice and what the implications are for the theory. There is one difficulty in the meta-theoretic study: with the more general rules considered here, the transitivity elimination result at the kind level fails to hold. Note that transitivity elimination was used to prove the conservativity theorem as sketched above, we have not succeeded in proving the conservativity result for these more general rules. We leave these to future research.

4 The π_2-coercions

Dependent coercions can either be introduced by the user (eg, the dependent coercion between lists and vectors), or formed by composition with other coercions, which can be simple, parameterised, or dependent. Although dependent coercions can be rather complicated, it is interesting to note that they can all be represented in some canonical form.

In fact, all dependent coercions can be represented as compositions of non-dependent coercions with a special dependent coercion – the second projection for Σ-types.

Consider dependent sum types (or strong sums) $\Sigma(A, B)$ for type $A : \mathbf{Type}$ and family $B : (A)\mathbf{Type}$ indexed by objects in A. Let π_1 and π_2 be the first projection and second projection, respectively. Then, we can declare the second projection to be a dependent coercion:

$$p{:}\Sigma(A, B) \xrightarrow{\pi_2} B(\pi_1(p)) : \mathbf{Type}.$$

Then, for any dependent coercion from A to B,

$$x{:}A \xrightarrow{c} B(x) : \mathbf{Type},$$

we can define the following non-dependent coercion

$$A \xrightarrow{d_c} \Sigma(A, B) : \mathbf{Type},$$

where $d_c(x) = pair(A, B, x, c(x)) : \Sigma(A, B)$. Then, the composition of d_c and π_2 is a dependent coercion from A to B and $\pi_2 \circ d_c = c : (x{:}A)B(x)$.

So, any dependent coercion can be represented as the composition of a π_2-coercion with a non-dependent coercion. Furthermore, this representation preserves coherence, as the following theorem shows.

Theorem 4.1 *Let T' be the type theory obtained from T by adding a new Σ-type constructor. Then $T[\mathcal{R}]_0$ is coherent with*

$$x{:}A \xrightarrow{c} B(x) : \mathbf{Type}$$

if and only if $T'[\mathcal{R}]_0$ *is coherent with the following coercions*

$$A \xrightarrow{d_c} \Sigma(A, B) : \mathbf{Type},$$

$$p{:}\Sigma(A, B) \xrightarrow{\pi_2} B(\pi_1(p)) : \mathbf{Type},$$

where $d_c(x) =_{\rm df} pair(A, B, x, c(x))$.

Proof sketch The if part is trivial, since by the transitivity and congruence rules, we have $x{:}A \xrightarrow{c} B(x)$: **Type**. For the only-if part, we only have to show that T'$[\mathcal{R}]_0$ is a conservative extension of T$[\mathcal{R}]_0$. This is the case because the Σ-types involved are new. □

Remark The condition in the above theorem that the Σ-type constructor is *new* is important. The type theory T may have other strong sum types over which there may be other coercions defined, but the added Σ-type constructor is a different copy, distinct from the other strong sum types.

Considering the intuitive meaning of a dependent coercion, the above result is not surprising, if one notes that the Σ-type $\Sigma(A, B)$ intuitively represents the 'union' of the family B.

5 Applications and implementation

In this section, we briefly discuss applications of dependent coercions, and its implementation and related issues.

5.1 Applications

Exisiting applications. Coercive subtyping has applications in large proof development [Bai98] and provides useful mechanisms for inductive reasoning, overloading, and representation of some implicit syntax, etc (see [Luo99]). Dependent coercions extend the power of the framework in these areas. For example, the π_2-coercions allow more flexible structuring and reuse of proofs in formalisation of mathematical theories.

Application to functional programming with dependent types. When we consider functional programming with dependent types as well as non-dependent types, it is often crucial and very useful if one can reuse programs with dependent types (eg, functions concerning vectors) in the world of non-dependent types (eg, that of lists). Dependent coercions (eg, the coercion from lists to vectors), together with other coercions, are useful in such transformations. For example, one can define a function from lists to lists by means of a similar function from vectors to vectors, rather than defining the former directly. This provides a basis for reusing functional programs and makes the use of dependent types easier in programming.

Formalisation of mathematical concepts. Some mathematical concepts involve a set being a subset of the union of a family of sets, and with dependent coercions, it is possible to model such concepts at the level of types. The notion of covering is such an example: we can consider a type A and a family of types A_s such that every element of A can be regarded as an element of some A_s, while each A_s is a subtype of A.

For example, let Nat be the type of natural numbers with constructors $zero$ and $succ$. As in [Luo99], we can consider the subtypes of even and odd numbers as the copies of Nat, $Even =_{\mathrm{df}} Nat_0$ (with constructors $zero_0$ and $succ_0$) and $Odd =_{\mathrm{df}} Nat_1$ (with constructors $zero_1$ and $succ_1$) with the following coercions:

$$Nat_0 \xrightarrow{c_0'} Nat : \mathbf{Type} \quad and \quad Nat_1 \xrightarrow{c_1'} Nat : \mathbf{Type},$$

where for $i = 1, 2$,

$$c_0'(zero_0) = zero, \;\; c_1'(zero_1) = succ(zero), \;\; and \;\; c_i'(succ_i(x)) = succ(succ(c_i'(x))).$$

One may define also a dependent coercion

$$x{:}Nat \xrightarrow{c} Nat_{i(x)} : \mathbf{Type},$$

where $i(x) = \begin{cases} 0, & if\ x\ is\ even \\ 1, & if\ x\ is\ odd \end{cases}$. The coercion c of kind $(x{:}Nat)Nat_{i(x)}$ maps the even natural numbers of type Nat $(0, 2, ...)$ to the even numbers of type Nat_0 $(zero_0, succ_0(zero_0), ...)$, and the odd natural numbers $(1, 3, ...)$ to the odd numbers of type Nat_1 $(zero_1, succ_1(zero_1), ...)$.

The above coercions form a covering in the intuitive sense. Furthermore, they themselves constitute a coherent set of basic coercions (note that there is no composition of the coercion c_i' with c to form a coercion from Nat to Nat, since Nat_0 (or Nat_1) is not computationally equal to $Nat_{i(x)}$.)

Some remarks on extensionality. Note that in the above example, the coercions c_0' and c_1' may be written in the parametric form as $x{:}Nat \vdash Nat_{i(x)} \xrightarrow{c_{i(x)}'} Nat : \mathbf{Type}$. If we took this as a coercion as well, together with the coercion c above, the whole system of coercions would not be coherent, since in this case, we could compose c with $c_{i(x)}'$ to obtain a coercion from Nat to Nat that is not computationally equal to id_{Nat}. This is an example where coercions only satisfy what we may call 'extensional coherence', ie, two coercions with the same domain and range types are only extensionally equal, but are not intensionally (or computationally) equal.

In this paper and in the study of coercive subtyping in general, we have assumed that our underlying type theories are intensional. However, if we consider extensional type theories (cf, [ML84]), which sometimes are good in direct formalisation of mathematical concepts, then our notion of coherence becomes extensional and the above system of coercions would be (extensionally) coherent. Extensional coherence is sometimes a very useful notion and needs further study.

5.2 Implementation

The proof system Plastic [Cal99], implemented by Callaghan at Durham, supports coercive subtyping, including the use of dependent coercions. Several coercion mechanisms have been implemented in Plastic, allowing a mixture of simple coercions, parametrised coercions, coercion rules, and dependent coercions. The mechanism makes use of meta-variable facilities (including unification) in the system to calculate the coercion terms. Plastic is being used for experiments which investigate use of coercive subtyping, especially dependent coercions, in functional programming.

As mentioned before, Plastic implements the typed LF with several extensions, such as for inductive types and universes. There are several motivations for Plastic: to support research on coercive subtyping, mathematical vernacular [LC98], functional programming with dependent types [KLM99], and interfaces to type theory based proof assistants [CL98]. The system is described in more detail on the WWW page `http://www.dur.ac.uk/CARG/plastic.html`.

Coherence checking

Parameterised coercions and dependent coercions introduce infinitely many coercions; therefore, coherence checking is in general undecidable. In practice, checking coherence of user-defined coercions is also a very difficult task. However, there are at least two possible approaches to this problem.

Firstly, it is possible to consider different classes of coercions useful for certain applications and prove (by hand, and at the meta-level) that each constitutes a coherent set of basic subtyping rules. As to dependent coercions, for example, we can easily show that the coercion from lists to vectors in our example above is coherent (without considering other coercions.) We call this approach of using external proofs to guarantee coherence of coercion sets as an approach of 'meta-arguments'.

However, in practice, it is often the case that we cannot predict what coercions a user might use. Therefore, in implementing coercions, some form of coherence checking is necessary. When we have dependent coercions or parameterised coercions, one possibility is to consider *dynamic checking*. In this method, the system keeps a set of coercion instances *used so far*, and guarantees that any use of coercions does not introduce conflicting instances. Note that, dynamic checking is completely a practical approach: it only makes sure that the coercion instances used are not in conflict, but it does not guarantee that the declared coercions are coherent.

The approach of using meta-arguments to ensure coherence can be combined with dynamic checking to make coherence checking more efficient: one does not need to check whether two coercion instances are in conflict if the coercions concerned have been proved to be coherent with each other. We are exploring this idea of dynamic checking using the implementation of Plastic.

Acknowledgement Thanks to Paul Callaghan and James McKinna who have read drafts of this paper and given their comments. It is also a pleasure to work with Paul Callaghan who implemented coercions in Plastic. A discussion with Alex Jones on dynamic checking has been very useful. Thanks also go to the CTCS referees who have made very useful comments that have helped improve the paper.

A The inference rules for LF

The inference rules for the typed logical framework LF are given in Figure 4.

Contexts and assumptions

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

Equality rules

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

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

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

Substitution rules

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

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

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

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

The kind Type

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

Dependent product kinds

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

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

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

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

Figure 4: The inference rules of LF.

References

[Bai96] A. Bailey. Lego with implicit coercions. 1996. Draft.

[Bai98] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory*. PhD thesis, University of Manchester, 1998.

[BCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. *Information and Computation*, 93, 1991.

[Cal99] P.C. Callaghan. Plastic: an implementation of typed LF with coercions. Talk given in the Annual Conf of TYPES'99, June 1999.

[Che98] G. Chen. *Subtyping, Type Conversion and Transitivity Elimination*. PhD thesis, University of Paris VII, 1998.

[CL98] P. Callaghan and Z. Luo. Mathematical vernacular in type theory based proof assistants. *User Interfaces for Theorem Provers (UITP'98), Eindhoven*, 1998.

[Coq96] Coq. *The Coq Proof Assistant Reference Manual (version 6.1)*. INRIA-Rocquencourt and CNRS-ENS Lyon, 1996.

[CPM90] Th. Coquand and Ch. Paulin-Mohring. Inductively defined types. *Lecture Notes in Computer Science*, 417, 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17, 1985.

[Dyb91] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[JLS98] A. Jones, Z. Luo, and S. Soloviev. Some proof-theoretic and algorithmic aspects of coercive subtyping. *Types for proofs and programs (eds, E. Gimenez and C. Paulin-Mohring), Proc. of the Inter. Conf. TYPES'96, LNCS 1512*, 1998.

[KLM99] R. Kiessling, Z. Luo, and J. M. McKinna. Some issues in functional programmiung with dependent types. Talk in the Annual Conference of TYPES'99, June 1999.

[LC98] Z. Luo and P. Callaghan. Mathematical vernacular and conceptual well-formedness in mathematical language. *Proceedings of the 2nd Inter. Conf. on Logical Aspects of Computational Linguistics, LNCS/LNAI 1582*, 1998.

[LMS95] G. Longo, K. Milsted, and S. Soloviev. A logic of subtyping. In *Proc. of LICS'95*, 1995.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[Luo92] Z. Luo. A unifying theory of dependent types: the schematic approach. *Proc. of Symp. on Logical Foundations of Computer Science (Logic at Tver'92), LNCS 620*, 1992. Also as LFCS Report ECS-LFCS-92-202, Dept. of Computer Science, University of Edinburgh.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[Luo97] Z. Luo. Coercive subtyping in type theory. *Proc. of CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic, Utrecht. LNCS 1258*, 1997.

[Luo99] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[Sai97] A. Saibi. Typing algorithm in type theory with inheritance. *Proc of POPL'97*, 1997.

[SL98] S. Soloviev and Z. Luo. Coercion completion and conservertivity in coercive subtyping. Draft submitted, 1998.