Coercive subtyping*

Zhaohui Luo
Department of Computer Science
University of Durham
Email: Zhaohui.Luo@durham.ac.uk

Abstract

We propose and study *coercive subtyping*, a formal extension with subtyping of dependent type theories such as Martin-Löf's type theory [38] and the type theory UTT [30]. In this approach, subtyping with specified implicit coercions is treated as a feature at the level of the logical framework; in particular, the meaning of an object being in a supertype is given by *coercive definition rules* for the definitional equality. This provides a conceptually simple and uniform framework to understand subtyping and inheritance relations in type theories with sophisticated type structures such as inductive types and universes. The use of coercive subtyping in formal development and in reasoning about subsets of objects is discussed in the context of computer-assisted formal reasoning.

Key words: type theory, subtyping, coercion, formal reasoning, logical framework.

1 Introduction

A type in type theory is often intuitively thought of as a set. For example, types in Martin-Löf's type theory [36, 38] can be considered as inductively defined sets. A fundamental difference between type theory and set theory is that in the former we do not have a notion of subtype that corresponds to the notion of subset in the latter. The lack of useful subtyping mechanisms in dependent type theories with inductive types [17, 20, 30] and the associated proof development systems [35, 14, 19, 34] is one of the obstacles in their applications to large-scale formal development.

Although subtyping is conceptually natural and pragmatically important, it has not been clear how useful and suitable subtyping mechanisms can be introduced into dependent type theories. Particularly, in the presence of inductive types which include types of natural numbers, lists, and trees, and types of mathematical structures such as Σ-types, it is not clear how subtyping should be introduced to reason about subsets and represent inheritance, without compromising with good proof-theoretic properties. More recently, in Aczel's project on formalising abstract algebraic theories (Galois theory), Bailey has implemented various forms of coercions in the Lego system [34], which are very useful in practical large-scale development of mathematical

*Journal of Logic and Computation, Vol. 9, No. 1, pp. 105-130. 1999.

theories [3].

In this paper, we present an equational formulation of *coercive subtyping*, a novel approach to subtyping and inheritance for dependent type theories that can be formulated in a logical framework. Examples of such type theories include Martin-Löf's intensional type theory [38] and the type theory UTT [30], which are the underlying type theories of the proof systems ALF and Lego, respectively. Our approach has the following features:

- Conceptually, in coercive subtyping, subtyping is considered as an abbreviational mechanism. It generalises both notions of subtyping — that based on the notion of subset (eg, between the types of natural numbers and integers) and that based on inheritance (eg, between record types). Therefore, it provides a general theory to study subtyping and inheritance.

- With (possibly user defined) implicit coercions, the meaning of an object being in a supertype is given by *coercive definition rules* for the definitional equality. This gives a proof-theoretic (and direct meaning-theoretic) treatment of subtyping as implicit coercions, as compared with its possible model-theoretic semantic counterpart (cf., [8]).

- Subtyping is treated as an extension of the underlying logical framework—the meta-language in which type theories are formulated. Making essential use of a *typed* logical framework LF [30], it gives a general extension of various intensional type theories with subtyping. As a formal system, the extended framework is just a simple extension of LF.

- The extended framework provides a generic and uniform setting to understand various forms of coercions in type systems (e.g., those for structured types and universes and those found in Bailey's implementation in Lego) and some other useful mechanisms such as overloading and type-casting.

Coercive subtyping can be seen to represent a conceptually simple but powerful approach to introducing subtyping into type theory. In the practice of computer-assisted formal reasoning, we believe that coercive subtyping provides easier and more powerful reasoning mechanisms for reasoning about subsets of objects as well as for reusing proven results in developed formal theories (cf., [1]).

In the following section, we briefly introduce the logical framework and explain how to use it as a meta-language to specify type theories. In Section 3, the basic ideas of coercive subtyping are explained and the extended logical framework is formally presented. The use of coercive subtyping is considered in Section 4. Related work and further research topics are discussed in the Conclusion.

2 The logical framework LF and formulation of type theories

The logical framework LF [28, 30] is a *typed* version of Martin-Löf's logical framework (see Chapter 19 of [38] for a presentation of the latter). We should also point out that LF is different from the Edinburgh Logical Framework (ELF) [22].

The presentation of LF and discussions on how it should be used in specifying type theories can be found in Chapter 9 of [30]. The inference rules of LF are given in Figure 1, which include general rules, the rules for the kind of all types (**Type**, which represents the conceptual universe of types), and the rules for dependent product kinds of the form $(x{:}K)K'$ (kinds of functional operations). In the following, we give a brief introduction to LF and its use in specifying type theories, with discussions on several aspects with which we do not assume the familiarity of the reader.

Notation 1 In this paper, we shall use the following notational conventions:

- Equality signs: we shall use $M \equiv N$ for *syntactic identity*, meaning that M and N are the same up to α-conversion, and use $=$ for definitional and computational equality in type theory.
- 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 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'.
- Functional composition: for $f : (K_1)K_2$ and $g : (K_2)K_3$, define $g \circ f = [x{:}K_1]g(f(x)) : (K_1)K_3$, where x does not occur free in f or g.

2.1 Functional operations in LF

As in Martin-Löf's meaning explanation for his type theory, a functional operation of kind $(x{:}K)K'$ in LF can be applied to any object k of kind K to yield an object of kind $[k/x]K'$. The meaning of a functional operation is given by explaining its application results. For example, abstractions are special forms of functional operations whose meaning is essentially reflected by the definitional equality rule (β).

Remark 2 In LF, the functional operations that express abstraction are of the form $[x{:}K]k$, rather than the untyped $[x]k$ as found in Martin-Löf's logical framework. In other words, we regard the meta-level functional operations as having specific domains (and codomains).[1] This feature, as we shall see below, is essential in the formulation of coercive subtyping (see Section 3.2.4). It is also worth remarking that type-checking for the logical framework with untyped abstraction terms is not decidable (cf, [18]), while that for LF is.

The functional operations, in the form of abstraction or those introduced by declaring constants for a specified type theory (see below), are weakly extensional in the sense that the following rule is derivable by means of the (ξ) and (η) rules:

$$(Ext) \qquad \frac{\Gamma \vdash f : (x{:}K)K' \quad \Gamma \vdash g : (x{:}K)K' \quad \Gamma, x{:}K \vdash f(x) = g(x) : K'}{\Gamma \vdash f = g : (x{:}K)K'}$$

[1] One may want to consider a more philosophical argument of whether an operation should be considered as typed. See, for example, [7] for some relevant discussions.

Contexts and assumptions

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

General equality rules

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

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

Equality typing rules

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

Substitution rules

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

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

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

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

The kind Type

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

Dependent product kinds

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

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

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

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

Figure 1: The inference rules of LF.

This reflects the idea that LF provides meta-level schematic and definitional mechanisms, and the fact that definitional equality for abbreviations is weakly extensional (in particular, the η-rule holds). For example, as in ordinary mathematical practice, a definition $f(g,x) = g(x)$ has the same effect as $f(g) = g$. This is in contrast with the functions of Π-types in type theory for which η-rule should not hold since it makes little sense to be a computation rule (see below).

Remark 3 In fact, in the presence of the (β)-rule, the above rule (Ext) is equivalent to $(\eta)(\xi)$ as equational rules. However, it is easy to see that (Ext) cannot be used as a reduction rule because of the symmetry between f and g.

2.2 Specifying type theories in LF

In general, a specification of a type theory in the logical framework consists of a collection of declarations of new constants and a collection of computation rules. Formally, declaring a new constant k to be of kind K is to introduce the following inference rule to the specified type theory:

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

and, for a kind K which is either **Type** or of the form $El(A)$, one can assert computation rules by writing

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

which introduces the following rule for *computational equality*:

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

For example, one can introduce Π-types by declaring the following constants:

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

and asserting the following computation rule:

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

where $A : \mathbf{Type}$, $B : (A)\mathbf{Type}$, $C : (\Pi(A,B))\mathbf{Type}$, $f : (g{:}(x{:}A)B(x))C(\lambda(A,B,g))$, and $g : (x{:}A)B(x)$. Then, the usual application operator can be defined as

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

and we have $\mathbf{app}(A,B,\lambda(A,B,f)) = f$. However, the η-rule does not hold:

$$\lambda(A,B,\mathbf{app}(A,B,F)) \neq F,$$

when $F:\Pi(A,B)$ is a variable. In other words, functions of a Π-type are not weakly extensional. Note that a Π-type $\Pi(A,B)$ is different from the kind $(x{:}A)B(x)$ (or more formally, $(x{:}El(A))El(B(x))$), whose objects are weakly extensional functional operations.

Notation 4 For types A and B, when x does not occur free in B, we shall write $A \to B$ for $\Pi(A, [x{:}A]B)$, the type of functions from A to B. Similarly, $A \to B$ is different from the kind $(A)B$.

Remark 5 LF-presentations of type theories should be regarded as 'better' formulations as compared with their direct formulations. For instance, η-rules are not computational rules for Π-types; their corresponding propositional equalities are provable in the LF-presentation (eg, in UTT), while they are not for direct formulations (eg, for ECC). See [30] for more discussions.

One can similarly specify various types or type constructors, including inductive types, predicative or impredicative type universes, etc. [17, 20, 28]. The following gives a brief description of how inductive schemata can be introduced for introducing a large class of inductive data types (for formal details, see Chapter 9 of [30]), which we shall use later to introduce subtyping for parameterised inductive types.

First, we say that a kind is *small* if it is either of the form $El(A)$ or of the form $(x{:}K_1)K_2$ with K_1 and K_2 small kinds. An *inductive schema* Θ with respect to a type variable X is of one of the following forms:

1. $\Theta \equiv X$, or
2. $\Theta \equiv (x{:}K)\Theta_0$, where K is a small kind and Θ_0 is an inductive schema, or
3. $\Theta \equiv (\Phi)\Theta_0$, where Θ_0 is an inductive schema and Φ is of the form $(x_1{:}K_1)...(x_m{:}K_m)X$, with $m \geq 0$ and K_i being small kinds in which X does not occur free.

An inductive type $\mathcal{M}[\bar{\Theta}]$ is generated by a sequence of inductive schemata $\bar{\Theta} \equiv \Theta_1, ..., \Theta_n$ (with respect to the same type variable X which becomes bound in $\mathcal{M}[\bar{\Theta}]$). Associated with the inductive type are its introduction operators $\iota_i[\bar{\Theta}]$ $(i = 1, ..., n)$ and an elimination operator $\mathbf{E}[\bar{\Theta}]$ with appropriate computation rules.

For example, a type Nat of natural numbers can be defined as $Nat =_{\mathrm{df}} \mathcal{M}[\bar{\Theta}_0]$, where $\bar{\Theta}_0 \equiv X, (X)X$. The associated introduction operators are $0 =_{\mathrm{df}} \iota_1[\bar{\Theta}_0] : Nat$ and $succ =_{\mathrm{df}} \iota_2[\bar{\Theta}_0] : (Nat)Nat$, and the elimination operator is

$$\mathbf{E}_{Nat} =_{\mathrm{df}} \mathbf{E}[\bar{\Theta}_0] : (C{:}(Nat)\mathbf{Type})(c{:}C(0))(f{:}(x{:}Nat)(C(x))C(succ(x)))(n{:}Nat)C(n),$$

with the computation rules

$$\begin{array}{rcl} \mathbf{E}_{Nat}(C,c,f,0) & = & c \\ \mathbf{E}_{Nat}(C,c,f,succ(x)) & = & f(x, \mathbf{E}_{Nat}(C,c,f,x)) \end{array}$$

Note that, as discussed in [30], we can introduce *different* inductive types with isomorphic structure. For instance, a type $Even$ isomorphic to Nat can be introduced as $Even =_{\mathrm{df}} \mathcal{M}'[\bar{\Theta}_0]$, which is just another copy of Nat but with a different name and with different names for its introduction and elimination operators (e.g., $e_0 =_{\mathrm{df}} \iota'_1[\bar{\Theta}_0]$, $e_1 =_{\mathrm{df}} \iota'_2[\bar{\Theta}_0]$, and $\mathbf{E}_{Even} =_{\mathrm{df}} \mathbf{E}'[\bar{\Theta}_0]$). As we shall see below, with coercive subtyping, $Even$ can be regarded as a subtype of Nat—the type of even numbers.

The type theories specified in LF are intensional type theories, an example of which is the type theory UTT [30]. UTT consists of an impredicative type universe of propositions, inductive data types (and inductive families, not covered above), and predicative type universes. It has nice meta-theoretic properties such as Church-Rosser, Subject Reduction, and Strong Normalisation [21]. Implemented in the Lego proof development system, UTT has been applied to verification of functional programs [10, 11], imperative programs [42] and concurrent programs [44], specification and data refinement [29], and formalisation of mathematics [40].

2.3 Definitional equality and computational equality

We use LF seriously as a meta-level language (see Section 9.1.2 of [30] for a discussion). Along the same line, we make a distinction between the notion of definitional equality (abbreviational equality, reflected as $\beta\eta$-equality for functional operations in LF) and that of computational equality introduced by asserting computation rules when specifying a type theory.

This distinction is also reflected in our restriction above that new computation rules $(*)$ can only be asserted between two types or two objects of a type, but not between two functional operations of a dependent product kind. For instance, under this restriction, for the constant *app* declared as follows:

$$app : (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(\Pi(A,B))(x{:}A)B(x)$$

one cannot directly assert $app(A,B,\lambda(A,B,f)) = f$ as a computational equality, though asserting $app(A,B,\lambda(A,B,f),x) = f(x)$ is legitimate. However, taking this latter as a reduction rule (from the left to the right) is problematic since weak extensionality does not hold for the reduction relation and the Church-Rosser property would fail to hold. Another possibility, pointed out to me by Healfdene Goguen, is to declare the application operator in a slightly different way:

$$app : (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(x{:}A)(\Pi(A,B))B(x),$$

and to assert $app(A,B,x,\lambda(A,B,f)) = f(x)$.

As we shall see below, in the coercive subtyping approach, coercions between types introduce new definitional equalities since implicit coercions are essentially an apparatus for abbreviation. In other words, coercive subtyping is regarded as abbreviational mechanisms similar to definitional mechanisms. The choice of considering coercive subtyping in the meta-level logical framework, rather than in some object-level type systems is important and beneficial.[2]

[2]The so-called object-level type systems include, for example, Pure Type Systems [5] such as the Calculus of Constructions (CC) [16]. See [6] for an attempt to introduce coercions into PTS.

3 Coercive subtyping

In this section, we first introduce the basic ideas and give informal meaning explanations of the judgements in the extended framework with coercive subtyping. Then, a formal presentation is given, followed by a discussion of its properties.

3.1 Basic ideas and informal explanation

Introducing subtyping into dependent type theories with inductive types raises new issues that have not been considered before in research on subtyping for simpler type systems. We first consider the basic problems and introduce the idea of coercive subtyping.

3.1.1 Subtyping between inductive types: the problems

An inductive type can be understood as consisting of its canonical objects (values of the type). For instance, the type Nat of natural numbers consists of 0 and the successors, and any natural number is regarded as a representation of a canonical natural number, to which it can be computed. If A is a subtype of B, then every object of type A is (regarded as) an object of type B. One of the basic considerations in studying subtyping for inductive types is to look for a suitable approach with which the understanding of types based on the notions of canonical object and computation still applies.

The traditional approaches based on direct overloading (eg, overloading λ-terms to stand for objects of different function types) do not generalise to inductive types. A natural consideration might be to form a subtype A of type B by selecting some (canonical) objects from B, which are regarded as the (canonical) objects of A. For example, we may introduce a subtype $Even$ of Nat by declaring its canonical objects to be those natural numbers which are either 0 or of the form $succ(succ(e))$, where e is of type $Even$. However, in such a setting, type-checking is difficult (and in general undecidable). It is not clear how one may introduce suitable restrictions on subtype formation to ensure decidable type-checking. One suggestion that has been made in the literature is to specify a subtype by declaring its constructors to be a subset of the constructors of an existing supertype [15], but this would exclude even the example of $Even$ and other interesting applications of subtyping such as inheritance between mathematical theories represented as Σ-types.

A related problem is that, in the presence of subtyping, the usual elimination rules for inductive types become inadequate since they do not take into the account (the forms of) the canonical objects in the subtypes. For instance, subtyping between two Σ-types as found in the Extended Calculus of Constructions (ECC) [25, 26] is not quite compatible with the general elimination rules as found in Martin-Löf's type theory and UTT. A simple combination would lead to a system for which the subject reduction property fails to hold (see Section 4.3).

In general, there are two notions of subtyping that have been studied in the literature — one based on subset relationship (eg, between Even and Nat) and the other based on inheritance relationship (eg, between theories represented by Σ-types and between record types). Besides the above technical problems with subtyping for inductive types, it is in general unclear how the two notions of subtyping can be understood uniformly in a single framework. Coercive subtyping starts from a different basic concept of subtyping, ie, subtyping based on coercion and provides a setting to study subtyping in general.

3.1.2 Coercive subtyping: informal explanation

There are two basic ideas on which coercive subtyping is based: implicit coercion and coercive rules for definitional equality.

Implicit coercions

With coercive subtyping, for any proper subtype A of type B, there is a unique coercion c from A to B and every object a of type A can be regarded as the object $c(a)$ of type B, that is its image under the coercion. The coercion is implicit in the sense that one may use the object a to stand for $c(a)$ in an expression where an object of type B is expected (cf., the new application rules in Figure 3).

Subtyping relations between basic inductive types are specified by defining coercion functions, which should be definable in the type theory. For instance, the inductive type $Even$ with constructors e_0 and e_1, as specified in Section 2.2, can be introduced as an (inductive) subtype of Nat (i.e., $Even < Nat$), by giving the coercion defined by means of structural recursion over $Even$ as follows: $c(e_0) = 0$ and $c(e_1(e)) = succ(succ(c(e)))$. These basic coercions can then be generalised to other (structured) types.

Judgements and their meaning explanation

When subtyping is introduced, a type theory does not have the property of unique typing (type uniqueness) anymore. Instead, a notion of *principal typing* is the best that one can expect (see, eg, [26] for a definition of the notion of principal type for ECC). Intuitively, K is a *principal kind* of object k if and only if k is of kind K and, for any kind K', k is of kind K' if and only if K is a subkind of K'. Note that being a principal kind is more than just being a minimal or minimum kind, and in general, a principal kind of an object is unique up to the computational equality.

In a type theory specified in LF with coercive subtyping, besides judgements for context validity, we have the following forms of judgements as in LF (but with new meaning explanations):

- K **kind** asserts that K is a *kind*.
- $k : K$ asserts that K is the *principal kind* of k. When $K \equiv El(A)$, it asserts that A is the *principal type* of k, which means that k computes to a canonical object of type A.
- $K = K'$ asserts that K and K' are equal kinds, which means that $k : K$ if and only if $k : K'$.
- $k = k' : K$ asserts that k and k' are equal objects with principal kind K. When $K \equiv El(A)$, it means that k and k' are computationally equal and compute to the same canonical object of type A.

With coercive subtyping, if kind K is a proper subkind of kind K', then there is a *coercion* from K to K', which is a functional operation $c : (K)K'$ and, as coercion, is *unique* up to computational equality. We have two new forms of judgements in our theory with coercive subtyping, the subkinding judgements and the subtyping judgements:

- $K <_c K'$ asserts that kind K is a *proper subkind* of kind K' with a unique coercion c.
- $A <_c B : \mathbf{Type}$ asserts that type A is a *proper subtype* of type B with a unique coercion c, which means that $El(A) <_c El(B)$.

The above are called the *basic forms of judgements*. With them, we can consider several *definable forms of judgements* which include the following:

- $K < K'$ (K is a proper subkind of K') stands for '$K <_c K'$ for some c'.
- $K \leq K'$ (K is a subkind of K') stands for '$K = K'$ or $K <_c K'$ for some c'.
- $k :: K$ (k is of kind K) stands for '$k : K$ or $k : K_0$ for some K_0 such that $K_0 <_c K$ for some c'.
- $k = k' :: K$ (k and k' are equal objects of kind K) stands for '$k = k' : K$ or $k = k' : K_0$ for some K_0 such that $K_0 <_c K$ for some c'.

Note that $k : K$ is now the judgement form for *principal* kinding/typing, while the usual kinding/typing judgement, asserting that k is an object of kind K, is represented by the definable judgement $k :: K$. Also, $K <_c K'$ is the judgement form for *proper* subkinding with specified coercion c, while the usual subkinding/subtyping judgements are defined via those with coercions. If $K < K'$, then the coercion from K to K' should be unique up to computational equality; in other words, if $K <_c K'$ and $K <_{c'} K'$, then $c = c' : (K)K'$.

The meaning explanations of judgements as sketched above coherently extends the meaning theory developed by Martin-Löf for intensional type theories to coercive subtyping. Our formulation of coercive subtyping is strongly guided by such a meaning explanation.[3] The meaning explanation for the product kinds and functional operations is now extended as follows. A functional operation f of kind $(x{:}K)K'$ is an object that can be applied to any object k of kind K, including those whose principal kinds are proper subkinds of K. If $k : K_0 <_c K$, then the result is K, then the result of the application $f(k)$ is the same as that of $f(c(k))$, ie, the application of f to the image of k under the coercion c from K_0 to K. This meaning explanation is reflected in our formal system as the new application rules and the coercive definition rule in Figure 3; it is the latter of which we explain below.

[3] I should note that, unlike Per Martin-Löf, we do not exclude the possibility of giving coherent meaning-theoretic explanations to impredicative type theories where there is an internal totality of logical propositions, which include, for example, the type theory UTT and various Pure Type Systems. A detailed analysis on this is out of the scope of this paper.

The coercive definition rule

To give meanings for the objects of a proper subkind being in a superkind, we introduce *coercive definition rules*. One of the key points is to design suitable coercive definition rules so that the resulting type theory reflects the intended meanings of judgements and has nice proof-theoretic properties. The following is the basic coercive definition rule:

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

Intuitively, when a functional operation with domain K is applied to an object k_0 whose principal kind is a proper subkind of K, $f(k_0)$ is definitionally equal to $f(c(k_0))$, that is, the application result is the same as that of applying f to the image of k_0 under the intended coercion.

Remark 6 As to be made precise below, we do not have any coercions (including the indentity function) from a type to itself (in the above rule, K_0 is a *proper* subkind of K). Because the notion of coercive subtyping is not based on that of subset relationship between sets of terms, type equality and proper subtyping are very different. This is why we consider proper subtyping as more basic than the relation of "$\leq$" as basic.

Note that the above rule exactly conforms to our meaning explanation for functional operations f, since in the presence of subkinding, the domain of a functional operation also has as objects those of any of its subkinds and the coercive rules explain the effect of applying the functional operation to an object of a proper subkind of its domain. Note that we do *not* have the following rule:

$$(**) \qquad \frac{k\ :\ K \qquad K <_c K'}{k = c(k)\ ::\ K'}$$

This rule for dependent functional operations is not appropriate as meaning-giving, since the meaning of a functional operation is not given by its canonical form, but rather by its behaviour when applied to its arguments (this is exactly captured by our coercive definition rule). We note that the coercive definition rule preserves principal kinds when regarded as reduction rules from the left to the right (i.e., the subject reduction property with respect to principal kinding), while the above rule $(**)$ does not. Furthermore, taking the above rule $(**)$ as a reduction rule could also lead to infinite reduction sequences.

Remark 7 Note that the rule $(**)$ makes an essential use of the judgement form $k\ ::\ K$; in other words, it cannot be reasonably stated with only the basic judgement forms. In [32], where coercive subtyping was first introduced, $k\ ::\ K$ is taken as a basic judgement form. The presentation of coercive subtyping in this paper makes it clearer from another angle that the $(**)$ rule is not adequate.

3.2 Coercive subtyping: a formal presentation

In this section, we give a formal presentation of type theories with coercive subtyping. We consider how to extend any type theory T specified in LF with coercive subtyping.

Examples of such type theories include Martin-Löf's intensional type theory, UTT, and many others.

Let T be any type theory specified in LF. We shall present the system $T[\mathcal{R}]$, the extension of T with coercive subtyping, 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 $T[\mathcal{R}]_0$.

3.2.1 $\mathbf{T}[\mathcal{R}]_0$ and the subtyping relation

As a formal system, $T[\mathcal{R}]_0$ is an extension of T (only) with the subtyping judgement form $\Gamma \vdash A <_c B : \mathbf{Type}$ and the following rules:

- A set $\mathcal{R}$ of basic subtyping rules whose conclusions are subtyping judgements of the form $\Gamma \vdash A <_c B : \mathbf{Type}$.
- The general subtyping rules in Figure 2.

Congruence rule

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

Transitivity rule

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

Substitution rule

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

Figure 2: General subtyping 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, we have the following lemma.

Lemma 8 (conservativity of $\mathbf{T}[\mathcal{R}]_0$ over T) $T[\mathcal{R}]_0$ is a conservative extension of T; that is, if J is not of the form $A <_c B : \mathbf{Type}$, then $\Gamma \vdash J$ is derivable in T if and only if $\Gamma \vdash J$ is derivable in $T[\mathcal{R}]_0$.

Definition 9 (coherence condition) We say that the basic subtyping rules are coherent if $T[\mathcal{R}]_0$ has the following coherence properties:

1. If $\Gamma \vdash A <_c B : \mathbf{Type}$, then $\Gamma \vdash A : \mathbf{Type}$, $\Gamma \vdash B : \mathbf{Type}$, and $\Gamma \vdash c : (A)B$.

2. $\Gamma \not\vdash A <_c A : \mathbf{Type}$ for any Γ, A and c.

3. If $\Gamma \vdash A <_c B : \mathbf{Type}$ and $\Gamma \vdash A <_{c'} B : \mathbf{Type}$, then $\Gamma \vdash c = c' : (A)B$.

The above conditions are the most basic requirements for the basic subtyping rules.

Remark 10 Some remarks on the basic subtyping rules are worth making.

- 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 (see Section 4). For most of the applications, these coercions are introduced between *data types*, rather than between logical propositions (eg, propositions in an impredicative type theory), although the latter is formally possible and can be used in analysing subtyping in different systems.

- The basic subtyping rules may include those as introduced by a set $\mathcal{C}$ of basic coercions between closed types in T:

$$\frac{\Gamma\ \mathbf{valid} \quad (A, c, B) \in \mathcal{C}}{\Gamma \vdash A <_c B : \mathbf{Type}}$$

  This would correspond to the system in [32], where coercive subtyping was first introduced. The framework presented here is more general. In particular, families of coercions such as parameterised coercions c in $\bar{x}{:}\bar{A} \vdash A[\bar{x}] <_{c[\bar{x}]} B[\bar{x}] : \mathbf{Type}$ are allowed, as long as they, together with other coercions, satisfy the coherence conditions.

3.2.2 The system $\mathbf{T}[\mathcal{R}]$

Let $\mathcal{R}$ be a set of coherent basic subtyping rules. The system $\mathrm{T}[\mathcal{R}]$, the extension of T with coercive subtyping with respect to $\mathcal{R}$, is the system obtained from $\mathrm{T}[\mathcal{R}]_0$ by adding the new subkinding judgement form $\Gamma \vdash K <_c K'$ and the rules in Figure 3.

Remark 11 If one wants to add any or all of the definable forms of judgement considered Section 3.1.2, one can add their characterisation rules, which are given in Figure 4.

3.2.3 Explanation and remarks

Here, we give informal explanations and some remarks. We first emphasise that the judgement $\Gamma \vdash k : K$ means that k is an object with *principal* kind K, while the definable judgement $\Gamma \vdash k :: K$ means that k is of kind K (see informal meaning explanations in the above section). This reflects the fact that, when no subtyping is present, the notions of typing and principal typing coincide.

The new rules for application in Figure 3, together with those in LF, allow a functional operation $f : (x{:}K)K'$ to be applied to any object k of kind K (ie, $k :: K$), whose principal kind is either K or a proper subkind of K. And, as explained above, the coercive definition rule gives the meaning to the object formed by an application in the latter case.

New rules for application

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

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

Coercive definition rule

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

Basic subkinding rule

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

Subkinding for dependent product kinds

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

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

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

Congruence rule for subkinding

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

Transitivity rule for subkinding

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

Substitution rule for subkinding

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

Figure 3: New inference rules in $\mathrm{T}[\mathcal{R}]$.

Characterisation rules for $<$ and $\leq$

$$\frac{\Gamma \vdash K <_c K'}{\Gamma \vdash K < K'} \quad \frac{\Gamma \vdash K = K'}{\Gamma \vdash K \leq K'} \quad \frac{\Gamma \vdash K <_c K'}{\Gamma \vdash K \leq K'}$$

Characterisation rules for $::$

$$\frac{\Gamma \vdash k \ : \ K}{\Gamma \vdash k \ :: \ K} \quad \frac{\Gamma \vdash k \ : \ K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash k \ :: \ K}$$

$$\frac{\Gamma \vdash k = k' \ : \ K}{\Gamma \vdash k = k' \ :: \ K} \quad \frac{\Gamma \vdash k = k' \ : \ K_0 \quad \Gamma \vdash K_0 <_c K}{\Gamma \vdash k = k' \ :: \ K}$$

Figure 4: Characterisation rules for definable judgemnts.

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. (See Section 4 for examples and further discussions.) The basic coercions are extended to dependent product kinds. For example, for $Even <_c Nat$, we shall have $(Nat)\mathbf{Type} <_{c'} (Even)\mathbf{Type}$, where $c'(f) =_{\mathrm{df}} [x{:}Even]f(c(x))$.

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 $K <_c K'$ and $K <_{c'} K'$, then we have $c = c' : (K)K'$.[4] This also conforms to our meaning explanation of subtyping judgements.

It is obvious that T is a subsystem of T[$\mathcal{R}$]; for instance, if $\Gamma \vdash^{T} k : K$, then $\Gamma \vdash^{T[\mathcal{R}]} k \ : \ K$. We note that when $\mathcal{R}$ is empty (there is no basic subtyping rule), we have: $\Gamma \vdash^{T[\mathcal{R}]} k \ : \ K$ iff $\Gamma \vdash^{T[\mathcal{R}]} k \ :: \ K$ iff $\Gamma \vdash^{T} k : K$; in this sense, T[$\emptyset$] is just the original type theory T. When $\mathcal{R}$ is not empty, there are in general more typable terms in T[$\mathcal{R}$] than in T.

3.2.4 Typed equality and typed reduction

The above gives an equational presentation of the extension of type theory with coercive subtyping. The intended notion of computation for the extended type theory is the notion of *typed reduction*. The typed reduction relation $\Rightarrow$ is the reflexive and transitive closure generated from the rules in Figure 5 and the rules for computational equality (ie, the $(*)$ rules in Section 2) taken as reduction rules from the left to the right.

[4] This was pointed out to me by Sergei Soloviev. We have: $c = [x{:}K]c(x) = [x{:}K]([y{:}K']y)(c(x)) =$

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

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

$$\frac{\Gamma \vdash f \Rightarrow f' \quad \Gamma \vdash k_1 \Rightarrow k_2}{\Gamma \vdash f(k_1) \Rightarrow f'(k_2)} \qquad (\Rightarrow_c)\ \frac{\Gamma \vdash f \ : \ (x{:}K)K' \ \Gamma \vdash k_0 \ : \ K_0 \ \Gamma \vdash K_0 <_c K}{\Gamma \vdash f(k_0) \Rightarrow f(c(k_0))}$$

Figure 5: Typed reduction.

It is worth remarking that, besides the coercive definition rule, the (β) and (η) rules for definitional equality and those for computational equality are all governed by principal kinding requirements, which prevent unintended equalities or reductions. For instance, it is well known that, in the presence of subtyping and untyped $\beta\eta$-reductions, Church-Rosser fails even for well-typed terms. Typical examples include the terms such as $t \equiv [x{:}Even]([y{:}Nat]y)(x)$ with $Even < Nat$ (and $Even \neq Nat$). As illustrated in Figure 6, with untyped reduction, t β-reduces to $t_1 \equiv [x{:}Even]x$ and η-reduces to $t_2 \equiv [y{:}Nat]y$. However, in our system, we only have

$$t = [x{:}Even]([y{:}Nat]y)(c(x)) = [x{:}Even]c(x) = c,$$

where $c : (Even)Nat$ is the coercion from $Even$ to Nat and, in particular, t is not equal to $[x{:}Even]x$ or $[y{:}Nat]y$, which have principal kinds $(Even)Even$ and $(Nat)Nat$, respectively; they are different from $(Even)Nat$, the principal kind of t. This example also shows that untyped reduction does not preserve principal kinding. However, if we apply the coercions c_1 from $(Even)Even$ to $(Even)Nat$ and c_2 from $(Nat)Nat$ to $(Even)Nat$, to the incompatible terms t_1 and t_2, respectively, both result in terms that reduce to the term $[x{:}Even]c(x)$ under typed reduction; in other words, the intended reduction result is recovered.

It is interesting to note that certain subtyping relations between types can introduce forms of self-application. An example of this is to consider some types A and B and the basic subtyping relations such that $A <_{c_A} (A \to A) <_{c_{AB}} (B \to B) <_{c_B} B$, where $B \not\leq A$. (Note that the arrow here is the constructor for function *types*, not functional kinds. For any kind K, it is impossible to have in our system, for example, $K < (K)K$ or $(K)K < K$.) Then, for any $x{:}A$, $\mathbf{app}(A, [y{:}A]A, x, x)$ is of type A, because the first occurrence of x is of type $A \to A$ and the second of type A. Now, consider the following term M, which stands for $\lambda x{:}A.xx$ in the usual λ-notation:

$$M \equiv \lambda(A, [y{:}A]A, [x{:}A]\mathbf{app}(A, [y{:}A]A, x, x)),$$

we have that M is of type $A \to A$ and $Y \equiv \mathbf{app}(B, [y{:}B]B, M, M)$ is of type B. It is obvious that the term Y computes to itself under untyped reduction and hence has an infinite reduction sequence. Under typed reduction, however, the term Y cannot

$[x{:}K]([y{:}K']y)(x) = [x{:}K]([y{:}K']y)(c'(x)) = [x{:}K]c'(x) = c'$.

$$
\begin{tikzcd}
 & {[y{:}Nat]y} \arrow[dr, "c_1"] & & \\
{[x{:}Even]([y{:}Nat]y)(x)} \arrow[ur, "\text{untyped } \beta"] \arrow[dr, "\text{untyped } \eta"'] \arrow[r, "\Rightarrow_c"] & {[x{:}Even]([y{:}Nat]y)(c(x))} \arrow[r, "\Rightarrow_\beta"] & {[x{:}Even]c(x)} \arrow[r, "\Rightarrow_\eta"] & c \\
 & {[x{:}Even]x} \arrow[ur, "c_2"'] & &
\end{tikzcd}
$$

Figure 6: Reduction behaviour for $\beta\eta$: an example.

compute to itself, because the principal kinding requirement is not satisfied for the reduction to happen; instead, Y must first compute to $\mathbf{app}(B, [y{:}B]B, c_{AB}(M), c_B \circ c_{AB}(M))$, with the appropriate coercions inserted.

A systematic meta-theoretic study of proof-theoretic properties of coercive subtyping is out of the scope of this paper. When the type theory T has nice proof-theoretic properties (e.g., when T is UTT or Martin-Löf's intensional type theory), the typed reduction for T[$\mathcal{R}$] is expected to have good properties as well. There is a coercion completion mapping δ from T[$\mathcal{R}$] to T that inserts all of the appropriate coercions and we have $M \Rightarrow \delta(M)$. It is then clear that the extended type system is weakly normalising if type theory T is. Typed reduction also preserves principal kinding (and principal typing). We conjecture that T[$\mathcal{R}$] satisfies the strong normalisation property with respect to the typed reduction, if the original type theory T does.

4 Reasoning with coercive subtyping and applications

In this section, it is shown that the simple framework of coercive subtyping provides a powerful setting in reasoning about subsets of objects, representing inheritance between formal theories in proof development, understanding various forms of coercions as implemented in proof systems, and understanding some issues in implementing type theories such as universe inclusion, overloading, and type-casting.

4.1 Subtyping between basic inductive types

Besides our earlier example of even and natural numbers, one meets many other applications where subtyping between basic inductive types is useful in practice. For instance, in a formalisation of the syntax of an imperative programming language, the primitive statements form an inductive type which is a subtype of the inductive type of programs.

Coercive subtyping represents such subtyping relations in a natural way and provides new power for expressing and reasoning about subsets in a more concise way. The following gives a simple illustrative example how the induction principles of subtypes can be used to reason about corresponding objects in a supertype. For instance, the induction principle for $Even$ can be used to show that every even natural number of type Nat has certain properties. Let us consider the proof of showing the simple property that 'none of the even natural number is equal to one'. Let D be the predicate defined over natural numbers as $D(n) \equiv (n \neq_{Nat} succ(0))$, where $=_{Nat}$ is a propositional equality over natural numbers. First, without subtyping, we would have to define a predicate $PEven$ over Nat and the above statement can be expressed as the proposition $\forall n{:}Nat.\ PEven(n) \supset D(n)$. With coercive subtyping, it can be expressed more concisely as $\forall e{:}Even.D(e)$, which can then be proved by means of the induction principle expressed by $\mathbf{E}_{Even}$. In such a proof, we only have to consider the base case $D(e_0) = D(0)$ and the induction step $(e{:}Even)(D(e))D(e_1(e)) = (e{:}Even)(D(c(e)))D(succ(succ(c(e))))$, which only concern with the even natural numbers. Note that, without subtyping, to use the induction principle for Nat to prove the statement involving $PEven$, we would have to consider the cases for the other natural numbers (the odd ones) as well.

Such a power in reasoning about subsets of objects through coercive subtyping comes from an effective use of computational functions (coercions) to represent subsets. This represents a typical advantage of type theory being also a computational language, as compared with traditional logical systems.

We note that not every predicate P over Nat can be represented as a function from Nat to Nat in the sense that the images of the function are exactly those natural numbers that satisfy P. In general, not every subset of a type can be represented as a subtype as above to assist inductive reasoning. For every predicate P over a type A, the Σ-type $\Sigma(A, P)$ can be regarded as a subtype of A with the first projection as coercion. However, for example, the subtype $\Sigma(Nat, PEven)$ does not help us reason about even numbers inductively as we have shown above.

4.2 Subtyping between parameterised inductive types

Another class of subtyping relations are between parameterised inductive types. For instance, if $A \leq B$, $List(A) \leq List(B)$. Similarly, since Σ-types are a special kind of inductive types, we naturally expect that, e.g., if $A \leq A'$ and $B \leq B'$, then $A \times B \leq A' \times B'$. In a proof development system, it is natural to assume such extensions of the basic subtyping relation, unless requested otherwise by the user (e.g., by giving a specific coercion between two Σ-types). In [28] we have suggested how subtyping can in general be extended to inductive types on the basis of the inductive schemata. Here, we extend the idea to coercive subtyping, which gives a systematic extension of the user-specified subtyping to structured inductive types.

Consider two inductive types with the same number of constructors (introduction operators): $A \equiv \mathcal{M}[\bar{\Theta}]$ and $A' \equiv \mathcal{M}[\bar{\Theta}']$ (see Section 2.2 and Chapter 9 of [30] for the notations of inductive types used here). Intuitively, A can be regarded as a subtype of A', notation $A \leq_p A'$, if the corresponding constructors have the same number of parameters and any parameter kind of a constructor of A is a subkind of that of the corresponding parameter for A'. To give a more precise definition of this, we can first consider a binary relation $\leq_s$ between inductive schemata as the smallest relation generated by the following rules, where Θ, Θ_0, and Θ_0' stand for arbitrary inductive schemata with respect to type variable X:

$$\frac{}{\Theta \leq_s \Theta} \qquad \frac{K \leq K' \quad \Theta_0 \leq_s \Theta_0'}{(x{:}K)\Theta_0 \leq_s (x{:}K')\Theta_0'} \qquad \frac{\Phi \leq \Phi' \quad \Theta_0 \leq_s \Theta_0'}{(\Phi)\Theta_0 \leq_s (\Phi')\Theta_0'}$$

Let $\bar{\Theta} \equiv \Theta_1, ..., \Theta_n$ and $\bar{\Theta}' \equiv \Theta_1', ..., \Theta_n'$ be sequences of inductive schemata of the same length. Then, define $\bar{\Theta} \leq \bar{\Theta}'$ if and only if $\Theta_i \leq \Theta_i'$ for $i = 1, ..., n$, and $\bar{\Theta} < \bar{\Theta}'$ if and only if $\bar{\Theta} \leq \bar{\Theta}'$ and $\Theta_i \neq \Theta_i'$ for some i. Then, we can define: $\mathcal{M}[\bar{\Theta}] <_p \mathcal{M}[\bar{\Theta}']$ if and only if $\bar{\Theta} < \bar{\Theta}'$. The implicit coercions for the subtyping relation $<_p$ can be defined straightforwardly. The subscript in the relations $\leq_p$ and $<_p$ is to indicate that this is not necessarily a subtyping relation used in practice because, for example, user-defined implicit coercions may (and should) have higher priority and therefore override such a 'default' relation between inductive types.

Note that K and K' in the second rule above are both small kinds; we can define the above subtyping relation by means of basic subtyping rules. Without giving the general rules here, we illustrate this by the following examples.

- Lists: $List =_{\mathrm{df}} [A{:}\mathbf{Type}]\ \mathcal{M}[X, (A)(X)X]$, with constructors nil and $cons$. We have $List(A) \leq_p List(B)$ if $A \leq B$. We can introduce the following basic subtyping rule for lists:

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

  where $c_L(nil(A)) = nil(B)$ and $c_L(cons(A, a, l)) = cons(B, c(a), c_L(l))$.

- Σ-types: $\Sigma =_{\mathrm{df}} [A{:}\mathbf{Type}][B{:}(A)\mathbf{Type}]\ \mathcal{M}[(x{:}A)(B(x))X]$, with constructor $pair_\Sigma$. We have $\Sigma(A, B) \leq_p \Sigma(A', B')$ if $A \leq A'$ and $B(x) \leq B'(x)$. We can introduce two basic subtyping rules:

$$\frac{\Gamma \vdash B : (A)\mathbf{Type} \quad \Gamma \vdash B' : (A)\mathbf{Type} \quad \Gamma, x{:}A \vdash B(x) <_{c[x]} B'(x) : \mathbf{Type}}{\Gamma \vdash \Sigma(A, B) <_{c_1} \Sigma(A, B') : \mathbf{Type}}$$

  where $c_1(pair_\Sigma(A, B, a, b)) = pair_\Sigma(A, B', a, c[a](b))$, and

$$\frac{\Gamma \vdash A <_c A' : \mathbf{Type} \quad \Gamma \vdash B' : (A')\mathbf{Type}}{\Gamma \vdash \Sigma(A, B' \circ c) <_{c_2} \Sigma(A', B') : \mathbf{Type}}$$

  where $c_2(pair_\Sigma(A, B' \circ c, a, b)) = pair_\Sigma(A', B', c(a), b))$. Note that, by the coercive definition rule, $\Sigma(A, B') = \Sigma(A, B' \circ c) : \mathbf{Type}$ if $A <_c A' : \mathbf{Type}$ and $B' : (A')\mathbf{Type}$.

- Π-types: $\Pi =_{\mathrm{df}} [A{:}\mathbf{Type}][B{:}(A)\mathbf{Type}]\ \mathcal{M}[((x{:}A)B(x))X]$, with constructor λ. We have $\Pi(A, B) \leq_p \Pi(A', B')$ if $(x{:}A)B(x) \leq (x{:}A')B'(x)$. We can introduce basic subtyping rules for Π-types as follows:

$$\frac{\Gamma \vdash B : (A)\mathbf{Type} \quad \Gamma \vdash B' : (A)\mathbf{Type} \quad \Gamma, x{:}A \vdash B(x) <_{c[x]} B'(x) : \mathbf{Type}}{\Gamma \vdash \Pi(A,B) <_{c_1} \Pi(A,B') : \mathbf{Type}}$$

where $c_1(\lambda(A,B,f)) = \lambda(A,B',[x{:}A]c[x](f(x)))$.

$$\frac{\Gamma \vdash A' <_c A : \mathbf{Type} \quad \Gamma \vdash B : (A)\mathbf{Type}}{\Gamma \vdash \Pi(A,B) <_{c_2} \Pi(A', B \circ c) : \mathbf{Type}}$$

where $c_2(\lambda(A,B,f)) = \lambda(A', B \circ c, f \circ c)$. Note that, by the coercive definition rule, $\Pi(A',B) = \Pi(A', B \circ c) : \mathbf{Type}$ if $A' <_c A : \mathbf{Type}$ and $B : (A)\mathbf{Type}$.

- Disjoint union types: $+ =_{\mathrm{df}} [A{:}\mathbf{Type}][B{:}\mathbf{Type}]\ \mathcal{M}[(A)X,(B)X]$, with constructors *inl* and *inr*. We have $A + B \leq_p A' + B'$ if $A \leq A'$ and $B \leq B'$. We can introduce

$$\frac{\Gamma \vdash A <_c A' : \mathbf{Type} \quad \Gamma \vdash B : \mathbf{Type}}{\Gamma \vdash A + B <_{c_1} A' + B : \mathbf{Type}} \qquad \frac{\Gamma \vdash A : \mathbf{Type} \quad \Gamma \vdash B <_c B' : \mathbf{Type}}{\Gamma \vdash A + B <_{c_2} A + B' : \mathbf{Type}}$$

  where $c_1(inl(A,B,a)) = inl(A',B,c(a))$, $c_1(inr(A,B,b)) = inr(A',B,b)$, and c_2 is defined similarly.

- Branching trees: $Tree =_{\mathrm{df}} [B{:}\mathbf{Type}]\mathcal{M}[X,((B)X)X]$, with constructors *empty* and *mk*. We have $Tree(B) \leq_p Tree(B')$ if $B' \leq B$. We can introduce the following basic subtyping rule:

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

  where $c'(empty(B')) = empty(B)$ and $c'(mk(B',f)) = mk(B, f \circ c)$.

For instance, with $Even < Nat$, we have $List(Even) < List(Nat)$ and $Tree(Nat) < Tree(Even)$. It can be shown that such basic subtyping rules for the parameterised inductive types are coherent. In practice, they may be adopted as a default generalisation of the user-specified basic subtyping relation between basic inductive types, with appropriate overriding by the latter (eg, a user-defined coercion between $\Sigma(Even, B)$ and $\Sigma(Nat, B)$ will override the default coercion generated from the coercion between $Even$ and Nat).

4.3 Subtyping between type universes

The inclusion relation between type universes may either be introduced by means of explicit lifting operators (e.g., adopted in the presentation of UTT in [30]) or direct subtyping (e.g., adopted in the presentation of the Extended Calculus of Constructions [26] and used in systems such as Lego). These are called by Martin-Löf as universes à la Tarski and universes à la Russell, respectively [36]. The latter approach is based on overloading common term operators and is not quite compatible with the elimination rules when general inductive types are introduced; in particular, the subject reduction property would fail to hold. For instance, for the product types with introduction operator $pair_\times$ and elimination operator $\mathbf{E}_\times$, we would have, in the following context,

$$x,y{:}Type_0,\ C{:}(Type_1 \times Type_1)\mathbf{Type},\ f{:}(x,y{:}Type_1)C(pair_\times(Type_1,Type_1,x,y)),$$

that the following well-typed term (since $Type_0 \times Type_0 < Type_1 \times Type_1$)

$$p \equiv \mathbf{E}_\times(Type_1,Type_1,C,f,pair_\times(Type_0,Type_0,x,y)),$$

computes to $f(x,y)$, which is of type $C(pair_\times(Type_1,Type_1,x,y))$ but not of type $C(pair_\times(Type_0,Type_0,x,y))$, a type of p. This has caused a problem in extending subtyping to parameterised inductive types in systems such as Lego. The reason is essentially that the formulation of the elimination rule has not taken into the account that, with subtyping, a supertype also contains canonical objects of its subtypes.[5]

With coercive subtyping, a natural bridge between the (more semantics-oriented) formulation à la Tarski and the more practically useful formulation à la Russell can be established because we can declare subtyping relations between universes and take the explicit lifting operators as the intended implicit coercions. For example, inclusions between predicative universes in UTT are introduced by means of explicit lifting operators as follows (here we omit the introduction of names of inductive types in universes):

$$Type_i : \mathbf{Type},\quad type_i : Type_{i+1},$$

$$\mathbf{T}_i : (Type_i)\mathbf{Type},\quad \mathbf{T}_{i+1}(type_i) = Type_i : \mathbf{Type},$$

$$\mathbf{t}_{i+1} : (Type_i)Type_{i+1},\quad \mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) = \mathbf{T}_i(a) : \mathbf{Type}.$$

We can introduce basic subtyping rules

$$\frac{\Gamma\ \ \mathbf{valid}}{\Gamma \vdash Type_i <_{\mathbf{t}_{i+1}} Type_{i+1} : \mathbf{Type}}$$

This fits well with the implementation of subtyping in proof development systems such as Lego and solves the above problem.

4.4 Interpreting practical forms of coercion

Anthony Bailey has implemented various interesting and useful coercion mechanisms in Lego [3]. (Also see [41] for a related development in the Coq system.) Bailey has given interesting examples to illustrate the use of implicit coercion to make proof development easier (more readable etc.). We briefly discuss how these basic coercion mechanisms may be understood in our setting of coercive subtyping.

On the basis of the Lego system, the implementation of which is not based on a logical framework description of type theory, Bailey has introduced several kinds of implicit coercions, called *argument coercion*, *kind coercion*, and Π-*coercion*. Argument-coercion deals with application of a function of a Π-type $\Pi x{:}A.B(x)$ to an object a whose principal type is a subtype of A. This is similar to our treatment of the application for dependent product kinds. In our setting, argument coercion for Π-types can be understood as a special case when applying $\mathbf{app}(A,B,f)$ to a.

[5]Furthermore, it is not clear how it can be modified to do so, since it may require a substantial extension with new forms of judgement. We do not discuss this issue here.

Bailey's notion of kind-coercion may better be called type-coercion in our terminology. It is introduced to deal with the following situation. For instance, suppose *Group* is the Σ-type representing the theory of groups and $G : Group$. One may hope to write, for example, $\Pi x{:}G.C(x)$ for 'for all x in (the carrier type of) G, $C(x)$', though this is not allowed since G is not its carrier type. This becomes possible when a kind-coercion el from *Group* to types in a universe U is declared to obtain the carrier type of every group structure, and the above Π-type would stand for $\Pi x{:}el(G).C(x)$. Kind-coercion can be understood as a special case of argument coercion in our formulation. For instance, for the above example, $\Pi x{:}G.C(x)$ abbreviates $\Pi(G, C) = \Pi(el(G), C)$ when $Group < U$ with coercion el.

The notion of Π-coercion is to deal with the cases where an object F supposed to be of a Π-type is applied to an object but F does not have the right Π-type. This can also be regarded as a special case of coercive definition: with an implicit coercion c from the type of F to the expected Π-type $\Pi(A, B)$, we have $\mathbf{app}(A, B, F, a) = \mathbf{app}(A, B, c(F), a)$.

4.5 Type-casting and overloading

In a private communication with the author, Peter Aczel has recently pointed out a close relationship between coercive subtyping and type-casting. Type-casting is a way to form terms in proof systems such as Lego, which was introduced primarily for dealing with type ambiguity introduced by subtyping (eg, universe inclusion) or omission of syntax to provide user-friendliness. For example, in the Lego system, one may write a term of the form `a:A`, which stands for 'the term `a` with principal type `A`'.

We can understand `a:A` as the term $([x{:}A]x)(a)$ in our system. When $a : A_0$ and $A_0 < A$, we have, by the coercive definition rule and (β), $([x{:}A]x)(a) = c(a)$, whose principal type is A. For instance, with universe subtyping introduced above, we will have $(type_0{:}Type_3) = ([x{:}Type_3]x)(type_0) = \mathbf{t}_3(\mathbf{t}_2(type_0))$.

In other words, in a type theory where every object has a principal type, type-casting is definable with coercive subtyping. Alternatively, on the basis of the above understanding, one may directly introduce type-casting as basic terms and introduce the following rules (modified from a suggestion by Aczel), which analyse the coercive definition rules into two parts:

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

These rules reflect the meaning of type-casting directly. If $(k_0{:}K)$ is defined as $([x{:}K]x)(k_0)$, the above rules are all derivable.

Type-casting pairs: an example

The above understanding can be used to understand the use of type-casting in resolving the possible type ambiguity of pairs (objects of Σ-types) as well. For instance, for $a : A$, the (untyped) pair (A, a) may have type $Type_0 \times A$ or $\Sigma X{:}Type_0.X$, which are incompatible. In the Lego system, the former is the default type and if it is the second type that is the intended one, one has to use type-casting to write explicitly

`(A,a : <X:Type(0)>X)`. Such a decision seems to be ad hoc, though very useful in practice.

To analyse this problem with coercive subtyping, we can consider a special family of unit types $\mathbf{1}(A,a)$ indexed by types $A : \mathbf{Type}$ and objects $a : A$, each of which has only one constructor $*(A,a) : \mathbf{1}(A,a)$. Then, for any type A, any family of types $B : (A)\mathbf{Type}$, and any $a : A$, we can define two coercions, c_1 from $\mathbf{1}(A,a) \times B(a)$ to $A \times B(a)$ and c_2 from $\mathbf{1}(A,a) \times B(a)$ to $\Sigma(A,B)$ (ie, $\Sigma x{:}A.B(x)$), as follows:

$$\begin{array}{rcl} c_1(pair_\times(\mathbf{1}(A,a), B(a), *(A,a), b)) & = & pair_\times(A, B(a), a, b), \\ c_2(pair_\times(\mathbf{1}(A,a), B(a), *(A,a), b)) & = & pair_\Sigma(A, B, a, b). \end{array}$$

where $pair_\times$ and $pair_\Sigma$ are the introduction operators for the product types and the Σ-types, respectively. Then, for $a : A$ and $b : B(a)$, we can regard the untyped pair (a,b) as standing for $pair_\times(\mathbf{1}(A,a), B(a), *(A,a), b)$. An implementation based on this understanding will then figure out the appropriate typing for an untyped pair by choosing one of the coercions to apply. Note that according to this analysis, the treatment in the Lego system is not quite adequate, while the implementation of type-checking pairs in the system PVS is more flexible and seems to be very close to our analysis.[6]

Overloading and sense selection

The above mechanism of using unit types for type-casting can be generalised into a genaral mechanism of overloading. The idea is to use several coercions from a unit type to encode the multiple senses of an expression. For example, "finite" can be represented by the object in the unit type, while the images of the coercions are its different senses (eg, "finite$_1$" ranging over sets and "finite$_2$" over sequences). When the expression is used in a context, its appropriate sense is selected, according to the coercive subtyping mechanism. For example, we shall have "finite sequence" = "finite$_2$ sequence" and, for $A : set$, "A is finite" = "A is finite$_1$".

Bailey has further developed this idea in his PhD thesis and applied it to formal development of mathematics [4]. It also has applications, for example, in lexical analysis of natural languages [33].

5 Conclusions, related work, and further research

The central idea of coercive subtyping is to introduce coercive definition rules so that subtyping and inheritance can obtain a uniform proof-theoretic treatment in type theory. This is different from using model-theoretic (denotational) semantics in understanding subtyping. Our approach is more syntactic and gives direct meaning-theoretic treatment of subtyping and coercions (and extends the meaning theory of intensional type theories in a coherent way). It offers the opportunity for subtyping to be introduced into the current proof development systems such as Coq, ALF, and Lego, and to make the task of formal development easier. Although the formal treatment deals with type theories formulated in LF directly, it is not difficult to be

[6] Thanks to Paul Jackson for an interesting conversation on this topic, which has helped to spot an error in an earlier version of this paper.

modified so that it can be applied to other type theories such as those implemented in the Coq system or the NuPRL system. We believe that direct inheritance supported by coercive subtyping is a very useful mechanism that provides a powerful tool in applications such as specification and data refinement (with refinement maps between specifications [29] as coercions), development of mathematical theories in proof development (with theory morphisms [27] as coercions [3]), and library structuring for proof reuse [31].

Subtyping is in general a subtle issue partly because, in the presence of (arbitrary) subtyping, a judgement of the form $k :: K$ is a synthetic judgement in the sense of Martin-Löf [37]; that is, the judgement form is essentially existential and hence in general undecidable, unless the formulation of the system has certain restrictions. Coercive subtyping offers one such restriction. It remains to see how further development can be made in this direction. For instance, we have not considered 'dependent' coercions between a type and a family of types, whose kinds can be of the form $(x{:}A)B(x)$. These coercions may be useful in understanding other forms of implicit syntax. Meta-theoretic study on proof-theoretic properties of coercive subtyping is in progress; some of basic results can be found in [23], including conservativity result of coercive subtyping system $\mathrm{T}[\mathcal{R}]$ wrt the original type theory T. Further results will appear in a forthcoming paper.

The formulation of coercive subtyping in this paper is more general than that given in [32]. In particular, more general basic subtyping rules are allowed to be introduced, including parameterised coercions. We have given a more general and accurate formulation of the coherence conditions for the basic subtyping rules. Note that we allow multiple inheritance in the sense that a type can have more than one subtype or supertype, though it is not allowed to have two different coercions (which are not computationally equal) between two types. For instance, two different mappings from a type of rings to a type of monoids, which map a ring to its different monoids, are not allowed to be coercions at the same time. The only possibility is to regard them as coercions from the ring type to two different types of monoids.

More research is called for to study implementation techniques of coercive subtyping. A particular problem is how to decide whether basic subtyping rules are coherent. In general, this problem is undecidable with possibly infinitely many coercions (eg, introduced by parameterised coercions). Bailey and Saibi have implemented coercions in Lego and Coq, respectively [3, 41]. Both implementations are based on the syntactical equality rather than computational equality in testing whether there is a coercion from one type to another. Although this allows certain forms of parameterised coercions, it is in general undesirable and unsatisfactory. How efficient methods of coherence checking based on computational equality needs to be studied. One of the ways to tackle this problem is to study coherence properties of classes of coercive subtyping relations in different applications so that meta-theoretic results can be obtained and used in helping automated checking. Another approach, complementary to the above, is to consider dynamic checking — rather than checking coherence of all declared coercions, we only check those which have been actually used. But how this can be done efficiently needs to be studied.

Among the closely related work, Pollack and Pierce's suggestion (private communication) to consider coercions as a basic mechanism in type-checking overloading methods for classes proposed by Aczel [1] was a major influence to the idea of coercive subtyping. This idea goes back to the work on giving coercion semantics to lambda calculi with subtyping by Breazu-Tannen et al [8]. Subtyping has also been studied extensively by many researchers in the context of typed functional programming, inspired by the notion of inheritance as found in object-oriented programming (cf., [13, 12]). Semantic studies on subtyping in type systems (without dependent types) such as the second-order lambda calculus (ie, the system F) include the use of the PER model [9] and the coercion-based approach [8]. A more recent logical study of subtyping in system F can be found in [24]. There is not much work on subtyping in dependent type systems in the context of proof development systems based on type theory. Betarte and Tasistro's work on record types and subtyping (or record kinds and subkinding in our terminology) in Martin-Löf's logical framework [43], Pfenning's work on refinement types [39], and Aspinall and Compagnoni's work on the decidability of Edinburgh LF with subtyping [2] are among the recent research development. It is clear that coercive subtyping is strongly motivated by the need in proof development, where inductive types are essential. Traditional approaches based on overloading term constructors (such as λ) cannot be generalised in this context. However, some ideas in the research on subtyping in programming language context may be very useful for proof languages, examples of which include introduction of subtyping assumptions into contexts and bounded quantifiers, among others. Further research is needed to see whether they are useful and how they may be incorporated for coercive subtyping.

Acknowledgements I would like to thank Peter Aczel, Anthony Bailey, Adriana Compagnoni, Healfdene Goguen, Paul Jackson, Alex Jones, Randy Pollack, and Sergei Soloviev, for their useful comments on this work or earlier drafts of this paper. I am very grateful to an anonymous referee who has made many useful comments which have led to further improvements of the paper. This work is partly supported by the UK EPSRC grant on 'Subtyping, Inheritance and Reuse' (GR/K79130).

References

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

[2] D. Aspinall and A. Compagnoni. Subtyping dependent types. *Proc. of LICS96*, 1996.

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

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

[5] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, 1992.

[6] G. Barthes. Implicit coercions in type systems. *Proceedings of Types'95, LNCS 1128*, 1996.

[7] M.J. Beeson. *Foundations of Constructive Mathematics.* Springer-Verlag, 1985.

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

[9] K. Bruce and G. Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87, 1990.

[10] R. Burstall. Extended Calculus of Constructions as a specification language. In R. Bird and C. Morgan, editors, *Mathematics of Program Construction*, 1993. Invited talk.

[11] R. Burstall and J. McKinna. Deliverables: a categorical approach to program development in type theory. LFCS report ECS-LFCS-92-242, Dept of Computer Science, University of Edinburgh, 1992.

[12] L. Cardelli. Type-checking dependent types and subtypes. *Lecture Notes in Computer Science*, 306, 1988.

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

[14] R.L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Pretice-Hall, 1986.

[15] Th. Coquand. Pattern matching with dependent types. Talk given at the BRA workshop on Proofs and Types, Bastad, 1992.

[16] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

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

[18] G. Dowek. The undecidability of typability in the lambda-Pi calculus. *Proc of Typed Lambda Calculi and Applications, LNCS 664*, 1993.

[19] G. Dowek et al. *The Coq Proof Assistent: User's Guide (version 5.6).* INRIA-Rocquencourt and CNRS-ENS Lyon, 1991.

[20] 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.

[21] H. Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh, 1994.

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

[23] A. Jones, Z. Luo, and S. Soloviev. Some proof-theoretic and algorithmic aspects of coercive subtyping. *Proc. of the Annual Conf on Types and Proofs (TYPES'96)*, 1997. To appear.

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

[25] Z. Luo. **ECC**, an extended calculus of constructions. In *Proc. of IEEE Fourth Ann. Symp. on Logic in Computer Science*, Asilomar, California, U.S.A., June 1989.

[26] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990. Also as Report CST-65-90/ECS-LFCS-90-118, Department of Computer Science, University of Edinburgh.

[27] Z. Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90(1), 1991.

[28] 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.

[29] Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3(3), 1993.

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

[31] Z. Luo. Developing reuse technology in proof engineering. *Proceedings of AISB95, Workshop on Automated Reasoning: bridging the gap between theory and practice, Sheffield, U.K.*, April 1995.

[32] 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.

[33] Z. Luo and P. Callaghan. Mathematical vernacular and conceptual well-formedness in mathematical language. *Proc of Logical Aspects of Computational Linguistics 1997.* To appear in LNCS series.

[34] 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.

[35] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proof and Programs, LNCS*, 1994.

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

[37] P. Martin-Löf. Analytic and synthetic judgements in type theory. In P. Parrini, editor, *Kant and Contemporary Epistemology.* Kluwer Accademic Publishers, 1994.

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

[39] F. Pfenning. Refinement types for logical frameworks. *Preliminary Proceedings of BRA Workshop on Types and Proofs*, 1993.

[40] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, Edinburgh University, 1994.

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

[42] Thomas Schreiber. Auxiliary variables and recursive procedures. *TAPSOFT'97: Theory and Practice of Software Development, LNCS 1214*, 1997.

[43] A. Tasistro. *Substitution, record types and subtyping in type theory.* PhD thesis, Chalmers University of Technology, 1997.

[44] S. Yu and Z. Luo. Implementing a model checker for Lego. *Proc. of the 4th Inter Symp. of Formal Methods Europe, FME'97: Industrial Applications and Strengthened Foundations of Formal Methods, Graz, Austria. LNCS 1313*, 1997.