Proof Reuse in Logical Frameworks

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

TYPES 2010, Warsaw
15 October 2010
I am a logician. What does a logician do?

- We write down a bunch of systems, and then we ask:
  - . . . which theorems are provable in each?
  - . . . which of these are equivalent?
  - . . . which of these are subsystems of one another?
  - . . . what translations exist from one to another?
  - . . . what are the metatheoretic properties of the systems?

I am interested in proof assistants as tools for experimenting with systems of logic. How well do proof assistants offer machine support for this work?

Robin Adams

Proof Reuse in Logical Frameworks
I am a logician. What does a logician do?
I am a logician. What does a logician do? We write down a bunch of systems, and then we ask:
I am a logician. What does a logician do? We write down a bunch of systems, and then we ask:

- . . . which theorems are provable in each?
- . . . which of these are equivalent?
- . . . which of these are subsystems of one another?
- . . . what translations exist from one to another?
- . . . what are the metatheoretic properties of the systems?
I am a logician. What does a logician do? We write down a bunch of systems, and then we ask:

- ...which theorems are provable in each?
- ...which of these are equivalent?
- ...which of these are subsystems of one another?
- ...what translations exist from one to another?
- ...what are the metatheoretic properties of the systems?
Motivation

I am a logician. What does a logician do? We write down a bunch of systems, and then we ask:

- ... which theorems are provable in each?
- ... which of these are equivalent?
- ... which of these are subsystems of one another?
- ... what translations exist from one to another?
- ... what are the metatheoretic properties of the systems?

I am interested in proof assistants as tools for experimenting with systems of logic.
I am a logician. What does a logician do? We write down a bunch of systems, and then we ask:

- ... which theorems are provable in each?
- ... which of these are equivalent?
- ... which of these are subsystems of one another?
- ... what translations exist from one to another?
- ... what are the metatheoretic properties of the systems?

I am interested in proof assistants as tools for **experimenting** with systems of logic. How well do proof assistants offer machine support for this work?
Not Very Well

The usual plan of attack:

1. Choose one system of logic.
2. Implement a proof checker for that one system.
3. Build up a big library of formalised results in that system.

We have to start from scratch with each new system.

There are also logical frameworks (Isabelle, TWELF, Plastic, ...)

- They can implement more than one system of logic.
- But there is no easy way to use results from one system when working in another.
The usual plan of attack:

- Choose one system of logic.
- Implement a proof checker for that one system.
- Build up a big library of formalised results in that system.

We have to start from scratch with each new system.
The usual plan of attack:

- Choose one system of logic.
- Implement a proof checker for that one system.
- Build up a big library of formalised results in that system.

We have to start from scratch with each new system. There are also logical frameworks (Isabelle, TWELF, Plastic, . . .)

- They can implement more than one system of logic.
- But there is no easy way to use results from one system when working in another.
The Problem

There exist *sound translations* between logical systems.
Proof Reuse in Logical Frameworks

The Problem

There exist *sound translations* between logical systems.

**Definition**

A *sound translation* from \( S \) to \( T \) is a mapping

\[
\Phi : \text{propositions of } S \rightarrow \text{propositions of } T
\]

such that

*If* \( S \vdash \alpha \) *then* \( T \vdash \Phi(\alpha) \).

*(This includes the case* \( S \hookrightarrow \rightarrow \rightarrow T \) — *take* \( \Phi \) *to be the identity.)*
The Problem

There exist *sound translations* between logical systems.

**Definition**

A *sound translation* from $S$ to $T$ is a mapping

$$\Phi : \text{propositions of } S \rightarrow \text{propositions of } T$$

such that

*If $S \vdash \alpha$ then $T \vdash \Phi(\alpha)$."

(This includes the case $S \leftrightarrow T$ — take $\Phi$ to be the identity.)
There exist *sound translations* between logical systems.

**Definition**

A *sound translation* from $S$ to $T$ is a mapping

$$\Phi : \text{propositions of } S \rightarrow \text{propositions of } T$$

such that

*If $S \vdash \alpha$ then $T \vdash \Phi(\alpha)$.***

(This includes the case $S \leftrightarrow T$ — take $\Phi$ to be the identity.)

I want to:

- declare $S$ and $T$ in some logical framework;
- prove $\alpha$ in $S$
- and *immediately* have $\Phi(\alpha)$ available when working in $T$. 
Arithmetic in LF

To declare *Heyting arithmetic* in LF, we give ourselves:
- a kind `Term`, and constants
  
  \[
  0 : \text{Term}, \quad S : \text{Term} \rightarrow \text{Term}, \\
  + : \text{Term} \rightarrow \text{Term} \rightarrow \text{Term}, \quad \times : \text{Term} \rightarrow \text{Term} \rightarrow \text{Term};
  \]
To declare *Heyting arithmetic* in LF, we give ourselves:

- a kind `Term`, and constants
  
  
  \[0 : \text{Term}, \quad S : \text{Term} \rightarrow \text{Term},\]
  
  \[+ : \text{Term} \rightarrow \text{Term} \rightarrow \text{Term}, \quad \times : \text{Term} \rightarrow \text{Term} \rightarrow \text{Term} ;\]

- a kind `Prop_I`, and constants
  
  \[=_I : \text{Term} \rightarrow \text{Term} \rightarrow \text{Prop}_I, \quad \lor_I : \text{Prop}_I \rightarrow \text{Prop}_I \rightarrow \text{Prop}_I,\]
  
  \[\forall_I : (\text{Term} \rightarrow \text{Prop}_I) \rightarrow \text{Prop}_I, \ldots ;\]
To declare *Heyting arithmetic* in LF, we give ourselves:

- a kind $\text{Term}$, and constants
  
  $0 : \text{Term}, \quad S : \text{Term} \to \text{Term},$
  
  $+: \text{Term} \to \text{Term} \to \text{Term}, \quad \times : \text{Term} \to \text{Term} \to \text{Term}$;

- a kind $\text{Prop}_I$, and constants
  
  $=_I : \text{Term} \to \text{Term} \to \text{Prop}_I, \quad \lor_I : \text{Prop}_I \to \text{Prop}_I \to \text{Prop}_I,$
  
  $\forall_I : (\text{Term} \to \text{Prop}_I) \to \text{Prop}_I, \ldots$;

- for every $P : \text{Prop}_I$, a kind $\text{Prf}(P)$;
Arithmetic in LF

To declare *Heyting arithmetic* in LF, we give ourselves:

- a kind `Term`, and constants

\[
0 : \text{Term}, \quad S : \text{Term} \to \text{Term}, \\
+ : \text{Term} \to \text{Term} \to \text{Term}, \quad \times : \text{Term} \to \text{Term} \to \text{Term};
\]

- a kind `Prop_I`, and constants

\[
\equiv_I : \text{Term} \to \text{Term} \to \text{Prop}_I, \quad \lor_I : \text{Prop}_I \to \text{Prop}_I \to \text{Prop}_I, \\
\forall_I : (\text{Term} \to \text{Prop}_I) \to \text{Prop}_I, \ldots ;
\]

- for every \( P : \text{Prop}_I \), a kind `Prf (P)`;

- constants for the rules of deduction:

\[
\begin{array}{c}
[P] \\
\vdots \\
R \\
\hline
R \\
\end{array}
\begin{array}{c}
[Q] \\
\vdots \\
R \\
\hline
R \\
\end{array}
\begin{array}{c}
P \lor Q \\
\hline
R
\end{array}
\]
Arithmetic in LF

To declare *Heyting arithmetic* in LF, we give ourselves:

- a kind $\text{Term}$, and constants
  
  $0 : \text{Term}$, $\text{S} : \text{Term} \rightarrow \text{Term}$,
  
  $\oplus : \text{Term} \rightarrow \text{Term} \rightarrow \text{Term}$, $\times : \text{Term} \rightarrow \text{Term} \rightarrow \text{Term}$;

- a kind $\text{Prop}_I$, and constants
  
  $=_I : \text{Term} \rightarrow \text{Term} \rightarrow \text{Prop}_I$, $\lor_I : \text{Prop}_I \rightarrow \text{Prop}_I \rightarrow \text{Prop}_I$,
  
  $\forall_I : (\text{Term} \rightarrow \text{Prop}_I) \rightarrow \text{Prop}_I$, ...

- for every $P : \text{Prop}_I$, a kind $\text{Prf}(P)$;

- constants for the rules of deduction:

  $\lor E : (P, Q, R : \text{Prop}_I)$
  
  $(\text{Prf}(P) \rightarrow \text{Prf}(R)) \rightarrow$
  
  $(\text{Prf}(Q) \rightarrow \text{Prf}(R)) \rightarrow$
  
  $\text{Prf}(P \lor_I Q) \rightarrow \text{Prf}(R)$
To declare \textit{Peano arithmetic} in LF, we give ourselves:

- a kind \textbf{Term}, and constants
  \[
  0 : \text{Term}, \quad S : \text{Term} \rightarrow \text{Term},
  \]
  \[
  + : \text{Term} \rightarrow \text{Term} \rightarrow \text{Term}, \quad \times : \text{Term} \rightarrow \text{Term} \rightarrow \text{Term};
  \]
- a kind \textbf{Prop}, and constants
  \[
  =_I : \text{Term} \rightarrow \text{Term} \rightarrow \text{Prop}, \quad \lor_I : \text{Prop} \rightarrow \text{Prop} \rightarrow \text{Prop},
  \]
  \[
  \forall_I : (\text{Term} \rightarrow \text{Prop}) \rightarrow \text{Prop}, \ldots;
  \]
- for every \( P : \text{Prop} \), a kind \textbf{Prf}(P);
- constants for the rules of deduction:
  \[
  \lor E : (P, Q, R : \text{Prop})
  \]
  \[
  (\text{Prf}(P) \rightarrow \text{Prf}(R)) \rightarrow
  \]
  \[
  (\text{Prf}(Q) \rightarrow \text{Prf}(R)) \rightarrow
  \]
  \[
  \text{Prf}(P \lor_I Q) \rightarrow \text{Prf}(R)
  \]
To declare *Peano arithmetic* in LF, we give ourselves:

- a kind \textbf{Term}, and constants
  
  \[
  0 : \textbf{Term}, \quad S : \textbf{Term} \rightarrow \textbf{Term},
  \]
  
  \[
  + : \textbf{Term} \rightarrow \textbf{Term} \rightarrow \textbf{Term}, \quad \times : \textbf{Term} \rightarrow \textbf{Term} \rightarrow \textbf{Term} ;
  \]

- a kind \textbf{Prop}_C, and constants
  
  \[
  =_C : \textbf{Term} \rightarrow \textbf{Term} \rightarrow \textbf{Prop}_C, \quad \lor_C : \textbf{Prop}_C \rightarrow \textbf{Prop}_C \rightarrow \textbf{Prop}_C
  \]
  
  \[
  \forall_C : (\textbf{Term} \rightarrow \textbf{Prop}_C) \rightarrow \textbf{Prop}_C, \ldots ;
  \]

- for every \( P : \textbf{Prop}_C \), a kind \textbf{Prf} (\( P \));

- constants for the rules of deduction:

  \[
  \lor E : (P, Q, R : \textbf{Prop}_C)
  \]
  
  \[
  (\textbf{Prf} (P) \rightarrow \textbf{Prf} (R)) \rightarrow
  \]
  
  \[
  (\textbf{Prf} (Q) \rightarrow \textbf{Prf} (R)) \rightarrow
  \]
  
  \[
  \textbf{Prf} (P \lor I Q) \rightarrow \textbf{Prf} (R)
  \]
To declare *Peano arithmetic* in LF, we give ourselves:

- a kind \( \textbf{Term} \), and constants

  \[
  0 : \text{Term}, \quad S : \text{Term} \to \text{Term},
  \]
  \[
  + : \text{Term} \to \text{Term} \to \text{Term}, \quad \times : \text{Term} \to \text{Term} \to \text{Term} ;
  \]

- a kind \( \textbf{Prop}_C \), and constants

  \[
  =_C : \text{Term} \to \text{Term} \to \text{Prop}_C, \quad \lor_C : \text{Prop}_C \to \text{Prop}_C \to \text{Prop}_C
  \]
  \[
  \forall_C : (\text{Term} \to \text{Prop}_C) \to \text{Prop}_C, \ldots ;
  \]

- for every \( P : \textbf{Prop}_C \), a kind \( \text{Prf} (P) \);
- constants for the rules of deduction
- a constant for the law of excluded middle:

  \[
  EM : (P : \textbf{Prop}_C) \text{Prf} (P \lor_C \neg_C P)
  \]
Double Negation Translation

Let $PA$ be Peano Arithmetic, and $HA$ be Heyting Arithmetic.
Proof Reuse — The Idea

Double Negation Translation

Let $PA$ be Peano Arithmetic, and $HA$ be Heyting Arithmetic. For every $PA$-formula $\phi$, define the $HA$-formula $\phi^\sim$:

$(s = t)^\sim \equiv \neg\neg(s = t)$

$(\neg\phi)^\sim \equiv \neg\phi^\sim$

$(\phi \land \psi)^\sim \equiv \phi^\sim \land \psi^\sim$

$(\phi \lor \psi)^\sim \equiv \neg(\neg\phi^\sim \land \neg\psi^\sim)$

$(\phi \rightarrow \psi)^\sim \equiv \phi^\sim \rightarrow \psi^\sim$

$(\forall x \phi)^\sim \equiv \forall x \phi^\sim$

$(\exists x \phi)^\sim \equiv \neg\forall x \neg\phi^\sim$
Double Negation Translation

Let $PA$ be Peano Arithmetic, and $HA$ be Heyting Arithmetic. For every $PA$-formula $\phi$, define the $HA$-formula $\phi^{\neg\neg}$:

\[
\begin{align*}
(s = t)^{\neg\neg} & \equiv \neg\neg(s = t) \\
(\neg \phi)^{\neg\neg} & \equiv \neg \phi^{\neg\neg} \\
(\phi \land \psi)^{\neg\neg} & \equiv \phi^{\neg\neg} \land \psi^{\neg\neg} \\
(\phi \lor \psi)^{\neg\neg} & \equiv \neg(\neg \phi^{\neg\neg} \land \neg \psi^{\neg\neg}) \\
(\phi \rightarrow \psi)^{\neg\neg} & \equiv \phi^{\neg\neg} \rightarrow \psi^{\neg\neg} \\
(\forall x \phi)^{\neg\neg} & \equiv \forall x \phi^{\neg\neg} \\
(\exists x \phi)^{\neg\neg} & \equiv \neg \forall x \neg \phi^{\neg\neg}
\end{align*}
\]

Theorem (Gödel, 1933)

If $PA \vdash \phi$ then $HA \vdash \phi^{\neg\neg}$. 

Robin Adams

Proof Reuse in Logical Frameworks
Double Negation Translation

Let $PA$ be Peano Arithmetic, and $HA$ be Heyting Arithmetic. For every $PA$-formula $\phi$, define the $HA$-formula $\phi^{\neg\neg}$:

\[
\begin{align*}
(s = t)^{\neg\neg} & \equiv \neg\neg(s = t) \\
(\neg \phi)^{\neg\neg} & \equiv \neg \phi^{\neg\neg} \\
(\phi \land \psi)^{\neg\neg} & \equiv \phi^{\neg\neg} \land \psi^{\neg\neg} \\
(\phi \lor \psi)^{\neg\neg} & \equiv \neg(\neg \phi^{\neg\neg} \land \neg \psi^{\neg\neg}) \\
(\phi \rightarrow \psi)^{\neg\neg} & \equiv \phi^{\neg\neg} \rightarrow \psi^{\neg\neg} \\
(\forall x \phi)^{\neg\neg} & \equiv \forall x \phi^{\neg\neg} \\
(\exists x \phi)^{\neg\neg} & \equiv \neg \forall x \neg \phi^{\neg\neg}
\end{align*}
\]

Theorem (Gödel, 1933)

If $PA \vdash \phi$ then $HA \vdash \phi^{\neg\neg}$.

How can we make use of this theorem when working in LF?
How It Works

```
class.1f
orC : ...
```

Declare the classical system.
How It Works

Declare the intuitionistic system.
How It Works

Prove $\alpha$ in the classical system.
I don’t want to do much work now,
Proof Reuse — The Idea

How It Works

I don’t want to do much work now, because I’m lazy.
Proof Reuse — The Idea

How It Works

Write a module such that ...
... we have a proof of $\alpha$ in the intuitionistic system.
First Attempt

class.lf

\[
\begin{align*}
\neg C & : \text{Prop}_C \rightarrow \text{Prop}_C \\
\lor C & : \text{Prop}_C \rightarrow \text{Prop}_C \rightarrow \text{Prop}_C \\
\exists C & : (\text{Term} \rightarrow \text{Prop}_C) \rightarrow \text{Prop}_C
\end{align*}
\]

alpha.lf

import class;

Theorem alpha : \(A \lor_C \neg_C A\)
First Attempt

class2int.lf

\[ \mathsf{Prop}_C = \mathsf{Prop}_I \]
\[ \vdots \]
\[ \lnot C = \lnot I \]
\[ : \mathsf{Prop}_C \rightarrow \mathsf{Prop}_C \]
\[ \forall C = [P, Q : \mathsf{Prop}_C] \lnot I (\lnot I P \land I \lnot I Q) \]
\[ : \mathsf{Prop}_C \rightarrow \mathsf{Prop}_C \rightarrow \mathsf{Prop}_C \]
\[ \exists C = [P : \text{Term} \rightarrow \mathsf{Prop}_C] \lnot I \forall I [x : \text{Term}] \lnot I (Px) \]
\[ : (\text{Term} \rightarrow \mathsf{Prop}_C) \rightarrow \mathsf{Prop}_C \]
\[ \vdots \]

alpha.lf

import class2int;

Theorem alpha : \( A \lor_C \lnot_C A \)
First Attempt

class2int.lf

\[
\begin{align*}
\text{Prop}_C &= \text{Prop}_I \\
: \quad \neg C &= \neg I \\
: \quad \text{Prop}_C \to \text{Prop}_C \\
\land C &= [P, Q : \text{Prop}_C] \neg_I (\neg_I P \land_I \neg_I Q) \\
\land C &= [P, Q : \text{Prop}_C] \neg_I (\neg_I P \land_I \neg_I Q) \\
\text{Prop}_C \to \text{Prop}_C \to \text{Prop}_C \\
\exists C &= [P : \text{Term} \to \text{Prop}_C] \neg_I \forall_I [x : \text{Term}] \neg_I (P x) \\
\exists C &= [P : \text{Term} \to \text{Prop}_C] \neg_I \forall_I [x : \text{Term}] \neg_I (P x) \\
\exists C &= (\text{Term} \to \text{Prop}_C) \to \text{Prop}_C \\
\exists C &= (\text{Term} \to \text{Prop}_C) \to \text{Prop}_C
\end{align*}
\]

alpha.lf

import class2int;

Theorem alpha : A \lor C \not\equiv C A \equiv \neg_I (\neg_I A \land_I \neg_I \neg_I A)
We need to define an object

\[ \forall C E : (P, Q, R : \text{Prop}) \]
\[ (\text{Prf}(P) \rightarrow \text{Prf}(R)) \rightarrow \]
\[ (\text{Prf}(Q) \rightarrow \text{Prf}(R)) \rightarrow \]
\[ \text{Prf}(P \lor C Q) \rightarrow \text{Prf}(R) \]

but this is not derivable in intuitionistic logic!
We need to prove this rule of deduction is derivable:

\[
\begin{align*}
\vdash & \phi & \vdash & \psi \\
\vdash & \vdash & \vdash & \vdash \\
\chi & & \chi & \vdash \neg (\neg \phi \land \neg \psi) \\
\hline
& & & \chi
\end{align*}
\]
The Problem

We need to prove this rule of deduction is derivable:

\[
\begin{array}{c}
\vdash \phi \\
\vdash \psi \\
\vdash \chi
\end{array}
\quad \vdash \chi
\]

\[
\chi \quad \chi \quad \neg(\neg\phi \land \neg\psi)
\]

...but this is not derivable in intuitionistic logic!
What Would Gödel Do?

The proof of the soundness of $\neg\neg\phi$ uses this lemma:

**Lemma**

For every formula $\phi$, $\phi$ is stable; i.e. $HA \vdash \neg\neg\phi \rightarrow \phi$.

Idea: Define $Prop_C$ to be the kind of all stable formulas.

For now, I used this hack. In class2int.lf, declare the constants:

- $pair : (p : Prop) (\neg\neg I \neg\neg) \rightarrow Prop$,
- $\pi_1 : Prop_C \rightarrow Prop_I$,
- $\pi_2 : (p : Prop_C) \neg\neg I \neg\neg p \rightarrow \pi_1 (p)$,

and the computation rule $\pi_1 (pair p q) = p$:
What Would Gödel Do?

The proof of the soundness of $\neg\neg$ uses this lemma:

\[ \text{Lemma} \]
\[ \forall \phi. (\phi \rightarrow \neg\neg\phi) \]

Idea: Define $\text{Prop} \ C$ to be the kind of all stable formulas.

For now, I used this hack. In `class2int.lf`, declare the constants:

\[ \text{pair} : \left( \text{Prop} I \rightarrow \text{I} \rightarrow \text{Prop} C \right) \]
\[ \pi_1 : \text{Prop} C \rightarrow \text{Prop} I \]
\[ \pi_2 : (p : \text{Prop} C) \rightarrow \text{I} \rightarrow \pi_1(p) \]

and the computation rule
\[ \pi_1(\text{pair} p q) = p : \text{Prop} I \]
What Would Gödel Do?

The proof of the soundness of $\neg\neg$ uses this lemma:

**Lemma**

*For every formula $\phi$, $\neg\neg\phi$ is stable; i.e.*

$$HA \vdash \neg\neg\phi \rightarrow \phi$$
The proof of the soundness of $\neg\neg$ uses this lemma:

**Lemma**

For every formula $\phi$, $\phi^{\neg\neg}$ is stable; i.e.

$$HA \vdash \neg\neg\phi^{\neg\neg} \rightarrow \phi^{\neg\neg}$$

**Idea:** Define $\text{Prop}_C$ to be the kind of all *stable* formulas.
The proof of the soundness of $\neg\neg\phi$ uses this lemma:

**Lemma**

For every formula $\phi$, $\phi \neg\neg$ is stable; i.e.

$$\text{HA} \vdash \neg\neg\phi \rightarrow \phi$$

**Idea**: Define $\text{Prop}_C$ to be the kind of all stable formulas. We would like to write:

$$\text{Prop}_C = \Sigma p : \text{Prop}_I. \neg\neg I \neg\neg I p \rightarrow p$$

but LF does not have $\Sigma$-kinds.
What Would Gödel Do?

The proof of the soundness of $\neg\neg$ uses this lemma:

**Lemma**

For every formula $\phi$, $\phi^{\neg\neg}$ is stable; i.e.

$$HA \vdash \neg\neg\phi^{\neg\neg} \rightarrow \phi^{\neg\neg}$$

**Idea**: Define $\text{Prop}_C$ to be the kind of all *stable* formulas. For now, I used this hack. In class2int.lf, declare the constants:

\[
\begin{align*}
\text{pair} & : (p : \text{Prop}_I)(\neg\neg\neg \neg p \rightarrow p) \rightarrow \text{Prop}_C \\
\pi_1 & : \text{Prop}_C \rightarrow \text{Prop}_I \\
\pi_2 & : (p : \text{Prop}_C)\neg\neg\neg \neg \pi_1(p) \rightarrow \pi_1(p)
\end{align*}
\]

and the computation rule

$$\pi_1(\text{pair } p \ q) = p : \text{Prop}_I$$
This method copes with:
- Friedman’s \( A \)-translation
- The Russell-Prawitz modality

\[
FOL(\neg, \rightarrow, \land, \lor, \forall, \exists) \rightarrow SOL(\forall, \rightarrow)
\]

System T \rightarrow System F

It does not quite work with:
- The Dialectica interpretation

\[
HA \rightarrow \text{System T}
\]
I have shown a method for *proof reuse*:
I have shown a method for *proof reuse*:

*Given two systems declared in a logical framework, to use results proved in one system when working in another.*
I have shown a method for proof reuse:

*Given two systems declared in a logical framework, to use results proved in one system when working in another.*

The method should be very general, applying to translations between first-order systems, type theories, LTTs, . . .
Conclusion

I have shown a method for proof reuse:

Given two systems declared in a logical framework, to use results proved in one system when working in another.

The method should be very general, applying to translations between first-order systems, type theories, LTTs, . . .

Future Work:

- Implement a module mechanism that makes it more convenient.
I have shown a method for *proof reuse*:

*Given two systems declared in a logical framework, to use results proved in one system when working in another.*

The method should be very general, applying to translations between first-order systems, type theories, LTTs, . . .

**Future Work:**

- Implement a module mechanism that makes it more convenient.
- Formalise a piece of pluralist mathematics (e.g. *Metamathematics of First-Order Arithmetic*).
I have shown a method for proof reuse:

*Given two systems declared in a logical framework, to use results proved in one system when working in another.*

The method should be very general, applying to translations between first-order systems, type theories, LTTs, . . .

Future Work:

- Implement a module mechanism that makes it more convenient.
- Formalise a piece of pluralist mathematics (e.g. *Metamathematics of First-Order Arithmetic*).
- Use a logical framework to investigate translations between LTTs.