Coercive Subtyping in Lambda-free Logical Frameworks

LFMTP'09

Robin Adams

robin@cs.rhul.ac.uk

Royal Holloway, University of London

2 August 2009

Introduction

A lambda-free logical framework:

- uses only *canonical forms*

  β-short, η-long forms

- does not use:

  application, framework-level reduction, substitution

Introduction

A lambda-free logical framework:

- uses only *canonical forms*

  β-*short*, η-*long forms*

- does not use:

  application, *framework-level reduction*, *substitution*

Coercive subtyping:

- To treat A as if it were a subtype of B,
- we can declare $c : (A)B$ to be a *coercion*.
- Identify $a : A$ with $ca : B$.

Introduction

A lambda-free logical framework:

- uses only *canonical forms*

  β-*short*, η-*long forms*

- does not use:

  application, *framework-level reduction*, *substitution*

Coercive subtyping:

- To treat A as if it were a subtype of B,
- we can declare $c : (A)B$ to be a *coercion*.
- Identify $a : A$ with $ca : B$.

The Problem

How to add coercive subtyping to a lambda-free logical framework? Lambda-free frameworks do not have meta-functions, kinds $(A)B$, application, ...

1. Lambda-free Logical Frameworks
   - Problems with Logical Frameworks
   - The Modular Hierarchy

2. Coercive Subtyping in Lambda-free Logical Frameworks
   - Coercive Subtyping
   - Coercive Application and Typecasting
   - Typecasting in TF
   - Coherence and Decidability

Logical Frameworks

A logical framework F is a *metalanguage* in which we represent an *object theory* T.
We have objects x, c, $[x : K]k$, kk', ...
and kinds **Type**, $\mathrm{El}\,(k)$, $(x : K)K'$, ...
We declare *constants* and *computation rules* such that

- the objects in **Type** behave like the types in T;
- the objects in $\mathrm{El}\,(A)$ behave like the terms of type A in T.

Logical Frameworks

A logical framework F is a *metalanguage* in which we represent an *object theory* T.
We have objects x, c, $[x : K]k$, kk', ...
and kinds **Type**, $\mathrm{El}\,(k)$, $(x : K)K'$, ...
We declare *constants* and *computation rules* such that

- the objects in **Type** behave like the types in T;
- the objects in $\mathrm{El}\,(A)$ behave like the terms of type A in T.

Example:

$$\Pi : (A : \mathbf{Type})((A)\mathbf{Type})\mathbf{Type}$$

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x : A \vdash B \text{ type}}{\Gamma \vdash \Pi x : A.B \text{ type}}$$

$$\frac{\Gamma \vdash A = A' \quad \Gamma, x : A \vdash B = B'}{\Gamma \vdash \Pi x : A.B = \Pi x : A'.B'}$$

Problems with Logical Frameworks

Typically:

- Each entity of the object theory is represented by a $\beta\eta$-*equivalence class* of objects.
- There are objects that do not represent entities (partial application).
- Framework-level reduction and object theory reduction often interact.

Example

The object ΠA does not correspond to any object theory entity.
Instead, it represents the *meta-function* that maps B to $\Pi x : A.B$.

Problems with Logical Frameworks

Typically:

- Each entity of the object theory is represented by a $\beta\eta$-*equivalence class* of objects.
- There are objects that do not represent entities (partial application).
- Framework-level reduction and object theory reduction often interact.

These features are desirable in practice (e.g. abbreviation mechanisms) but awkward for theoretical work.

Problems with Logical Frameworks

Typically:

- Each entity of the object theory is represented by a $\beta\eta$-*equivalence class* of objects.
- There are objects that do not represent entities (partial application).
- Framework-level reduction and object theory reduction often interact.

These features are desirable in practice (e.g. abbreviation mechanisms) but awkward for theoretical work.

Idea:

- construct a logical framework L which deals with only β-*short*, η-*long forms.*
  - Each object theory is represented by a *unique* object.
  - Partial application is disallowed.
  - There is no framework-level reduction.

Problems with Logical Frameworks

Typically:

- Each entity of the object theory is represented by a $\beta\eta$-*equivalence class* of objects.
- There are objects that do not represent entities (partial application).
- Framework-level reduction and object theory reduction often interact.

These features are desirable in practice (e.g. abbreviation mechanisms) but awkward for theoretical work.

Idea:

- construct a logical framework L which deals with only β-*short*, η-*long forms.*
- Embed L in F — show that L is isomorphic to a conservative subsystem F.

Problems with Logical Frameworks

Typically:

- Each entity of the object theory is represented by a $\beta\eta$-*equivalence class* of objects.
- There are objects that do not represent entities (partial application).
- Framework-level reduction and object theory reduction often interact.

These features are desirable in practice (e.g. abbreviation mechanisms) but awkward for theoretical work.

Idea:

- construct a logical framework L which deals with only β-*short*, η-*long forms.*
- Embed L in F — show that L is isomorphic to a conservative subsystem F.
- Lift results — prove results in L, deduce they hold for F.

The Framework TF

The *Type Framework* TF, developed by Aczel (2001).

The Framework TF

The *Type Framework* TF, developed by Aczel (2001).
Each variable and constant z has an *order*, $o(z)$.
An object M has the form

$$z[[x_{11}, \ldots, x_{1r_1}]M_1, \ldots, [x_{n1}, \ldots, x_{nr_n}]M_n]$$

where $o(x_{ij}) \leq o(z) - 2$.

The Framework TF

The *Type Framework* TF, developed by Aczel (2001).
Each variable and constant z has an *order*, $o(z)$.
An object M has the form

$$z[[x_{11}, \ldots, x_{1r_1}]M_1, \ldots, [x_{n1}, \ldots, x_{nr_n}]M_n]$$

where $o(x_{ij}) \leq o(z) - 2$.
An abstraction is an expression

$$[x_1, \ldots, x_r]M$$

Its *order* is $\max_i o(x_i) + 1$.

The Framework TF

The *Type Framework* TF, developed by Aczel (2001).
Each variable and constant z has an *order*, $o(z)$.
An object M has the form

$$z[[x_{11},\ldots,x_{1r_1}]M_1,\ldots,[x_{n1},\ldots,x_{nr_n}]M_n]$$

where $o(x_{ij}) \leq o(z) - 2$.
An abstraction is an expression

$$[x_1,\ldots,x_r]M$$

Its *order* is $\max_i o(x_i) + 1$.
In place of substitution, use instantiation (*hereditary substitution*):
$\{F/x\}M$ is the normal form of $[F/x]M$.
Definition

$$\begin{aligned} \{F/x\}z[\vec{G}] &\equiv z[\{F/x\}\vec{G}] \qquad (x \not\equiv z) \\ \{[\vec{y}]M/x\}x[\vec{G}] &\equiv \{\{[\vec{y}]M/x\}\vec{G}/\vec{y}\}M \end{aligned}$$

Termination is guaranteed by the orders.

Judgements of TF

The *judgements* of TF have the forms

$$\Gamma \text{ valid}$$
$$\Gamma \vdash A : \mathbf{Type}$$
$$\Gamma \vdash A = B : \mathbf{Type}$$
$$\Gamma \vdash M : \mathrm{El}\,(A)$$
$$\Gamma \vdash M = N : \mathrm{El}\,(A)$$

Judgements of TF

The *judgements* of TF have the forms

$$\Gamma \text{ valid}$$
$$\Gamma \vdash A : \mathbf{Type}$$
$$\Gamma \vdash A = B : \mathbf{Type}$$
$$\Gamma \vdash M : \mathrm{El}\,(A)$$
$$\Gamma \vdash M = N : \mathrm{El}\,(A)$$

Declaring the constant $\Pi : (A : \mathbf{Type})(B : (A)\mathbf{Type})\mathbf{Type}$ introduces the rule of deduction

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

The Modular Hierarchy of Logical Frameworks

A set of subsystems of TF that extend one another conservatively.

	No equation declarations	Equation declarations
Parameters of small type only	$\mathbf{SPar}\,(n)^-$	$\mathbf{SPar}\,(n)$
Parameters of small type and large kind	$\mathbf{LPar}\,(n)^-$	$\mathbf{LPar}\,(n)$

The Modular Hierarchy of Logical Frameworks

A set of subsystems of TF that extend one another conservatively.

	No equation declarations	Equation declarations
Parameters of small type only	$\mathbf{SPar}\,(n)^-$	$\mathbf{SPar}\,(n)$
Parameters of small type and large kind	$\mathbf{LPar}\,(n)^-$	$\mathbf{LPar}\,(n)$

If we declare $c : (x_1 : (\Delta_1)\,T_1) \cdots (x_m : (\Delta_m)\,T_m)\,T$, then:

- the *parameters* of c are $x_1, \ldots, x_n$;
- x_i is of *small type* if $T_i \equiv \mathrm{El}\,(A_i)$;
- x_i is of *large kind* if $T_i \equiv \mathbf{Type}$.

The number n is the largest order of constant that may appear.

Frameworks in the Modular Hierarchy

Lambda-free frameworks can be *embedded* in traditional frameworks:

- $\mathbf{SPar}\,(\omega)^- \hookrightarrow \mathit{ELF}$
- $\mathbf{LPar}\,(\omega) \hookrightarrow$ Martin-Löf Logical Framework
- $\mathbf{LPar}\,(\omega) \hookrightarrow \mathit{PAL}^+$
- $\mathbf{LPar}\,(1)^- \hookrightarrow \mathit{PAL}$, $\mathbf{SPar}\,(\omega)^- \hookrightarrow \mathit{AUT}-68$, ...

▸ Section II

History of Lambda-free Frameworks

- 1996 — Linear Logical Framework (Cervesato)
- 2000 — PAL$^+$ (Luo)
  Does not allow partial application
  Does have abstractions, framework-level reduction
- 2001 — Type Framework, TF (Aczel)
- 2003 — Concurrent Logical Framework (Watkins, Cervesato, Pfenning, Walker)
- 2004 — Modular Hierarchy of Logical Frameworks (Adams [Ada04, Ada09])
- 2006 — DMBEL (Plotkin [Plo06])
- 2007 — Canonical LF (Harper, Licata [HL07])

Part II — Coercive Subtyping

We wish to regard A as a *subtype* of B, $A < B$:

Every object of A is an object of B

- Construct a meta-function $c : (A)B$;
- Declare c to be a *coercion*

$$A <_c B$$

We wish to *identify* every object $a : A$ with the object $c(a) : B$.

Examples

1. $Nat <_c Int$, where c maps n to $+n$.
2. $Bool <_d Prop$, where $d(b)$ is $b =_{Bool} tt$.
3. $Vec(A, n) < List(A)$

Theory studied by Luo, Luo and Soloviev.
Implemented in Plastic, Coq, LEGOwcs.

Coercive Application

We can extend a logical framework with λ (such as LF) with coercive subtyping and *coercive application*:

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

Coercive Application

We can extend a logical framework with λ (such as LF) with coercive subtyping and *coercive application*:

$$\frac{\Gamma \vdash f : (x : B)K \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A <_c B}{\begin{array}{c}\Gamma \vdash fa : [ca/x]K \\ \Gamma \vdash fa = f(ca) : [ca/x]K\end{array}}$$

$A <_c B$ means:

Coercive Application

We can extend a logical framework with λ (such as LF) with coercive subtyping and *coercive application*:

$$\frac{\Gamma \vdash f : (x : B)K \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A <_c B}{\begin{array}{c}\Gamma \vdash fa : [ca/x]K \\ \Gamma \vdash fa = f(ca) : [ca/x]K\end{array}}$$

$A <_c B$ means:
'Identify $a : A$ with $ca : B$'

Coercive Application

We can extend a logical framework with λ (such as LF) with coercive subtyping and *coercive application*:

$$\frac{\Gamma \vdash f : (x : B)K \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A <_c B}{\begin{array}{c}\Gamma \vdash fa : [ca/x]K \\ \Gamma \vdash fa = f(ca) : [ca/x]K\end{array}}$$

$A <_c B$ means:
'Whenever the machine wants an object of type B, the user may enter an object $a : A$; the machine is to proceed as if he had entered $c(a)$'

Coercive Application

We can extend a logical framework with λ (such as LF) with coercive subtyping and *coercive application*:

$$\frac{\Gamma \vdash f : (x : B)K \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A <_c B}{\begin{array}{c}\Gamma \vdash fa : [ca/x]K \\ \Gamma \vdash fa = f(ca) : [ca/x]K\end{array}}$$

$A <_c B$ means:
For any meta-function f that takes arguments of type B and $a : A$, fa is well-typed and definitionally equal to $f(ca)$.

Coercive Application

We can extend a logical framework with λ (such as LF) with coercive subtyping and *coercive application*:

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

$A <_c B$ means:
For any meta-function f that takes arguments of type B and $a : A$, fa is well-typed and definitionally equal to $f(ca)$.

Problem

How can we add coercive subtyping to a lambda-free logical framework, that does not have:
meta-functions, application, kinds $(x : B)K$?

Typecasting

Another way to add coercive subtyping to a logical framework: *typecasting*.

Typecasting

Another way to add coercive subtyping to a logical framework: *typecasting*.
If $a : A$ and $A < B$, introduce the object

$$a_B$$

'a considered as an object of type B'
or 'the object of type B that we are identifying with a.'

Typecasting

Another way to add coercive subtyping to a logical framework: *typecasting*.
If $a : A$ and $A < B$, introduce the object

$$a_B$$

'a considered as an object of type B'
or 'the object of type B that we are identifying with a.'

- If $a : A$ then $a_A : A$ and $a_A = a : A$.
- If $a : A$ and $A <_c B$ then $a_B : B$ and $a_B = ca : B$.

Coercive Application and Typecasting

Coercive application and typecasting are *equivalent* — given one, the other can be defined.

Coercive Application → Typecasting

$$a_B \equiv ([x : B]x)a$$

Typecasting → Coercive Application

If f takes arguments of type B, set

$$fa \equiv f(a_B)$$

(Requires inference of kinds.)

Typecasting in TF

Extend TF with:

- new judgements of the form $A <_{[x]N} B$:
  'A is a subtype of B,
  identify $M : A$ with $\{M/x\}N : B$.
- new objects of the form M_A,
  'M typecast to be of type A'
- new rules of deduction:

$$\frac{\Gamma \vdash M : \mathrm{El}\,(A)}{\begin{array}{c}\Gamma \vdash M_A : \mathrm{El}\,(A) \\ \Gamma \vdash M_A = M : \mathrm{El}\,(A)\end{array}}$$

$$\frac{\Gamma \vdash M : \mathrm{El}\,(A) \quad \Gamma \vdash A <_{[x]N} B}{\begin{array}{c}\Gamma \vdash M_B : \mathrm{El}\,(B) \\ \Gamma \vdash M_B = \{M/x\}N : \mathrm{El}\,(B)\end{array}}$$

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

Let

- LF be Martin-Löf's Logical Framework
- TF be the lambda-free logical framework
- $\mathrm{LF}_<$ be LF + coercive subtyping + coercive application
- $\mathrm{TF}_<$ be TF + coercive subtyping + typecasting

Typecasting and coercive application are equivalent.
We can make use of this equivalence to embed $\mathrm{TF}_<$ in $\mathrm{LF}_<$.

$$\mathrm{LF}_< \qquad \rightleftarrows \qquad \mathrm{TF}_<$$

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

Let

- LF be Martin-Löf's Logical Framework
- TF be the lambda-free logical framework
- $\mathrm{LF}_<$ be LF + coercive subtyping + coercive application
- $\mathrm{TF}_<$ be TF + coercive subtyping + typecasting

Typecasting and coercive application are equivalent.
We can make use of this equivalence to embed $\mathrm{TF}_<$ in $\mathrm{LF}_<$.

$$\begin{array}{ccc} \mathrm{LF}_< & \rightleftarrows & \mathrm{TF}_< \\ FA & \to & NF(F) \bullet NF(A)_{NF(B)} \end{array}$$

where F takes arguments of type B.

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

Let

- LF be Martin-Löf's Logical Framework
- TF be the lambda-free logical framework
- $\mathrm{LF}_<$ be LF + coercive subtyping + coercive application
- $\mathrm{TF}_<$ be TF + coercive subtyping + typecasting

Typecasting and coercive application are equivalent.
We can make use of this equivalence to embed $\mathrm{TF}_<$ in $\mathrm{LF}_<$.

$$\begin{array}{ccc} \mathrm{LF}_< & \rightleftarrows & \mathrm{TF}_< \\ FA & \to & NF(F) \bullet NF(A)_{NF(B)} \\ ([x:A]x)M & \leftarrow & M_A \end{array}$$

where F takes arguments of type B. These mappings are sound and mutually inverse.

Coherence

A type theory declared in $\mathrm{TF}_<$ (or $\mathrm{LF}_<$) is *coherent* iff:

- whenever $\Gamma \vdash A <_{[x]M} B$ and $\Gamma \vdash A <_{[x]N} B$, then $\Gamma, x : A \vdash M = N : B$.
- ...

Coherence

A type theory declared in $\mathrm{TF}_<$ (or $\mathrm{LF}_<$) is *coherent* iff:

- whenever $\Gamma \vdash A <_{[x]M} B$ and $\Gamma \vdash A <_{[x]N} B$, then $\Gamma, x : A \vdash M = N : B$.
- ...

Let T be a type theory declared in TF.
Extend with subtyping rules to a type theory $T[\mathcal{C}]$ in $\mathrm{TF}_<$

Coherence

A type theory declared in $\mathrm{TF}_<$ (or $\mathrm{LF}_<$) is *coherent* iff:

- whenever $\Gamma \vdash A <_{[x]M} B$ and $\Gamma \vdash A <_{[x]N} B$, then $\Gamma, x : A \vdash M = N : B$.
- ...

Let T be a type theory declared in TF.
Extend with subtyping rules to a type theory $T[\mathcal{C}]$ in $\mathrm{TF}_<$

Theorem (Insertion of Coercions)

If $T[\mathcal{C}]$ is coherent, and $\mathcal{J} \equiv \Gamma \vdash M : T$ is derivable in $T[\mathcal{C}]$, then there is a judgement $\overline{\mathcal{J}} \equiv \overline{\Gamma} \vdash \overline{M} : \overline{T}$ derivable in T such that

$$\Vdash \Gamma = \overline{\Gamma}, \qquad \Gamma \vdash T = \overline{T}, \qquad \Gamma \vdash M = \overline{M}$$

Coherence

A type theory declared in $\mathrm{TF}_<$ (or $\mathrm{LF}_<$) is *coherent* iff:

- whenever $\Gamma \vdash A <_{[x]M} B$ and $\Gamma \vdash A <_{[x]N} B$, then $\Gamma, x : A \vdash M = N : B$.
- ...

Let T be a type theory declared in TF.
Extend with subtyping rules to a type theory $T[\mathcal{C}]$ in $\mathrm{TF}_<$

Theorem (Insertion of Coercions)

If $T[\mathcal{C}]$ is coherent, and $\mathcal{J} \equiv \Gamma \vdash M : T$ is derivable in $T[\mathcal{C}]$, then there is a judgement $\overline{\mathcal{J}} \equiv \overline{\Gamma} \vdash \overline{M} : \overline{T}$ derivable in T such that

$$\Vdash \Gamma = \overline{\Gamma}, \qquad \Gamma \vdash T = \overline{T}, \qquad \Gamma \vdash M = \overline{M}$$

The proof consists of replacing a_B with $\{a/x\}M$, where $A <_{[x]M} B$. The derivation of $\mathcal{J}$ tells us which coercion $[x]M$ to use.

Coherence

A type theory declared in $\mathrm{TF}_<$ (or $\mathrm{LF}_<$) is *coherent* iff:

- whenever $\Gamma \vdash A <_{[x]M} B$ and $\Gamma \vdash A <_{[x]N} B$, then $\Gamma, x : A \vdash M = N : B$.
- ...

Let T be a type theory declared in TF.
Extend with subtyping rules to a type theory $T[\mathcal{C}]$ in $\mathrm{TF}_<$

Theorem (Insertion of Coercions)

If $T[\mathcal{C}]$ is coherent, and $\mathcal{J} \equiv \Gamma \vdash M : T$ is derivable in $T[\mathcal{C}]$, then there is a judgement $\overline{\mathcal{J}} \equiv \overline{\Gamma} \vdash \overline{M} : \overline{T}$ derivable in T such that

$$\Vdash \Gamma = \overline{\Gamma}, \qquad \Gamma \vdash T = \overline{T}, \qquad \Gamma \vdash M = \overline{M}$$

The theorem holds in $\mathrm{TF}_<$.
$\therefore$ The theorem holds in $\mathrm{LF}_<$.

Other Results

Let us say that the set of subtyping judgements in $T[\mathcal{C}]$ is *computable* iff there exists an algorithm that, given Γ, A, B,
decides if there exists c such that $\Gamma \vdash A <_c B$
and, if so, returns such a c.

Theorem

If the subtyping judgements of $T[\mathcal{C}]$ are computable, then $\overline{\mathcal{J}}$ depends only on the judgement $\mathcal{J}$, and not on the derivation of $\mathcal{J}$.

Corollary

If typechecking in T is decidable, and the coercion judgements of $T[\mathcal{C}]$ are computable, then typechecking in $T[\mathcal{C}]$ is decidable.

These theorems can be proved for all the systems in the modular hierarchy by exactly the same proofs.

Other Results

Using our embeddings, we therefore have:

Corollary

The previous three theorems all hold in $\mathrm{LF}_<$*,*
ELF extended with coercive subtyping,
PAL$^+$ *extended with coercive subtyping,*
DMBEL extended with coercive subtyping, . . .

Conclusion

- Lambda-free logical frameworks can be used as frameworks in their own right,
  or as tools for proving results about traditional frameworks.

Conclusion

- Lambda-free logical frameworks can be used as frameworks in their own right,
  or as tools for proving results about traditional frameworks.
- We can add coercive subtyping to a lambda-free framework
  by using *typecasting* in place of coercive application.

Conclusion

- Lambda-free logical frameworks can be used as frameworks in their own right,
  or as tools for proving results about traditional frameworks.
- We can add coercive subtyping to a lambda-free framework
  by using *typecasting* in place of coercive application.
- Many results become easier to prove in this setting:
  insertion of coercions and decidability of typechecking.

Conclusion

- Lambda-free logical frameworks can be used as frameworks in their own right,
  or as tools for proving results about traditional frameworks.
- We can add coercive subtyping to a lambda-free framework
  by using *typecasting* in place of coercive application.
- Many results become easier to prove in this setting:
  insertion of coercions and decidability of typechecking.
- They can be proved for a family of frameworks simultaneously.

Conclusion

- Lambda-free logical frameworks can be used as frameworks in their own right,
  or as tools for proving results about traditional frameworks.
- We can add coercive subtyping to a lambda-free framework
  by using *typecasting* in place of coercive application.
- Many results become easier to prove in this setting:
  insertion of coercions and decidability of typechecking.
- They can be proved for a family of frameworks simultaneously.
- The results can then be 'lifted' to traditional frameworks.

References


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


Robin Adams.
Lambda-free logical frameworks.
Submitted to *Annals of Pure and Applied Logic*, 2009.


Robert Harper and Daniel R. Licata.
Mechanizing metatheory in a logical framework.
Journal of Functional Programming, 17(4–5):613–673, July 2007.


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