Coercive subtyping in lambda-free logical frameworks

Coercive Subtyping in Lambda-free Logical Frameworks *

Robin Adams
Royal Holloway, University of London
robin@cs.rhul.ac.uk

Abstract

We show how coercive subtyping may be added to a lambda-free logical framework, by constructing the logical framework $\mathrm{TF}_{<}$, an extension of the lambda-free logical framework TF with coercive subtyping. Instead of coercive application, $\mathrm{TF}_{<}$ makes use of a typecasting operation. We develop the metatheory of the resulting framework, including providing some general conditions under which typecasting in an object theory with coercive subtyping is decidable. We show how $\mathrm{TF}_{<}$ may be embedded in the logical framework LF, and hence how results about LF may be deduced from results about $\mathrm{TF}_{<}$.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems, Mechanical theorem proving; I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving—Metatheory; I.2.4 [*Artificial Intelligence*]: Knowledge Representation Formalisms and Methods—Representation languages

General Terms Languages, Theory

Keywords Coercive subtyping, Type theory, Lambda-free logical framework, Typecasting, Metatheory

1. Introduction

When working with dependent type theories, we often find it convenient to introduce a notion of *subtyping*. Intuitively, to say the type A is a *subtype* of the type B is to say that every term of type A is also a term of type B. We often find ourselves in a situation where there are two types, A and B — say $\mathbb{N}$, the type of natural numbers, and $\mathbb{Z}$, the type of integers — such that we would like to work as if A is a subtype of B, but it is not literally true that every term of type A is itself a term of type B.

One approach to this problem is *coercive subtyping*. We construct a function $c : (A)B$ — say, the function that maps each natural number n to the integer $+n$. We declare this function to be a *coercion*, which we write as $A <_c B$, indicating that we intend to identify each term $a : A$ with the term $ca : B$. Whenever a context expects a term of type B, we allow the user to enter a term a of type A instead; the machine is to proceed as if the user had entered ca.

* This work was supported by EPSRC Fellowship number EP/D066638/1.

LFMTP '09 August 2, 2009, Montreal, Canada.

When working in a logical framework, this can be implemented using *coercive application*. If we have declared $A <_c B$, then given any meta-function $f : (x : B)K$ and term $a : A$, we allow the object fa to be typed. We make it computationally equal to $f(ca)$, and we make its kind $[ca/x]K$. This idea has been very important in the literature on the theory of coercive subtyping in dependent type theories.

A number of systems known as *lambda-free logical frameworks* have been developed fairly recently. These are logical frameworks that contain no features other than those that are essential for representing object theories. In particular, there are no product kinds, λ-abstractions, nor framework-level β- or η-reduction. They thus have a relatively small syntax and few rules of deduction, which makes certain technical questions easier to answer than in a 'traditional' framework; that is, a framework with lambda-abstraction.

It would therefore be very desirable to construct lambda-free logical frameworks with coercive subtyping. However, it is not immediately obvious how to do so. We cannot use coercive application, because there are no meta-functions f in a lambda-free framework.

Our solution in this paper is to use *typecasting*. We allow the user to form objects M_B, the object M *typecast* to have type B. If M is of a subtype of B, then M_B will be the result of applying the appropriate coercion to M. There is a close connection between coercive application and *typecasting*, and it is possible to add typecasting to a lambda-free logical framework in a straightforward manner.

Our aim in this paper is to extend the lambda-free logical framework TF [Adams 2009, 2004b] with coercive subtyping, to form the framework we shall call $\mathrm{TF}_{<}$. This system extends TF with *subtyping judgements* of the form $\Gamma \vdash A <_{[x]N} B$, denoting that A is a subtype of B, with an arbitrary object $x : A$ being coerced to the object $N : B$. If this judgement is derivable and $\Gamma \vdash M : A$, then we may form the object M_B, which is M that has been *typecast* to have type B. This object will be computationally equal to $\{M/x\}N$.

1.1 Logical Frameworks

A logical framework is a typing system that is intended to be used as a metalanguage for specifying other formal systems, the *object theories*. These object theories are often typing systems themselves.

Let us first fix some terminology. When the object theory is a typing system, we shall call the entities of the object theory *terms* and *types*, while the entities of the framework we shall call *objects* and *kinds*[1]. However, we shall still talk an object k being 'typable', meaning it has a kind.

[1] This terminology is not universal. In particular, in Martin-Löf's Logical Framework, our 'types' are called 'sets' and our 'kinds' are called 'types'. Unfortunately, there is no universal choice of terminology here yet.

What does it mean to 'specify' or 'represent' an object theory in a logical framework? Typically, there is a special kind $\mathbf{Type}$ and, for each $A : \mathbf{Type}$, a kind $\mathrm{El}\,(A)$. To represent a type theory T in the framework is to declare a number of constants and computation rules in the framework such that we can *map* or *encode* the terms and types of T as objects of the framework. That is, such that

- for each type A of T, we can find an object $\overline{A} : \mathbf{Type}$;
- for each term M of type A, we can find an object $\overline{M} : \mathrm{El}\,(\overline{A})$.

This should be done in such a way that the object $\overline{M}$ behaves in the framework similarly to the way M behaves in T; for example, we should have $\overline{M} = \overline{N}$ if and only if $M = N$. The result that states that such a correspondence holds for T is called the *adequacy theorem* for T.

The correspondence between the entities of T and the objects of the framework is generally not as close as we might wish. It would be ideal if the correspondence were both injective and surjective. However, we typically have neither of these properties:

1. Each entity of T is represented by more than one object of the framework. Typically, if two framework objects are $\beta\eta$-convertible, then they represent the same entity of T.
2. There are typable objects in the framework that do not correspond to entities of T.

Example Suppose T contains non-dependent product types $A \times B$. We may represent these by declaring a constant

$$\mathtt{times} : (\mathbf{Type}, \mathbf{Type})\mathbf{Type} \ .$$

The objects of the form $\mathtt{times}\ a\ b$ then represent the types of the form $A \times B$:

$$\overline{A \times B} \equiv \mathtt{times}\ \overline{A}\ \overline{B} \ .$$

However:

1. this representation is not unique; $A \times B$ is represented by both $\mathtt{times}\ \overline{A}\ \overline{B}$ and $([x : \mathbf{Type}]\mathtt{times}\ x\ \overline{B})\overline{A}$;
2. there are objects such as $\mathtt{times}\ A$, of kind $(\mathbf{Type})\mathbf{Type}$, which does not represent a type or term of T; rather, it represents the *meta-function* that maps a type B to the type $A \times B$.

Adequacy theorems are often notoriously difficult to prove, largely because of disparities such as these between an object theory and its representation in a logical framework.

1.1.1 Lambda-free Logical Frameworks

Lambda-free logical frameworks are a recently constructed family of logical frameworks intended to remedy this problem. A lambda-free logical framework contains nothing but what is essential for representing object theories. In particular, they contain:

- the kinds $\mathbf{Type}$ and $\mathrm{El}\,(A)$;
- the ability to declare *constants* and *variables* with parameters;
- the ability to *apply* constants and variables to arguments.

but they do not contain:

- the ability to apply *abstractions* to arguments, to form β-redexes;
- any framework-level notion of β- or η-reduction;
- *partial application* — that is, the ability to apply an n-ary constant or variable to fewer than n arguments.

Because of this, the correspondence between an object theory and its representation in a lambda-free logical framework is extremely close; each entity in the object theory is typically represented by an object in the framework that is *unique* (up to α-conversion), and each typable object in the framework represents an entity in the object theory.

Several lambda-free logical frameworks have been constructed over the last seven years, including the Canonical LF [Harper and Licata 2007, Lovas and Pfenning 2008], TF and its subsystems [Adams 2009, 2004b], and DMBEL and its subsystems [Plotkin 2006, Pollack 2007]. We describe the history of lambda-free logical frameworks in more detail in Section 5.

1.1.2 Embedding Lambda-free Frameworks in Traditional Frameworks

We do not claim that lambda-free logical frameworks are 'better' than traditional frameworks in every respect. The features that have been removed from lambda-free frameworks, while extraneous in theory, are often useful in practice. For example, β-redexes can provide a useful abbreviational mechanism. However, many theoretical questions are easier to study in a lambda-free logical framework, because a lambda-free framework has a smaller syntax and fewer rules of deduction.

Fortunately, it is possible to enjoy the benefits of both worlds. It is possible to *embed* a lambda-free framework L in a traditional framework F; that is, to define mutually inverse translations between L and F. By means of these translations, L can be regarded as a conservative subsystem of F. For example, TF can be seen as a subsystem of LF[2] [Luo 1994] — it is the fragment of LF that deals only with objects β-normal, η-long form, and LF is conservative over this fragment. Likewise, the Canonical LF can be regarded as the fragment of ELF that deals only with objects in β-normal, η-long form, and ELF is conservative over this fragment.

Once L has been embedded in F in this way, we can *lift* results from L to F; we can prove that a theoretical property holds in L, and deduce that the same property holds in F. The proof is often easier in L than in F. There is a price to be paid for this, of course; the basic metatheoretic properties of L, and the properties of the translations between L and F, are often quite difficult to establish. This can be seen as a 'one-time' cost of using L, however; once these properties have been established, they can be used to lift any number of results from L to F.

There is another advantage of working with lambda-free logical frameworks. They are *modular*, in the following sense.

We shall shortly introduce the lambda-free logical framework TF. A large family of subsystems of TF have been constructed that extend one another conservatively, in a *modular hierarchy* of logical frameworks [Adams 2004a,b]. These differ in the class of object theories they allow to be specified, and the traditional frameworks in which they can be embedded. For example, the subsystem $\mathbf{SPar}\,(\omega)^-$ can be embedded in the Edinburgh Logical Framework, ELF, while TF itself cannot [Adams 2004a, 2009].

It is very often possible to find general results and general techniques that apply to all these subsystems at once. The techniques developed in this paper, for example, could be used with hardly any modification to add coercive subtyping to any of the frameworks in the modular hierarchy. Thus, the techniques in this paper could be used to prove more easily results about coercive subtyping, not just in LF, but also in PAL$^+$, ELF, and any other framework in which some member of the modular hierarchy can be embedded.

1.2 Outline

In Section 2, we shall give a summary of the background on coercive subtyping and lambda-free logical frameworks. In Section

[2] The framework we refer to as LF in this paper is a Church-typed version of Martin-Löf's Logical Framework. It is not to be confused with the Edinburgh Logical Framework, which is also often called LF.

3, we shall present the formal definition of $\mathrm{TF}_<$, and investigate its metatheoretic properties. In particular, we shall give conditions under which a type theory with coercive subtyping is conservative over the same theory without coercive subtyping, and conditions under which typechecking in a theory with coercive subtyping is decidable. In Section 4, we shall show how $\mathrm{TF}_<$ can be embedded in the traditional framework LF, and hence that the results proved in Section 3 also hold for LF.

2. Background

2.1 The Lambda-free Logical Framework TF

We shall now introduce the lambda-free logical framework TF. We shall first discuss the intuitive ideas behind its construction, then present a summary of its syntax and rules of deduction. For more technical details, see [Adams 2009].

As we described in the Introduction, TF is designed so that, when an object theory has been specified, the objects of TF are in one-to-one correspondence with the terms and types of the object theory.

For example, suppose the object theory we wish to represent contains Π-types; whenever A is a type, and B is a type that depends on $x : A$, then $\Pi x : A.B$ is to be a type. We can achieve this in TF — as in many logical frameworks — by declaring the *constant*

$$\Pi : (A : \mathbf{Type})(B : (A)\mathbf{Type})\mathbf{Type} \ . \tag{1}$$

However, the rules that govern how this constant Π behaves in TF are different to the rules of traditional logical frameworks.

Once the constant Π has been declared, the following is an instance of a rule of deduction in TF:

$$\frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash B : \mathbf{Type}}{\Gamma \vdash \Pi[A, [x : A]B] : \mathbf{Type}}$$

Thus, the only way Π can occur is in an object of the form $\Pi[A, [x : A]B]$. In particular, Π cannot appear on its own on the left-hand side of a judgement, and nor can the partial application $\Pi[A]$. Likewise, the *abstraction* $[x : A]B$ cannot occur on its own on the left-hand side of a judgement, but only as the argument to a variable or constant.

Nevertheless, it will be very convenient to be able to talk about abstractions such as $[x : A]B$, and to be able to write the first premise $\Gamma, x : A \vdash B : \mathbf{Type}$ above as

$$\Gamma \Vdash [x : A]B : (x : A)\mathbf{Type} \ . \tag{2}$$

We shall refer to expressions like $[x : A]B$ as *abstractions*, and expressions like $(x : A)\mathbf{Type}$ as *product kinds*. To distinguish them, we shall refer to the kinds $\mathbf{Type}$ and $\mathrm{El}\,(A)$ as *base kinds*.

We shall introduce *defined judgement forms*; when F is an abstraction and K a product kind, we shall write $\Gamma \Vdash [x : A]B : (x : A)\mathbf{Type}$ as an *abbreviation* for a set of judgements of TF. In particular, by our definition, the judgement (2) shall abbreviate the singleton $\{\Gamma, x : A \vdash B : \mathbf{Type}\}$. We shall always use the symbol $\Vdash$ in a defined judgement form.

We turn now to the formal definition of TF.

2.1.1 Syntax of TF

Arities The syntax of TF is organised by *arities* α. Every constant, variable, abstraction and product kind shall be associated with an arity α; we shall speak of α-ary variables, constants, etc. The set of arities is defined by the grammar

$$\alpha ::= (\alpha, \ldots, \alpha) \ .$$

The intuition behind arities is that an $(\alpha_1, \ldots, \alpha_n)$-ary object is a function that takes n arguments — namely an α_1-ary object, …, and an α_n-ary object — and returns a term or type of the object theory.

We write $\mathbf{0}$ for the arity $()$, $\mathbf{1}$ for $(\mathbf{0})$, $\mathbf{2}$ for $(\mathbf{0}, \mathbf{0})$, and so forth. For example, the constant Π declared above has arity $(\mathbf{0}, \mathbf{1})$, as it takes a $\mathbf{0}$-ary object A and a $\mathbf{1}$-ary abstraction $[x : A]B$, and returns an object $\Pi[A, [x : A]B]$.

We can also speak of the *order* of an arity, defined as follows: $\mathbf{0}$ is 0th-order; otherwise the order of $(\alpha_1, \ldots, \alpha_n)$ is the maximum of the orders of α_1, …, α_n plus one. Thus, $\mathbf{1}$ and $\mathbf{2}$ are first-order, $(\mathbf{1}, \mathbf{2})$ is 2nd-order, and so forth. We may speak of nth-order abstractions, product kinds and contexts similarly.

Objects and Kinds of TF The *objects* of TF are defined thus:

> If z is an $(\alpha_1, \ldots, \alpha_n)$-ary constant or variable, then
>
> $$z[[x_{11}, \ldots, x_{1r_1}]M_1, \ldots, [x_{n1}, \ldots, x_{nr_n}]M_n]$$
>
> is an object, where $\alpha_i \equiv (\alpha_{i1}, \ldots, \alpha_{ir_i})$, x_{ij} is an α_{ij}-ary variable, and M_i is an object.

For example, if z is a $\mathbf{0}$-ary constant or variable, then $z[]$ is an object. If z is a $\mathbf{1}$-ary constant or variable, then $z[M]$ is an object for every object M.

We define the α-ary *product kinds* as follows. An $(\alpha_1, \ldots, \alpha_n)$-ary product kind has the form

$$(x_1 : K_1, \ldots, x_n : K_n)T$$

where x_i is an α_i-ary variable, K_i an α_i-ary product kind, and T is a *base kind*; that is, T is either $\mathbf{Type}$ or $\mathrm{El}\,(M)$ for some object M.

An *abstraction* of arity $(\alpha_1, \ldots, \alpha_n)$ is an expression of the form $[x_1 : K_1, \ldots, x_n : K_n]M$, where each x_i is an α_i-ary variable, K_i is an α_i-ary product kind, and M is an object.

An $(\alpha_1, \ldots, \alpha_n)$-ary *context* in TF is a sequence of declarations $x_1 : K_1, \ldots, x_n : K_n$, where x_i is an α_i-ary variable and K_i an α_i-ary product kind of the same arity.

The judgements of TF are of three forms: Γ valid, $\Gamma \vdash M : T$, and $\Gamma \vdash M = N : T$, where Γ is a context, M and N objects, and T a base kind.

Remark

1. We could have written the definition on objects in the following form. An object of TF has the form $z[F_1, \ldots, F_n]$, where z is an $(\alpha_1, \ldots, \alpha_n)$-ary variable or constant, and each F_i is an α_i-ary abstraction.

   Likewise, we could have said: an α-ary abstraction has the form $[\Delta]M$, where Δ is an α-ary context and M an object.

2. All of the expressions we have introduced — objects, base kinds, product kinds, abstractions, contexts and judgements — are identified up to α-conversion.

Instantiation and Defined Judgement Forms We cannot use *substitution* in TF, as we can in most logical frameworks. In general, the result of substituting an abstraction for x in the object $x[\vec{F}]$ is not an object; rather, it would be a β-abstraction, which we have gone to such pains to exclude from TF.

Instead, the operation of *instantiation* (or *hereditary substitution*) plays the role in TF that substitution plays in a traditional framework. We shall define $\{F/x\}M$, the result of *instantiating* the variable x with the abstraction F. This can be thought of as the normal form of $[F/x]M$. It is defined iff F and x have the same arity α, and the definition is by recursion on the arity α.

Definition 1 (Instantiation) *If* $F \equiv [y_1 : K_1, \ldots, y_n : K_n]N$, *then*

$$\begin{aligned} \{F/x\}z[\vec{G}] &\equiv z[\{F/x\}\vec{G}] \qquad (\textit{if } x \not\equiv z) \\ \{F/x\}x[G_1, \ldots, G_n] &\equiv \{G_1/y_1, \ldots, G_n/y_n\}N \end{aligned}$$

We also define an operation of *employment*, which shall play the role in TF that application plays in a traditional framework. If F is an $(\alpha_1, \ldots, \alpha_n)$-ary abstraction, and G_1 is an α_1-ary abstraction, then the abstraction $F \bullet G_1$ is defined by

$$\begin{aligned}&([x_1 : K_1, \ldots, x_n : K_n]M) \bullet G_1\\ &\quad \equiv \quad \{G_1/x_1\}[x_2 : K_2, \ldots, x_n : K_n]M \ .\end{aligned}$$

We introduce *defined judgement forms* to denote an abstraction inhabiting a product kind $\Gamma \Vdash F : K$; or two abstractions being equal in a product kind $\Gamma \Vdash F = G : K$; or two kinds being equal $\Gamma \Vdash K = K'$. Each of these is defined as a set of judgements of the three primitive forms given above.

1. **Equality of Base Kinds** We define

$$\begin{aligned}(\Gamma \Vdash \mathbf{Type} = \mathbf{Type}) \quad &\equiv \quad \{\Gamma \text{ valid}\}\\ (\Gamma \Vdash \mathrm{El}\,(A) = \mathrm{El}\,(B)) \quad &\equiv \quad \{\Gamma \vdash A = B : \mathbf{Type}\}\end{aligned}$$

   $\Gamma \Vdash \mathbf{Type} = \mathrm{El}\,(B)$ and $\Gamma \Vdash \mathrm{El}\,(A) = \mathbf{Type}$ are left undefined.

2. **Equality of Product Kinds and Contexts** $\Gamma \Vdash K = K'$ is defined only if K and K' have the same arity. $\Gamma \Vdash \Delta = \Theta$ is defined only if Δ and Θ have the same arity.

$$\begin{aligned}&(\Gamma \Vdash (\Delta)S = (\Theta)T)\\ &\quad \equiv \quad (\Gamma \Vdash \Delta = \Theta)\\ &\qquad \cup(\Gamma, \Delta \Vdash S = T)\\ &(\Gamma \Vdash (x_1 : J_1, \ldots, x_n : J_n) = (x_1 : K_1, \ldots, x_n : K_n))\\ &\quad \equiv \quad \{\Gamma \text{ valid}\}\\ &\qquad \cup(\Gamma \Vdash J_1 = K_1)\\ &\qquad \cup(\Gamma, x_1 : J_1 \Vdash J_2 = K_2)\\ &\qquad \cup \cdots\\ &\qquad \cup(\Gamma, x_1 : J_1, \ldots, x_{n-1} : J_{n-1} \Vdash J_n = K_n)\end{aligned}$$

3. **Inhabitation of a Product Kind** $\Gamma \Vdash F : K$ is defined only if F and K have the same arity.

$$\begin{aligned}(\Gamma \Vdash [\Delta]M : (\Theta)T) \quad \equiv \quad &(\Gamma \Vdash \Delta = \Theta)\\ &\cup\{\Gamma, \Delta \vdash M : T\}\end{aligned}$$

4. **Equality of Abstractions** $\Gamma \Vdash F = G : K$ is defined only if F, G and K have the same arity.

$$\begin{aligned}&(\Gamma \Vdash [\Delta_1]M = [\Delta_2]N : (\Theta)T)\\ &\quad \equiv \quad (\Gamma \Vdash \Delta_1 = \Theta)\\ &\qquad \cup(\Gamma \Vdash \Delta_2 = \Theta)\\ &\qquad \cup\{\Gamma, \Delta_1 \vdash M = N : T\}\end{aligned}$$

A type theory is specified in TF by declaring a number of *constants* $c : K$ and *computation rules* $(x_1 : K_1, \ldots, x_n : K_N)(M = N : T)$. The rules of deduction of TF are then as given in Fig. 1. We shall write $\Gamma \vdash_T J$ to denote that the judgement $\Gamma \vdash J$ is derivable under the type theory specification T.

2.1.2 Metatheoretic Properties

The metatheoretic properties of a lambda-free framework are very difficult to prove. We shall list in Theorem 1 some properties that we would expect to hold in general. So far, however, these have only been proven for a few limited classes of type theory specifications.

Theorem 1 *If the type theory specification T satisfies the property*

Equation Validity *If $\Gamma \vdash M = N : S$ is derivable, so are $\Gamma \vdash M : S$ and $\Gamma \vdash N : S$.*

$$\frac{}{\langle\rangle \text{ valid}} \qquad \frac{\Gamma \Vdash K \text{ kind}}{\Gamma, x : K \text{ valid}}$$

$$\frac{\Gamma \Vdash \vec{F} :: \Delta}{\Gamma \vdash x\vec{F} : \{\vec{F}/\Delta\}S}\,(x : (\Delta)S \in \Gamma)$$

$$\frac{\Gamma \Vdash \vec{F} = \vec{G} :: \Delta}{\Gamma \vdash x\vec{F} = x\vec{G} : \{\vec{F}/\Delta\}S}\,(x : (\Delta)S \in \Gamma)$$

For each constant declaration $c : (\Delta)S$ in T:

$$\frac{\Gamma \Vdash \vec{F} :: \Delta}{\Gamma \vdash c\vec{F} : \{\vec{F}/\Delta\}S}$$

$$\frac{\Gamma \vdash \vec{F} = \vec{G} :: \Delta}{\Gamma \vdash c\vec{F} = c\vec{G} : \{\vec{F}/\Delta\}S}$$

For each equation declaration $(\Delta)(M = N : S)$ in T:

$$\frac{\Gamma \Vdash \vec{F} :: \Delta}{\Gamma \vdash \{\vec{F}/\Delta\}M = \{\vec{F}/\Delta\}N : \{\vec{F}/\Delta\}S}$$

Figure 1. Rules of Deduction of TF

then it satisfies the following properties.

Instantiation *If $\Gamma, x : K, \Gamma' \vdash J$ and $\Gamma \Vdash F : K$ are derivable, so is $\Gamma, \{F/x\}\Gamma' \vdash \{F/x\}J$.*

Functionality *If $\Gamma, x : K, \Gamma' \vdash M : T$ and $\Gamma \Vdash F = G : K$ are derivable, so is $\Gamma, \{F/x\}\Gamma' \vdash \{F/x\}M = \{G/x\}M : \{F/x\}T$.*

Context Conversion *If $\Gamma, x : K, \Gamma' \vdash J$ and $\Gamma \Vdash K = K'$ are derivable, so is $\Gamma, x : K', \Gamma' \vdash J$.*

Proof See [Adams 2009]. □

2.2 Coercive Subtyping

When working in a type theory with coercive subtyping, we may declare a meta-function $c : (A)B$ to be a *coercion* from A to B, written

$$A <_c B \ .$$

The type A is then to be regarded as a *subtype* of the type B, with each object $a : A$ being identified with, or *coerced* to, the object $ca : B$.

This is achieved in a traditional logical framework such as LF by means of *coercive application* and *coercive definition*. If $A <_c B$, then for any objects $a : A$ and $f : (x : A)K$, the object fa is well-typed, and is set equal to $f(ca)$. This is achieved by extending LF with additional rules of deduction, that include the following two.

$$\text{(CA)}\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash f : (x : A)K \quad \Gamma \vdash A <_c B : \mathbf{Type}}{\Gamma \vdash fa : [ca/x]K}$$

$$\text{(CD)}\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash f : (x : A)K \quad \Gamma \vdash A <_c B : \mathbf{Type}}{\Gamma \vdash fa = f(ca) : [ca/x]K}$$

Provided the subtyping judgements obey certain conditions, the resulting framework obeys some very nice metatheoretic properties. We say that the coercions form a *well-defined set of coercions* (WDC) when these conditions are satisfied (see [Luo and Luo 2005]). The conditions include *coherence*, the requirement that any two coercions between A and B be equal.

This approach has been studied in a series of papers by Luo, Luo, Soloviev and the present author [Luo 1999, Soloviev and

Luo 2002, Luo and Luo 2005, Luo and Adams 2006]. Much more recently, coercive subtyping has been used to allow an intensional construction of manifest fields in dependent record types [Luo 2009].

Coercive subtyping has been implemented in proof checkers such as Coq [Barras and et al. 2000], LEGO [Luo and Pollack 1992] and Plastic [Callaghan and Luo 2001], and several large proof developments have made extensive use of coercions [Bailey 1998, Geuvers et al. 2002].

2.2.1 Coercive Application and Typecasting

If we wish to add coercive subtyping to a logical framework, there is an alternative to coercive application. We can instead introduce the operation of *typecasting*.

By *typecasting*, we mean the operation of taking an object M and 'forcing' or *casting* it to have type A. We write M_A for 'the object M, considered as an object of type A'. If $M : A$, we can 'consider' M to have type A; if $M : A$ and $A <_c B$, we can 'consider' M to have type B. More formally:

- If $M : A$, then $M_A : A$ and $M_A = M : A$.
- If $M : A$ and $A <_c B$, then $M_B : B$ and $M_B = cM : B$.

There is a very close connection between typecasting and coercive application[3]. We can *define* either operation in terms of the other. Given coercive application, we can define M_A to be the identity on A applied to M:

$$M_A \equiv ([x : A]x)M \ .$$

This operation has the properties we require of a typecasting operation. If $M : A$, then $M_A = M$ by a β-reduction; if $M : A$ and $A <_c B$, then $M_B = cM$ by coercive definition.

Conversely, given a typecasting operation, we can define coercive application as follows. If $a : A$, $f : (x : B)K$, and $A <_c B$, then let

$$fa \equiv f(a_B) \ .$$

This is typable and equal to $f(ca)$, as required.

Thus, in a traditional logical framework, we may take either coercive application or typecasting as primitive, then define the other.

2.2.2 Coercive Subtyping in a Lambda-free Logical Framework

It is not immediately obvious how to extend a lambda-free logical framework with coercive subtyping. We cannot introduce coercive application; the rules for coercive application make use of objects fa and $f(ca)$, and these do not exist in a lambda-free framework. However, we can introduce typecasting in a straightforward manner; and, as we have just seen, this should give us the same power as coercive application.

In order to obtain a lambda-free framework with coercive subtyping, therefore, we shall introduce subtyping judgements $A <_c B$, (where c must be an abstraction of product kind $(A)B$), and an operation of *typecasting* with the following properties:

- If $M : \mathrm{El}\,(A)$, then $M_A : \mathrm{El}\,(A)$ and $M_A = M : \mathrm{El}\,(A)$.
- If $M : \mathrm{El}\,(A)$ and $A <_c B$, then $M_B : \mathrm{El}\,(B)$ and $M_B = c \bullet M : \mathrm{El}\,(B)$.

We shall call the resulting framework $\mathrm{TF}_<$.

We shall now give the formal definition of this framework, discuss its metatheoretic properties, and show how it may be embedded in LF with coercive subtyping.

[3] This connection was first observed by Aczel ([Aczel 1994], as reported in [Luo 1996]).

For each basic coercion $(\Delta; A; B; [x]M) \in \mathcal{C}$:

$$(\text{basic})\ \frac{\Gamma \Vdash \vec{F} :: \Delta}{\Gamma \vdash \{\vec{F}/\Delta\}A <_{[x]\{\vec{F}/\Delta\}M} \{\vec{F}/\Delta\}B}$$

$$(\text{cong})\ \frac{\begin{array}{c}\Gamma \vdash A <_{[x]M} B : \mathbf{Type} \\ \Gamma, x : \mathrm{El}\,(A) \vdash M = M' : \mathrm{El}\,(B) \\ \Gamma \vdash A = A' : \mathbf{Type} \\ \Gamma \vdash B = B' : \mathbf{Type}\end{array}}{\Gamma \vdash A' <_{[x]M'} B' : \mathbf{Type}}$$

$$(\text{cast})\ \frac{\Gamma \vdash M : \mathrm{El}\,(A) \quad \Gamma \vdash A <_{[x]N} B}{\Gamma \vdash M_B : \mathrm{El}\,(B)}$$

$$(\text{cast\_eq})\ \frac{\Gamma \vdash M = M' : \mathrm{El}\,(A) \quad \Gamma \vdash B = B' : \mathbf{Type} \quad \Gamma \vdash A <_{[x]N} B}{\Gamma \vdash M_B = M'_{B'} : \mathrm{El}\,(B)}$$

$$(\text{cast\_def})\ \frac{\Gamma \vdash M : \mathrm{El}\,(A) \quad \Gamma \vdash A <_{[x]N} B}{\Gamma \vdash M_B = \{M/x\}N : \mathrm{El}\,(B)}$$

$$(\text{tcast})\ \frac{\Gamma \vdash M : \mathrm{El}\,(A)}{\Gamma \vdash M_A : \mathrm{El}\,(A)} \qquad (\text{tcast\_def})\ \frac{\Gamma \vdash M : \mathrm{El}\,(A)}{\Gamma \vdash M_A = M : \mathrm{El}\,(A)}$$

Figure 2. Rules of Deduction of $\mathrm{TF}_<$

3. The Lambda-free Logical Framework $\mathrm{TF}_<$

We present here the formal definition of the lambda-free logical framework $\mathrm{TF}_<$, an extension of TF that allows for coercive subtyping.

3.1 Subtyping Judgements and Typecasting

The syntax of $\mathrm{TF}_<$ extends the syntax of TF with one new object constructor, and one new judgement form:

- If M and A are objects, then M_A is an object.
- If A, B and M are objects, then $\Gamma \vdash A <_{[x]M} B$ is a judgement.

The object M_A denotes the result of typecasting the object M to have type A. The judgement $\Gamma \vdash A <_{[x]M} B$ denotes that A and B are types, and that there is a coercion from A to B that maps an arbitrary object $N : A$ to the object $\{N/x\}M : B$.

A type theory with coercive subtyping is specified in $\mathrm{TF}_<$ by giving:

- A type theory specification T in TF;
- A set $\mathcal{C}$ of quadruples $(\Delta; A; B; [x]M)$, which we call *basic coercions*.

We write this specification as $T[\mathcal{C}]$.

The rules of deduction of $\mathrm{TF}_<$ are the rules of deduction of TF, together with the rules given in Fig. 2

3.2 Metatheoretic Properties

There are certain properties that the basic coercions must satisfy in order for the theory $T[\mathcal{C}]$ to be useful. These were summarised in the definition of a *well-defined set of coercions* (WDC). The definition was first given for LF in [Luo and Luo 2001] (see also [Luo and Luo 2005]), and it is a straightforward matter to adapt the definition for $\mathrm{TF}_<$.

Definition 2 (Well-Defined Set of Basic Coercions) *The set $\mathcal{C}$ is a* well-defined set of basic coercions *(WDC) with respect to the type theory T iff the following conditions are satisfied:*

1. *Whenever $(\Delta; A; B; c) \in \mathcal{C}$, then $\Delta \vdash A : \mathbf{Type}$, $\Delta \vdash B : \mathbf{Type}$ and $\Delta \Vdash c : (A)B$ in T.*
2. *Whenever $(\Delta; A; B; c) \in \mathcal{C}$, then $\Delta \nvdash A = B : \mathbf{Type}$.*
3. ***Coherence***
   Whenever $(\Delta; A; B; c) \in \mathcal{C}$ and $(\Delta; A'; B'; c') \in \mathcal{C}$, if $\Delta \vdash A = A' : \mathbf{Type}$ and $\Delta \vdash B = B' : \mathbf{Type}$, then $\Delta \Vdash c = c' : (A)B$.
4. *If $(\Delta; A; B; c) \in \mathcal{C}$, $\Delta \subseteq \Delta'$ and Δ' valid in T, then $(\Delta'; A; B; c) \in \mathcal{C}$.*
5. *If $(\Delta, x : K, \Theta; A; B; c) \in \mathcal{C}$ and $\Delta \Vdash F : K$ in T, then $(\Delta, \{F/x\}\Theta; \{F/x\}A; \{F/x\}B; \{F/x\}c) \in \mathcal{C}$.*
6. ***Transitivity***
   If $(\Delta; A; B; c) \in \mathcal{C}$, $(\Delta, B', C, d) \in \mathcal{C}$ and $\Delta \vdash B = B' : \mathbf{Type}$, then $(\Delta; A; C; d \circ c) \in \mathcal{C}$, where $d \circ c \equiv [x : A]d \bullet (c \bullet x)$.

The property of Coherence, in particular, is essential; if there were two unequal coercions from A to B, then the typechecker would not know which object of B to coerce a given object of A to.

The basic metatheoretic properties that $\mathrm{TF}_<$ enjoys are laid out in the following theorem.

Theorem 2 *Suppose $\mathcal{C}$ is a WDC. Then $T[\mathcal{C}]$ satisfies the following properties.*

Weakening *If $\Gamma \vdash \mathcal{J}$, Δ valid and $\Gamma \subseteq \Delta$, then $\Delta \vdash \mathcal{J}$.*
Context Validity *If $\Gamma \vdash \mathcal{J}$ then Γ valid.*

Further, if we assume that $T[\mathcal{C}]$ satisfies

Equation Validity *If $\Gamma \vdash M = N : S$, then $\Gamma \vdash M : S$ and $\Gamma \vdash N : S$.*

then $T[\mathcal{C}]$ satisfies the following four properties:

Cut *If $\Gamma, x : K, \Delta \vdash \mathcal{J}$ and $\Gamma \Vdash F : K$, then $\Gamma, \{F/x\}\Delta \vdash \{F/x\}\mathcal{J}$.*
Functionality *If $\Gamma, x : K, \Delta \vdash M : S$ and $\Gamma \Vdash F = G : K$, then $\Gamma, \{F/x\}\Delta \vdash \{F/x\}M = \{G/x\}M : \{F/x\}S$.*
Context Conversion *If $\Gamma, x : K, \Delta \vdash \mathcal{J}$ and $\Gamma \Vdash K = K'$, then $\Gamma, x : K', \Delta \vdash \mathcal{J}$.*
Type Validity *If $\Gamma \vdash M : A$ or $\Gamma \vdash M = N : A$, then $\Gamma \vdash A : \mathbf{Type}$.*

Proof The proof follows the same pattern as in [Adams 2004b]. We omit the details. □

For the remainder of this paper, we shall assume that $T[\mathcal{C}]$ is a good specification.

Lemma 1 *The following rule of deduction is admissible in $\mathrm{TF}_<$:*

$$\frac{\Gamma \vdash M = M' : A \quad \Gamma \vdash A = A' : \mathbf{Type}}{\Gamma \vdash M_A = M'_{A'} : A}$$

Proof We have $\Gamma \vdash M_A = M : A$ and $\Gamma \vdash M'_{A'} = M' : A'$, and the desired conclusion follows. □

We can also show that the properties of the set of basic coercions in Definition 2 hold for the set of all derivable subtyping judgements:

Theorem 3 *1. If $\Gamma \vdash A <_{[x]M} B$, then $\Gamma \vdash A : \mathbf{Type}$, $\Gamma \vdash B : \mathbf{Type}$ and $\Gamma, x : A \vdash M : B$.*
2. $\Gamma \nvdash A <_{[x]M} A$
3. If $\Gamma \vdash A <_{[x]M} B$ and $\Gamma \vdash A <_{[x]N} B$, then $\Gamma, x : A \vdash M = N : B$.
4. If $\Gamma \vdash A <_{[x]M} B$ and $\Gamma \vdash B <_{[y]N} C$, then $\Gamma \vdash A <_{[x]\{M/y\}N} C$.

Proof

1. This part is proven by a simple induction on derivations.
2. Prove that, if $\Gamma \vdash A <_{[x]M} B$, then $\Gamma \nvdash A = B : \mathbf{Type}$, by induction on the derivation of $\Gamma \vdash A <_{[x]M} B$.
3. Prove that, if $\Gamma \vdash A <_{[x]M} B$, $\Gamma \vdash A' <_{[x]N} B'$, $\Gamma \vdash A = A' : \mathbf{Type}$, and $\Gamma \vdash B = B' : \mathbf{Type}$, then $\Gamma, x : A \vdash M = N : B$, by double induction on the first two premises.
4. Prove that, if $\Gamma \vdash A <_{[x]M} B$, $\Gamma \vdash B' <_{[y]N} C$, and $\Gamma \vdash B = B' : \mathbf{Type}$, then $\Gamma \vdash A <_{[x]\{M/y\}N} C$, by double induction on the first two premises. □

3.3 Insertion of Coercions

In a theory $T[\mathcal{C}]$ with coercive subtyping, every object M is, in a sense, an abbreviation for an object of T. It should, in particular, be the case that every object of $T[\mathcal{C}]$ is computationally equal to an object of T.

We shall now show how, when $\mathcal{C}$ is *computable* (to be defined shortly), we can give an algorithm for *inserting coercions*; that is, given an object M of $T[\mathcal{C}]$, to compute the corresponding object $\overline{M}$ of T. We shall use this to show how to extend any typechecking algorithm for T to a typechecking algorithm for $T[\mathcal{C}]$.

Definition 3 (Computable Set of Basic Coercions) *A set of basic coercions $\mathcal{C}$ is* computable *iff there exists an algorithm $\mathcal{A}$ such that, given Γ, A and B, the algorithm $\mathcal{A}$ decides whether there exists c such that $(\Gamma, A, B, c) \in \mathcal{C}$ and, if so, returns such a c.*

In order to define the algorithm for insertion of coercions, we shall need the notion of the *principal kind* $K_\Gamma(M)$ of an object M. This is a base kind such that, if M is typable, then it has kind $K_\Gamma(M)$.

Definition 4 (Principal Kind) *The base kind $K_\Gamma(M)$ is defined as follows.*

If $x : (\Delta)S \in \Gamma$, then $K_\Gamma(x\vec{F}) \equiv \{\vec{F}/\Delta\}S$.

If c has been declared with kind $(\Delta)S$, then $K_\Gamma(c\vec{F}) \equiv \{\vec{F}/\Delta\}S$.

$K_\Gamma(M_A) \equiv \mathrm{El}\,(A)$.

We can easily prove that K_Γ has the property we require:

Lemma 2 *If $\Gamma \vdash M : S$, then $\Gamma \vdash S = K_\Gamma(M)$.*

Proof A simple induction on derivations. □

Now, the algorithm for inserting coercions can be provided.

Definition 5 (Insertion of Coercions) *Let T be a type theory specification, and suppose that typechecking in T is decidable. Let$\mathcal{C}$ be a computable WDC with algorithm $\mathcal{A}$. Given a $T[\mathcal{C}]$-object M and a T-context Γ, we define the T-object $\overline{M}^\Gamma$ as follows.*

$$\begin{array}{rcl}
\overline{x\vec{F}}^\Gamma & \equiv & x\overline{\vec{F}}^\Gamma \\
\overline{c\vec{F}}^\Gamma & \equiv & c\overline{\vec{F}}^\Gamma \\
\overline{M_A}^\Gamma & \equiv & \begin{cases}
\overline{M}^\Gamma & \text{if } K_\Gamma(\overline{M}^\Gamma) \equiv \mathrm{El}\,(B) \\
 & \text{and } \Gamma \Vdash_T \overline{A}^\Gamma = B : \mathbf{Type} \\
c\overline{M}^\Gamma & \text{if } K_\Gamma(\overline{M}^\Gamma) \equiv \mathrm{El}\,(B) \\
 & \text{and } \mathcal{A}(\Gamma, B, \overline{A}^\Gamma) \equiv c \\
\textit{undefined} & \textit{otherwise}
\end{cases}
\end{array}$$

(By coherence, the two conditions in the last equation are exclusive.)

We extend the definition to contexts, product kinds, abstractions and judgements in the obvious manner.

Note that, if M does not involve typecasting, then $\overline{M}^{\Gamma} \equiv M$.

This mapping satisfies the property we require:

Theorem 4 *If* $\Gamma \vdash_{T[\mathcal{C}]} M : S$, *then* $\overline{\Gamma} \vdash_T \overline{M}^{\overline{\Gamma}} : \overline{S}^{\overline{\Gamma}}$ *and* $\Vdash_{T[\mathcal{C}]} \Gamma = \overline{\Gamma}$, $\Gamma \Vdash_{T[\mathcal{C}]} S = \overline{S}^{\overline{\Gamma}}$, $\Gamma \vdash_{T[\mathcal{C}]} M = \overline{M}^{\overline{\Gamma}} : S$.

Proof We must first show the following result.

If $\overline{M}^{\Gamma,x:K,\Delta}$ is defined and $\Gamma \vdash_T F : K$, then $\overline{\{F/x\}M}^{\Gamma,\{F/x\}\Delta}$ is defined, and

$$\Gamma, \{F/x\}\Delta \vdash \{F/x\}\overline{M}^{\Gamma,x:K,\Delta} = \overline{\{F/x\}M}^{\Gamma,\{F/x\}\Delta} \quad .$$

The proof is by induction on M, making use of coherence in the case $M \equiv N_A$. The theorem is now proven by induction on derivations. □

Corollary 5 *Suppose* $\Gamma \vdash M : S$ *and* $\Gamma \vdash N : S$ *in* $T[\mathcal{C}]$. *Then* $\Gamma \vdash_{T[\mathcal{C}]} M = N : S$ *if and only if* $\overline{\Gamma} \vdash_T \overline{M} = \overline{N} : \overline{S}$.

Corollary 6 *If typechecking in* T *is decidable, and* $\mathcal{C}$ *is a computable WDC, then typechecking in* $T[\mathcal{C}]$ *is decidable.*

Proof To decide whether $\Gamma \vdash x\vec{F} : S$ is derivable in $T[\mathcal{C}]$, first decide whether Γ valid and $\Gamma \Vdash S$ kind. If either of these does not hold, answer 'no'. If $x \notin \operatorname{dom}\Gamma$, answer 'no'.

Otherwise, let $x : (\Delta)R \in \Gamma$. Then $\Gamma \vdash x\vec{F} : S$ if and only if $\Gamma \Vdash S = \{\vec{F}/\Delta\}R$, if and only if $\overline{\Gamma} \vdash_T \overline{S} = \overline{\{\vec{F}/\Delta\}R}$. Decide whether this last condition holds using the typechecking algorithm for T.

The other judgement forms are handled similarly. □

Another consequence of this theorem is that $T[\mathcal{C}]$ is a conservative extension on T:

Corollary 7 *If* J *is a judgement of* T *derivable in* $T[\mathcal{C}]$, *then* J *is derivable in* T.

3.3.1 Non-computable Sets of Coercions

In fact, it holds even for non-computable $\mathcal{C}$ that, given any typable object M of $T[\mathcal{C}]$, we can find the corresponding object $\overline{M}$ of T. However, the algorithm for calculating $\overline{M}$ implicit in this proof depends on the *derivation* of $\Gamma \vdash M : S$, not just on the object M and context Γ.

Theorem 8 (Insertion of Coercions) *If* $\Gamma \vdash_{T[\mathcal{C}]} M : S$, *then there exist* $\overline{\Gamma}$, $\overline{M}$ *and* $\overline{S}$ *such that* $\overline{\Gamma} \vdash_T \overline{M} : \overline{S}$ *and*

$$\Vdash_{T[\mathcal{C}]} \Gamma = \overline{\Gamma} \qquad \Gamma \Vdash_{T[\mathcal{C}]} S = \overline{S} \qquad \Gamma \vdash_{T[\mathcal{C}]} M = \overline{M} : S$$

Further, if M *does not involve typecasting, then* $\overline{M} \equiv M$*; similarly for* Γ *and* S.

Proof We prove the following three statements simultaneously by induction on derivations:

1. Suppose $\Gamma \vdash_{T[\mathcal{C}]} M : \mathrm{El}\,(A)$, and $\overline{\Gamma} \vdash_T \overline{A} : \mathbf{Type}$, $\Vdash_{T[\mathcal{C}]} \Gamma = \overline{\Gamma}$, and $\Gamma \Vdash_{T[\mathcal{C}]} A = \overline{A} : \mathbf{Type}$. Then there exists $\overline{M}$ such that $\overline{\Gamma} \vdash_T \overline{M} : \mathrm{El}\,(\overline{A})$ and $\Gamma \vdash_{T[\mathcal{C}]} M = \overline{M} : \mathrm{El}\,(A)$. If M does not involve typecasting, then $\overline{M} \equiv M$.
2. Suppose $\Gamma \vdash_{T[\mathcal{C}]} A : \mathbf{Type}$ and $\overline{\Gamma} \vdash_T$ valid, $\Vdash_{T[\mathcal{C}]} \Gamma = \overline{\Gamma}$. Then there exists $\overline{A}$ such that $\overline{\Gamma} \Vdash_T \overline{A}$ kind and $\Gamma \Vdash_{T[\mathcal{C}]} A = \overline{A} : \mathbf{Type}$. If A does not involve typecasting, then $\overline{A} \equiv A$.
3. Suppose $\Gamma \vdash_{T[\mathcal{C}]} A <_{[x]M} B$ and $\overline{\Gamma} \vdash_T$ valid, $\Vdash_{T[\mathcal{C}]} \Gamma = \overline{\Gamma}$. Then there exist $\overline{A}$, $\overline{B}$ and $\overline{M}$ such that $\Gamma, x : \overline{A} \vdash_T \overline{M} : \overline{B}$, $\Gamma \vdash_{T[\mathcal{C}]} A = \overline{A} : \mathbf{Type}$, $\Gamma \vdash_{T[\mathcal{C}]} B = \overline{B} : \mathbf{Type}$, and $\Gamma, x : A \vdash_{T[\mathcal{C}]} M = \overline{M} : B$. □

It follows immediately from this theorem that $T[\mathcal{C}]$ is a conservative extension of T, as in Corollary 7.

4. Embedding $\mathrm{TF}_<$ in LF

We now consider the traditional logical framework LF. It is possible to *embed* TF in LF — that is, to define mutually inverse translations between TF and LF. Using these translations, we can "lift" many results from the TF world to the LF world. That is, we can prove properties of the TF systems then, using the properties of the translations that we have established, deduce that the corresponding properties hold of the LF systems.

This is often a profitable approach as many properties are simpler to prove for the TF systems, since they have fewer constructors in their grammar and fewer rules of deduction. It was employed to prove the injectivity of type constructors in LF in [Luo and Adams 2006].

In this section, we shall show how $\mathrm{TF}_<$ may be embedded in LF with coercive subtyping, and show how the properties of $\mathrm{TF}_<$ proven in the previous section can be lifted to the LF systems.

4.1 The Logical Framework LF

The logical framework LF was introduced by Luo in [Luo 1994]. It is a Church-typed version of Martin-Löf's logical framework [Nordström et al. 1990], and is intended for use as a meta-language for specifying type theories. In particular, it contains the ability to declare computation rules in a type theory. It has been implemented in the proof checker Plastic [Callaghan and Luo 2001].

Its grammar deals with *objects* and *kinds*. The kinds K in LF are of the following forms: $\mathbf{Type}$, whose objects are the types of the type theory being specified; $\mathrm{El}\,(A)$, whose objects are the terms of type A; and $(x : K)K'$, whose objects are the meta-functions which, given an object k of kind K, return an object of kind $[k/x]K'$.

An object k in LF may be a variable, a constant, a lambda-abstraction $[x : K]k$, or an application $k_1 k_2$.

An object theory is specified in LF by declaring a number of *constants* and *computation rules*. We may declare a constant c of kind K, or a computation rule of the form '$k = k' : K$ where $x_1 : K_1, \ldots, x_n : K_n$'.

4.1.1 Embedding TF in LF

We have two type systems, TF and LF, intended for use as logical frameworks. LF can be seen as a conservative extension of TF. We can see TF as picking out the derivable judgements of LF in which everything is in normal form.

To make this precise, we introduce *translations* NF from LF to TF, and lift from TF to LF. The action of NF can be thought of as reducing every object to its normal form. The definitions are given in [Adams 2009], as well as the proof of the following properties:

Theorem 9 *Let* T *be a type theory specified in LF. Let* $\mathrm{NF}(T)$ *be the TF specification defined as follows: for every constant declaration* $c : K$ *in* T, *we declare* $c : \mathrm{NF}_{\langle\rangle}(K)$ *in* $\mathrm{NF}(T)$*; and for every computation rule declaration '*$k = k' : K$ *where* $x_1 : K_1, \ldots, x_n : K_n$*' in* T, *we declare* $(\mathrm{NF}(x_1 : K_1, \ldots, x_n : K_n))(\mathrm{NF}_{\vec{x}::\vec{K}}(k) = \mathrm{NF}_{\vec{x}::\vec{K}}(k') : \mathrm{NF}_{\vec{x}::\vec{K}}(K))$ *in* $\mathrm{NF}(T)$. *Then*

1. *If* J *is derivable in* T, *then* $\mathrm{NF}(J)$ *is derivable in* $\mathrm{NF}(T)$.
2. *For* J *an* $\mathrm{NF}(T)$ *judgement,* J *is derivable in* $\mathrm{NF}(T)$ *if and only if* $\mathrm{lift}(J)$ *is derivable in* T.
3. *(a)* $\mathrm{NF}_\Gamma(\mathrm{lift}(M)) \equiv M$
   (b) $\mathrm{NF}_\Gamma(\mathrm{lift}(K)) \equiv K$
4. *(a) If* $\Gamma \vdash k : K$ *is derivable in* T, *then so is* $\Gamma \vdash k = \mathrm{lift}(\mathrm{NF}_\Gamma(k)) : K$.
   (b) If $\Gamma \vdash K$ kind *is derivable in* T, *then so is* $\Gamma \vdash K = \mathrm{lift}(\mathrm{NF}_\Gamma(K))$.

It is thanks in particular to part 2 of this theorem that we can view LF as being a conservative extension of TF.

4.2 Translations between $\mathrm{TF}_<$ and LF

We now show how these translations NF and lift may be extended to $\mathrm{TF}_<$. We do so by making use of the ideas discussed in Section 2.2.1. The translation lift shall map M_A to the identity applied to $\mathrm{lift}(M)$:

$$\mathrm{lift}(M_A) \equiv ([x : \mathrm{lift}(A)]x)\mathrm{lift}(M)$$

It is harder to define the translation in the other direction. We shall give the name $\overline{\mathrm{NF}}$ to the translation from LF to $\mathrm{TF}_<$. Intuitively, when calculating $\overline{\mathrm{NF}}(fa)$, we should:

- infer the kind of $\overline{\mathrm{NF}}(f)$ — say $(x : K)K'$;
- *cast* $\overline{\mathrm{NF}}(a)$ to have the kind K;
- return $\overline{\mathrm{NF}}(f) \bullet \overline{\mathrm{NF}}(a)_K$.

The second step requires something stronger than typecasting; it requires us to cast *abstractions* to have a particular *product kind*. We shall call this operation *kindcasting*. We shall also refer to the operation in the last step as *coercive employment*. We begin by defining kindcasting and coercive employment formally.

4.2.1 Kindcasting

We extend typecasting to the kinds of higher arity. Given an abstraction F of kind K, we shall define the object $F_{K'}$, the result of casting F to have kind K'.

The formal definition is as follows. For each variable x and kind K of the same arity, let x^K be the η-long form of x considered as an abstraction of kind K. Given an abstraction F and kind K, both of the same arity α, we define the α-ary abstraction F_K by recursion on α thus:

$$M_{\mathrm{El}(A)} \equiv M_A \qquad M_{\mathbf{Type}} \equiv M$$
$$([x : K]G)_{(y:K')L'} \equiv [y : K'](\{(y^{K'})_K/x\}G)_{L'}$$

4.2.2 Coercive Employment

Suppose we wish to employ an abstraction F, which expects arguments of kind K, on an abstraction G, which may be of kind K or a subkind of K. We give the name of *coercive employment* to the following operation: first *cast* G to be of kind K, then employ F on the result. (It is for this reason that we included trivial typecasting: without it, we would need some method of deciding whether or not to use kindcasting on G.)

Formally, we define the abstraction $F \diamond G$ by:

$$([x : K]F') \diamond G \equiv \{G_K/x\}F' \ .$$

4.2.3 The Translation $\overline{\mathrm{NF}}$

We now wish to define a translation from LF to $\mathrm{TF}_<$. This translation cannot agree with NF on every object; for, if $k : A$ and $A < B$, then k and $([x : B]x)k$ have different typing properties in LF, yet $\mathrm{NF}(([x : B]x)k) \equiv \mathrm{NF}(k)$. For this reason, we use a new symbol $\overline{\mathrm{NF}}$ for the new translation. (We shall define it so that $\overline{\mathrm{NF}}(([x : B]x)k)$ is the result of typecasting $\overline{\mathrm{NF}}(k)$ to have type $\overline{\mathrm{NF}}(B)$.)

For the rest of this paper, assume we have declared a type theory $T[\mathcal{C}]$ in LF with coercive subtyping. Declare the theory $\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]$ in $\mathrm{TF}_<$, where $\mathrm{NF}(T)$ is as in Theorem 9, and $\mathcal{C}$ is the set of all basic coercions

$$(\mathrm{NF}(\Gamma); \mathrm{NF}(A); \mathrm{NF}(B); \mathrm{NF}(c))$$

such that $(\Gamma; A; B; c) \in \mathcal{C}$. We assume that $\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]$ is a WDC.

We now define $\overline{\mathrm{NF}}_\Gamma(k)$, $\overline{\mathrm{NF}}_\Gamma(K)$, $\overline{\mathrm{NF}}(\Gamma)$ and $\overline{\mathrm{NF}}(J)$ for every well-aritied object k, kind K, context Γ and judgement J. The definition is as follows.

- If c has been declared with kind K, then $\overline{\mathrm{NF}}_\Gamma(c) \equiv c^K$.
- If $x : K$ is in Γ, then $\overline{\mathrm{NF}}_\Gamma(x) \equiv x^K$.
- $\overline{\mathrm{NF}}_\Gamma([x : K]k) \equiv [x : \overline{\mathrm{NF}}_\Gamma(K)]\overline{\mathrm{NF}}_{\Gamma,x:K}(k)$.
- $\overline{\mathrm{NF}}_\Gamma(kk') \equiv \overline{\mathrm{NF}}_\Gamma(k) \diamond \overline{\mathrm{NF}}_\Gamma(k')$.
- $\overline{\mathrm{NF}}_\Gamma(\mathbf{Type}) \equiv \mathbf{Type}$.
- $\overline{\mathrm{NF}}_\Gamma(\mathrm{El}\,(k)) \equiv \mathrm{El}\,\big(\overline{\mathrm{NF}}_\Gamma(k)\big)$.
- $\overline{\mathrm{NF}}_\Gamma((x : K)L) \equiv (x : \overline{\mathrm{NF}}_\Gamma(K))\overline{\mathrm{NF}}_{\Gamma,x:K}(L)$.

We can define $\overline{\mathrm{NF}}(\Gamma)$ and $\overline{\mathrm{NF}}(J)$ for Γ a context and J a judgement of LF in the obvious way.

4.2.4 The Translation lift

The translation lift from $\mathrm{TF}_<$ to LF is much easier to define. We map typecasting to coercive application of identity functions. The definition of lift on objects is as follows:

$$\begin{aligned} \mathrm{lift}(x\vec{F}) &\equiv x\,\mathrm{lift}(\vec{F}) \\ \mathrm{lift}(c\vec{F}) &\equiv c\,\mathrm{lift}(\vec{F}) \\ \mathrm{lift}(M_A) &\equiv ([x : \mathrm{El}\,(\mathrm{lift}(A))]x)\,\mathrm{lift}(M) \end{aligned}$$

We extend this mapping to abstractions, contexts, etc. in the obvious manner.

We can show that these translations are sound and mutually inverse, up to equality in the relevant framework:

Theorem 10 *1. If J is derivable in LF, then $\overline{\mathrm{NF}}J$ is derivable in $\mathrm{TF}_<$.*
2. If J is derivable in $\mathrm{TF}_<$, then $\mathrm{lift}(J)$ is derivable in LF.
3. If $\Gamma \vdash k : K$ is derivable in LF, then so is $\Gamma \vdash k = \mathrm{lift}(\overline{\mathrm{NF}}_\Gamma(k)) : K$.
4. If $\Gamma \vdash M : T$ is derivable in $\mathrm{TF}_<$, then so is $\Gamma \vdash M = \overline{\mathrm{NF}}_{\mathrm{lift}(\Gamma)}(\mathrm{lift}(M)) : T$.

Proof Parts 1 and 2 are proven by induction on derivations, after establishing several lemmas about how substitution and instantiation behave under $\overline{\mathrm{NF}}$ and lift. Part 3 is proven by induction on k, simultaneously with a corresponding statement for kinds. Part 4 is proven by induction on M. We omit the details here. □

We have shown that NF and $\overline{\mathrm{NF}}$ do not agree on all the objects typable in LF; however, using the previous theorem, we can prove immediately that NF and $\overline{\mathrm{NF}}$ agree (up to judgemental equality in $\mathrm{TF}_<$) on the objects *typable in* T.

Theorem 11 *If $\Gamma \vdash_T k : K$, then $\overline{\mathrm{NF}}(\Gamma) \Vdash \overline{\mathrm{NF}}_\Gamma(k) = \mathrm{NF}_\Gamma(k) : \overline{\mathrm{NF}}_\Gamma(K)$ is derivable in $\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]$.*

Proof Suppose $\Gamma \vdash_T k : K$. Then $\Gamma \vdash_T k = \mathrm{lift}(\mathrm{NF}_\Gamma(k)) = \mathrm{lift}(\overline{\mathrm{NF}}_\Gamma(k)) : K$. Therefore, using the soundness of NF,

$$\mathrm{NF}(\Gamma) \Vdash \mathrm{NF}_\Gamma(\mathrm{lift}(\mathrm{NF}_\Gamma(k))) = \mathrm{NF}_\Gamma(\mathrm{lift}(\overline{\mathrm{NF}}_\Gamma(k))) : \mathrm{NF}_\Gamma(K)$$

is derivable in $\mathrm{NF}(T)$, hence in $\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]$. Therefore, as NF is a left inverse to lift up to equality (which can be proven in $\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]$ just as it was in $\mathrm{NF}(T)$), we have

$$\overline{\mathrm{NF}}(\Gamma) \Vdash_{\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]} \mathrm{NF}_\Gamma(k) = \overline{\mathrm{NF}}_\Gamma(k) : \overline{\mathrm{NF}}_\Gamma(K) \ . \quad \square$$

4.3 Lifting Results

Using the translations $\overline{\mathrm{NF}}$ and lift, we can deduce that the property we have called Insertion of Coercions holds in LF as well as in TF:

Corollary 12 *If $\Gamma \vdash_{T[\mathcal{C}]} k : K$, then there exist $\overline{\Gamma}$, $\overline{k}$ and $\overline{K}$ such that $\overline{\Gamma} \vdash_T \overline{k} : \overline{K}$, $\Vdash_{T[\mathcal{C}]} \Gamma = \overline{\Gamma}$, $\Gamma \vdash_{T[\mathcal{C}]} K = \overline{K}$, and $\Gamma \vdash_{T[\mathcal{C}]} k = \overline{k} : K$.*

Proof Suppose $\Gamma \vdash_{T[\mathcal{C}]} k : K$. Then, by the soundness of $\overline{\mathrm{NF}}$,

$$\overline{\mathrm{NF}}(\Gamma) \Vdash_{\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]} \overline{\mathrm{NF}}_\Gamma(k) : \overline{\mathrm{NF}}_\Gamma(K) \ .$$

Therefore, by Theorem 8, there exist Δ, F and L such that

$$\begin{array}{ccc} \Delta & \Vdash_{\mathrm{NF}(T)} & F : L \\ & \Vdash_{\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]} & \overline{\mathrm{NF}}(\Gamma) = \Delta \\ \overline{\mathrm{NF}}(\Gamma) & \Vdash_{\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]} & \overline{\mathrm{NF}}_\Gamma(K) = L \\ \overline{\mathrm{NF}}(\Gamma) & \Vdash_{\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]} & \overline{\mathrm{NF}}_\Gamma(k) = F : \overline{\mathrm{NF}}_\Gamma(K) \end{array}$$

Let $\overline{\Gamma} \equiv \mathrm{lift}(\Delta)$, $\overline{k} \equiv \mathrm{lift}(F)$, and $\overline{K} \equiv \mathrm{lift}(L)$. Then, by the soundness of lift, we have

$$\begin{array}{ccc} \overline{\Gamma} & \vdash_T & \overline{k} : \overline{K} \\ & \Vdash_{T[\mathcal{C}]} & \mathrm{lift}(\overline{\mathrm{NF}}(\Gamma)) = \overline{\Gamma} \\ \mathrm{lift}(\overline{\mathrm{NF}}(\Gamma)) & \vdash_{T[\mathcal{C}]} & \mathrm{lift}(\overline{\mathrm{NF}}_\Gamma(K)) = \overline{K} \\ \mathrm{lift}(\overline{\mathrm{NF}}(\Gamma)) & \vdash_{T[\mathcal{C}]} & \mathrm{lift}(\overline{\mathrm{NF}}_\Gamma(k)) = \overline{k} : \mathrm{lift}(\overline{\mathrm{NF}}_\Gamma(K)) \end{array}$$

The desired results follow using the fact that lift is a left inverse to $\overline{\mathrm{NF}}$. □

The pattern of this proof is typical of the method for lifting results from TF to LF.

This result was first proved for LF by Soloviev and Luo (it is an easy consequence of the main theorem in [Soloviev and Luo 2002]). The proof is long and difficult, and this result is a good example of how the use of lambda-free logical frameworks leads to results being easier to prove. (If we tried to prove Theorem 8 for LF by the same technique, the induction would fail for the cases of the rules governing application and coercive application.)

Corollary 7 cannot be lifted directly to the LF systems; in general, $T[\mathcal{C}]$ is not literally a conservative extension of T. (If $\Gamma \vdash_{T[\mathcal{C}]} A < B : \mathbf{Type}$, then $\Gamma, y : \mathrm{El}\,(A) \vdash_{T[\mathcal{C}]} ([x : \mathrm{El}\,(B)]x)y : \mathrm{El}\,(B)$. This judgement can be formed in the syntax of T, but is not derivable in T.)

However, a weaker version of this result can be lifted: the equational theory of the objects typable in T is the same in T and $T[\mathcal{C}]$.

Corollary 13 *If* $\Gamma \vdash_T k : K$, $\Gamma \vdash_T k' : K$, *and* $\Gamma \vdash_{T[\mathcal{C}]} k = k' : K$, *then* $\Gamma \vdash_T k = k' : K$.

Proof Suppose the hypotheses hold. By the soundness of $\overline{\mathrm{NF}}$,

$$\overline{\mathrm{NF}}(\Gamma) \vdash_{\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]} \overline{\mathrm{NF}}_\Gamma(k) = \overline{\mathrm{NF}}_\Gamma(k') : \overline{\mathrm{NF}}_\Gamma(K) \ .$$

By the soundness of NF, we have that $\mathrm{NF}_\Gamma(k)$ and $\mathrm{NF}_\Gamma(k')$ are typable in $\mathrm{NF}(T)$, hence do not involve typecasting. Therefore, by Corollary 7,

$$\mathrm{NF}(\Gamma) \vdash_{\mathrm{NF}(T)} \mathrm{NF}_\Gamma(k) = \mathrm{NF}_\Gamma(k') : \mathrm{NF}_\Gamma(K)$$

$$\therefore \Gamma \vdash_T k = \mathrm{lift}(\mathrm{NF}_\Gamma(k)) = \mathrm{lift}(\mathrm{NF}_\Gamma(k')) = k' : K \ . \quad \square$$

The result about typechecking, Corollary 6, can be lifted.

Corollary 14 *If typechecking in* T *is decidable, and* NF$\mathcal{C}$ *is a computable WDC, then typechecking in* $T[\mathcal{C}]$ *is decidable.*

Proof By Corollary 6, the $\mathrm{TF}_<$-theory $\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]$ is decidable. It follows that *equality* is decidable in $T[\mathcal{C}]$; that is, if $\Gamma \vdash k : K$ and $\Gamma \vdash k' : K$, it is decidable whether $\Gamma \vdash k = k' : K$. This is so because $\Gamma \vdash_{T[\mathcal{C}]} k = k' : K$ if and only if $\overline{\mathrm{NF}}(\Gamma) \vdash_{\mathrm{NF}(T)[\mathrm{NF}(\mathcal{C})]} \overline{\mathrm{NF}}_\Gamma(k) = \overline{\mathrm{NF}}_\Gamma(k') : \overline{\mathrm{NF}}_\Gamma(K)$. Once we have a decision procedure for equality, it is straightforward to produce a decision procedure for the whole of $T[\mathcal{C}]$. □

5. Related Work

5.1 History

The concept of a lambda-free logical framework has been invented three times independently over the last 13 years. The first lambda-free framework to appear was the Linear Logical Framework (LLF) [Cervesato 1996, Cervesato and Pfenning 2002], a framework for representing linear object theories. This framework only allows objects in *canonical form*; that is, β-normal, η-long form.

In 2000, Luo [Luo 2003] presented the logical framework PAL$^+$, and it was for this framework that the phrase 'lambda-free logical framework' was coined. However, it is important to note that PAL$^+$ is stronger than the other lambda-free frameworks we consider in this paper. Not all the comments we have made about lambda-free logical frameworks apply to PAL$^+$. PAL$^+$ does not allow partial application, but it *does* have a mechanism for forming abstractions, and makes use of framework-level reduction. It is *not* true that PAL$^+$ only makes use of canonical forms.

Aczel independently developed the framework TF [Aczel 2001], and the present author developed an infinite number of subsystems of TF, the *modular hierarchy* of logical frameworks [Adams 2004a,b].

LLF was extended to the Concurrent Logical Framework (CLF) [Watkins et al. 2003], which again deals only with canonical forms. A subsystem of both LLF and CLF was developed called the Canonical LF [Harper and Licata 2007, Lovas and Pfenning 2008]; this is also a subsystem of ELF. It can be seen as the fragment of ELF that deals only with objects in canonical form.

Independently, a family of four related systems were developed by Plotkin [Plotkin 2006, Pollack 2007]: BEL (Binding Equational Logic), MBEL (Multi-Sorted Binding Equational Logic), DBEl (Dependent Binding Equational Logic) and DMBEL (Dependent Multi-Binding Equational Logic). Again, these systems only deal with objects in β-normal, η-long form.

5.2 The Modular Hierarchy of Logical Frameworks

The *modular hierarchy of logical frameworks* [Adams 2004a,b] is an infinite family of subsystems of TF, with the names

$$\mathbf{SPar}\,(n)\,, \mathbf{LPar}\,(n)\,, \mathbf{SPar}\,(n)^-\,, \mathbf{LPar}\,(n)^-$$

where n may be a natural number or ω. The systems vary in strength in the following manner:

- The systems $\mathbf{SPar}$ allow only *small parameters* — that is, if a constant or variable has kind $(\Delta)T$, then the symbol $\mathbf{Type}$ may not occur in Δ. The systems $\mathbf{LPar}$ do not have this restriction; they allow *large parameters*.
- If n is a natural number, it is the largest order of constant that may be declared (e.g. only constants of order ≤ 2 may be declared in $\mathbf{SPar}\,(2)$.) If $n = \omega$, then constants of any order may be declared.
- The systems $\mathbf{SPar}\,(n)$ and $\mathbf{LPar}\,(n)$ allow computation rules to be declared; the systems $\mathbf{SPar}\,(n)^-$ and $\mathbf{LPar}\,(n)^-$ do not.

These systems are all constructed in a uniform manner, and may be embedded in different traditional frameworks. For example, $\mathbf{SPar}\,(\omega)^-$ may be embedded in the Edinburgh Logical Framework.

The Canonical LF and the four systems developed by Plotkin are very close to five of the systems in the modular hierarchy. In particular, the Canonical LF is isomorphic to $\mathbf{SPar}\,(\omega)^-$. The system DMBEL can easily be added to the hierarchy; it sits between $\mathbf{SPar}\,(2)^-$ and $\mathbf{SPar}\,(3)^-$. These correspondences are investigated in more detail in [Adams 2009].

The techniques developed in this paper apply equally well to all the other systems in the hierarchy. They could be used to add coercive subtyping to any of the systems in the hierarchy, and therefore could easily be adapted to add coercive subtyping to the Canonical LF or DMBEL. They could therefore be useful for proving theoretical results about coercive subtyping in ELF,

CLF, LLF or PAL$^+$. One of the strengths of lambda-free logical frameworks is that their modularity means that results usually have this general applicability.

6. Conclusion

We have shown how to extend the lambda-free logical framework TF with coercive subtyping. We have shown how several metatheoretic properties may be proven for such a system, and observed that, once we are past the basic metatheoretic properties, the proofs are generally shorter and simpler than when we work in a traditional logical framework. We have show how many of these results may then be lifted to a traditional logical framework.

Lambda-free logical frameworks provide an alternative to traditional logical frameworks, with some advantages and some disadvantages. They do not eliminate all technical complexity by any means; rather, they move it to a different place. They have a simpler syntax and fewer rules of deduction, at the expense of needing complex defined operations and judgement forms. Their advanced properties are easier to prove, but the basic metatheoretic properties are much more difficult to prove.

While we have worked in TF for this paper, the techniques developed can be applied to all the subsystems of TF in the modular hierarchy. The results we have proved here are very general, and should therefore prove a powerful tool for proving the theoretical properties of a wide range of logical frameworks with coercive subtyping.

Acknowledgments

Thanks to Zhaohui Luo and Sergei Soloviev for fruitful discussions during this research. Many thanks also to the anonymous referees for their very detailed comments and suggestions for additions to this paper.

References

Peter Aczel. Simple overloading for type theories. Draft communicated to Z. Luo., 1994.

Peter Aczel. Yet another logical framework. Unpublished, 2001.

Robin Adams. Lambda-free logical frameworks. *Annals of Pure and Applied Logic*, 2009. Submitted. Available at `http://arxiv.org/abs/0804.1879`.

Robin Adams. A modular hierarchy of logical frameworks. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *LNCS*, pages 1–16. Springer, 2004a.

Robin Adams. *A Modular Hierarchy of Logical Frameworks*. PhD thesis, University of Manchester, 2004b.

Anthony Bailey. *The Machine-Checked Literate Formalisation of Algebra in Type Theory*. PhD thesis, University of Manchester, 1998.

Bruno Barras and et al. *The Coq Proof Assistant Reference Manual*, May 2000.

Paul Callaghan and Zhaohui Luo. An implementation of LF with coercive subtyping and universes. *Journal of Automated Reasoning*, 27(1):3–27, 2001.

Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Università di Torino, 1996.

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, November 2002.

H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation*, 34(4): 271–286, 2002. Special issue on the integration of automated reasoning and computer algebra systems.

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007. ISSN 0956-7968. doi: 10.1017/S0956796807006430.

William Lovas and Frank Pfenning. A bidirectional refinement type system for LF. *Electron. Notes Theor. Comput. Sci.*, 196:113–128, 2008. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2007.09.021.

Yong Luo and Zhaohui Luo. Coherence and transitivity in coercive subtyping. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of the Artificial Intelligence on Logic for Programming*, volume 2250 of *LNCS*, pages 249–265. Springer, 2001.

Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

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

Zhaohui Luo. Coercive subtyping in type theory. In Dirk van Dalen and Mark Bezem, editors, *Computer Science Logic: 10th International Workshop, CSL'96, Utrecht, The Netherlands, September 1996, Selected Papers*, volume 1258 of *LNCS*, pages 276–296. Springer, 1996.

Zhaohui Luo. Manifest fields and module mechanisms in intensional type theory. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *Types for Proofs and Programs, Proc. of Inter. Conf. of TYPES'08*, volume 5497 of *LNCS*, pages 237–255. Springer, 2009.

Zhaohui Luo. PAL+: A lambda-free logical framework. *Journal of Functional Programming*, 13(2):317–338, 2003.

Zhaohui Luo and Robin Adams. Structural subtyping for inductive types with functorial equality rules. *Mathematical Structures in Computer Science*, 2006. To appear.

Zhaohui Luo and Yong Luo. Transitivity in coercive subtyping. *Information and Computation*, 197(1–2):122–144, 2005.

Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Department of Computer Science, University of Edinburgh, May 1992.

Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory: an Introduction*. Oxford University Press, 1990.

Gordon Plotkin. An algebraic framework for logics and type theories. Talk given at LFMTP'06, August 2006.

Randy Pollack. Some recent logical frameworks. Talk given at ProgLog, February 2007.

Sergei Soloviev and Zhaohui Luo. Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 113(1–3): 297–322, 2002.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: the propositional fragment. In *Types for Proofs and Programs*, LNCS, Torino, Italy, April 2003. Springer-Verlag.