Proof Reuse in Logical Frameworks

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

TYPES 2010, Warsaw
15 October 2010

Motivation

Motivation

I am a logician. What does a logician do?

Motivation

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

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?

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?

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.

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.
How well do proof assistants offer machine support for this work?

Not Very Well

Not Very Well

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.

Not Very Well

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.

The Problem

There exist *sound translations* between logical systems.

Definition

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

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

such that

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

The Problem

There exist *sound translations* between logical systems.

Definition

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

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

such that

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

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

The Problem

There exist *sound translations* between logical systems.

Definition

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

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

such that

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

(This includes the case $S \hookrightarrow T$ — take Φ to be the identity.)
I want to:

- declare S and T in some logical framework;
- prove α 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

$$0 : \mathbf{Term}, \qquad S : \mathbf{Term} \to \mathbf{Term},$$
$$+ : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term}, \qquad \times : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term} \ ;$$

Arithmetic in LF

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

- a kind **Term**, and constants

$$0 : \mathbf{Term}, \qquad S : \mathbf{Term} \to \mathbf{Term},$$
$$+ : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term}, \qquad \times : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term} \ ;$$

- a kind $\mathbf{Prop}_I$, and constants

$$=_I : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Prop}_I, \qquad \vee_I : \mathbf{Prop}_I \to \mathbf{Prop}_I \to \mathbf{Prop}_I,$$
$$\forall_I : (\mathbf{Term} \to \mathbf{Prop}_I) \to \mathbf{Prop}_I, \ldots \ ;$$

Arithmetic in LF

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

- a kind **Term**, and constants

$$0 : \mathbf{Term}, \qquad S : \mathbf{Term} \to \mathbf{Term},$$
$$+ : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term}, \qquad \times : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term} \ ;$$

- a kind $\mathbf{Prop}_I$, and constants

$$=_I : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Prop}_I, \qquad \vee_I : \mathbf{Prop}_I \to \mathbf{Prop}_I \to \mathbf{Prop}_I,$$
$$\forall_I : (\mathbf{Term} \to \mathbf{Prop}_I) \to \mathbf{Prop}_I, \ldots \ ;$$

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

Arithmetic in LF

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

- a kind **Term**, and constants

$$0 : \mathbf{Term}, \qquad S : \mathbf{Term} \to \mathbf{Term},$$
$$+ : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term}, \qquad \times : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term} \ ;$$

- a kind $\mathbf{Prop}_I$, and constants

$$=_I : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Prop}_I, \qquad \vee_I : \mathbf{Prop}_I \to \mathbf{Prop}_I \to \mathbf{Prop}_I,$$
$$\forall_I : (\mathbf{Term} \to \mathbf{Prop}_I) \to \mathbf{Prop}_I, \ldots \ ;$$

- for every $P : \mathbf{Prop}_I$, a kind $\mathrm{Prf}\,(P)$;
- constants for the rules of deduction:

$$\frac{\begin{array}{ccc} [P] & [Q] & \\ \vdots & \vdots & \\ R & R & P \vee Q \end{array}}{R}$$

Arithmetic in LF

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

- a kind **Term**, and constants

$$0 : \mathbf{Term}, \qquad S : \mathbf{Term} \to \mathbf{Term},$$
$$+ : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term}, \qquad \times : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term} \ ;$$

- a kind $\mathbf{Prop}_I$, and constants

$$=_I : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Prop}_I, \qquad \vee_I : \mathbf{Prop}_I \to \mathbf{Prop}_I \to \mathbf{Prop}_I,$$
$$\forall_I : (\mathbf{Term} \to \mathbf{Prop}_I) \to \mathbf{Prop}_I, \ldots \ ;$$

- for every $P : \mathbf{Prop}_I$, a kind $\mathrm{Prf}\,(P)$;
- constants for the rules of deduction:

$$\begin{aligned} \vee E \ : \ & (P, Q, R : \mathbf{Prop}_I) \\ & (\mathrm{Prf}\,(P) \to \mathrm{Prf}\,(R)) \to \\ & (\mathrm{Prf}\,(Q) \to \mathrm{Prf}\,(R)) \to \\ & \mathrm{Prf}\,(P \vee_I Q) \to \mathrm{Prf}\,(R) \end{aligned}$$

Arithmetic in LF

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

- a kind **Term**, and constants

$$0 : \mathbf{Term}, \qquad S : \mathbf{Term} \to \mathbf{Term},$$
$$+ : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term}, \qquad \times : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term} \ ;$$

- a kind $\mathbf{Prop}_I$, and constants

$$=_I : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Prop}_I, \qquad \vee_I : \mathbf{Prop}_I \to \mathbf{Prop}_I \to \mathbf{Prop}_I,$$
$$\forall_I : (\mathbf{Term} \to \mathbf{Prop}_I) \to \mathbf{Prop}_I, \ldots \ ;$$

- for every $P : \mathbf{Prop}_I$, a kind $\mathrm{Prf}\,(P)$;
- constants for the rules of deduction:

$$\begin{aligned} \vee E \ : \ & (P, Q, R : \mathbf{Prop}_I) \\ & (\mathrm{Prf}\,(P) \to \mathrm{Prf}\,(R)) \to \\ & (\mathrm{Prf}\,(Q) \to \mathrm{Prf}\,(R)) \to \\ & \mathrm{Prf}\,(P \vee_I Q) \to \mathrm{Prf}\,(R) \end{aligned}$$

Arithmetic in LF

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

- a kind **Term**, and constants

$$0 : \mathbf{Term}, \qquad S : \mathbf{Term} \to \mathbf{Term},$$
$$+ : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term}, \qquad \times : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term} \ ;$$

- a kind $\mathbf{Prop}_C$, and constants

$$=_C : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Prop}_C, \qquad \vee_C : \mathbf{Prop}_C \to \mathbf{Prop}_C \to \mathbf{Prop}_C$$
$$\forall_C : (\mathbf{Term} \to \mathbf{Prop}_C) \to \mathbf{Prop}_C, \ldots \ ;$$

- for every $P : \mathbf{Prop}_C$, a kind $\mathrm{Prf}\,(P)$;
- constants for the rules of deduction:

$$\begin{aligned} \vee E \ : \ & (P, Q, R : \mathbf{Prop}_C) \\ & (\mathrm{Prf}\,(P) \to \mathrm{Prf}\,(R)) \to \\ & (\mathrm{Prf}\,(Q) \to \mathrm{Prf}\,(R)) \to \\ & \mathrm{Prf}\,(P \vee_I Q) \to \mathrm{Prf}\,(R) \end{aligned}$$

Arithmetic in LF

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

- a kind **Term**, and constants

$$0 : \mathbf{Term}, \qquad S : \mathbf{Term} \to \mathbf{Term},$$
$$+ : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term}, \qquad \times : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Term} \ ;$$

- a kind $\mathbf{Prop}_C$, and constants

$$=_C : \mathbf{Term} \to \mathbf{Term} \to \mathbf{Prop}_C, \qquad \vee_C : \mathbf{Prop}_C \to \mathbf{Prop}_C \to \mathbf{Prop}_C$$
$$\forall_C : (\mathbf{Term} \to \mathbf{Prop}_C) \to \mathbf{Prop}_C, \ldots \ ;$$

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

$$EM : (P : \mathbf{Prop}_C)\mathrm{Prf}\,(P \vee_C \neg_C P)$$

Double Negation Translation

Let *PA* be Peano Arithmetic, and *HA* be Heyting Arithmetic.

Double Negation Translation

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

$$
\begin{aligned}
(s = t)^{\neg\neg} &\equiv \neg\neg(s = t) \\
(\neg\phi)^{\neg\neg} &\equiv \neg\phi^{\neg\neg} \\
(\phi \wedge \psi)^{\neg\neg} &\equiv \phi^{\neg\neg} \wedge \psi^{\neg\neg} \\
(\phi \vee \psi)^{\neg\neg} &\equiv \neg(\neg\phi^{\neg\neg} \wedge \neg\psi^{\neg\neg}) \\
(\phi \to \psi)^{\neg\neg} &\equiv \phi^{\neg\neg} \to \psi^{\neg\neg} \\
(\forall x\phi)^{\neg\neg} &\equiv \forall x\phi^{\neg\neg} \\
(\exists x\phi)^{\neg\neg} &\equiv \neg\forall x\neg\phi^{\neg\neg}
\end{aligned}
$$

Double Negation Translation

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

$$\begin{aligned}
(s = t)^{\neg\neg} &\equiv \neg\neg(s = t) \\
(\neg\phi)^{\neg\neg} &\equiv \neg\phi^{\neg\neg} \\
(\phi \wedge \psi)^{\neg\neg} &\equiv \phi^{\neg\neg} \wedge \psi^{\neg\neg} \\
(\phi \vee \psi)^{\neg\neg} &\equiv \neg(\neg\phi^{\neg\neg} \wedge \neg\psi^{\neg\neg}) \\
(\phi \to \psi)^{\neg\neg} &\equiv \phi^{\neg\neg} \to \psi^{\neg\neg} \\
(\forall x\phi)^{\neg\neg} &\equiv \forall x\phi^{\neg\neg} \\
(\exists x\phi)^{\neg\neg} &\equiv \neg\forall x\neg\phi^{\neg\neg}
\end{aligned}$$

Theorem (Gödel, 1933)

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

Double Negation Translation

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

$$\begin{aligned}
(s = t)^{\neg\neg} &\equiv \neg\neg(s = t) \\
(\neg\phi)^{\neg\neg} &\equiv \neg\phi^{\neg\neg} \\
(\phi \wedge \psi)^{\neg\neg} &\equiv \phi^{\neg\neg} \wedge \psi^{\neg\neg} \\
(\phi \vee \psi)^{\neg\neg} &\equiv \neg(\neg\phi^{\neg\neg} \wedge \neg\psi^{\neg\neg}) \\
(\phi \to \psi)^{\neg\neg} &\equiv \phi^{\neg\neg} \to \psi^{\neg\neg} \\
(\forall x\phi)^{\neg\neg} &\equiv \forall x\phi^{\neg\neg} \\
(\exists x\phi)^{\neg\neg} &\equiv \neg\forall x\neg\phi^{\neg\neg}
\end{aligned}$$

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.lf

orC : ...
```

—

Declare the classical system.

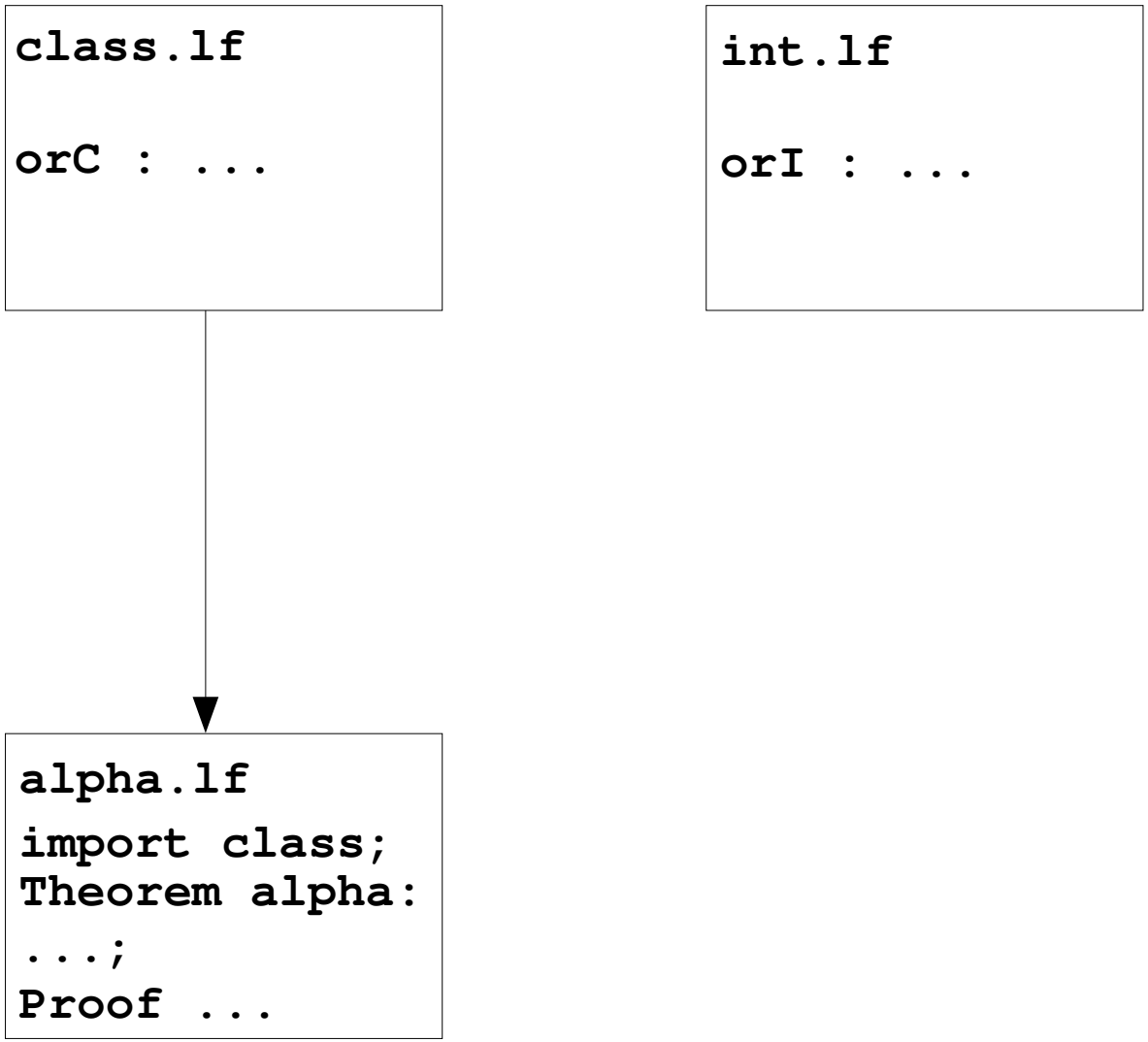
How It Works

```
class.lf

orC : ...
```

```
int.lf

orI : ...
```

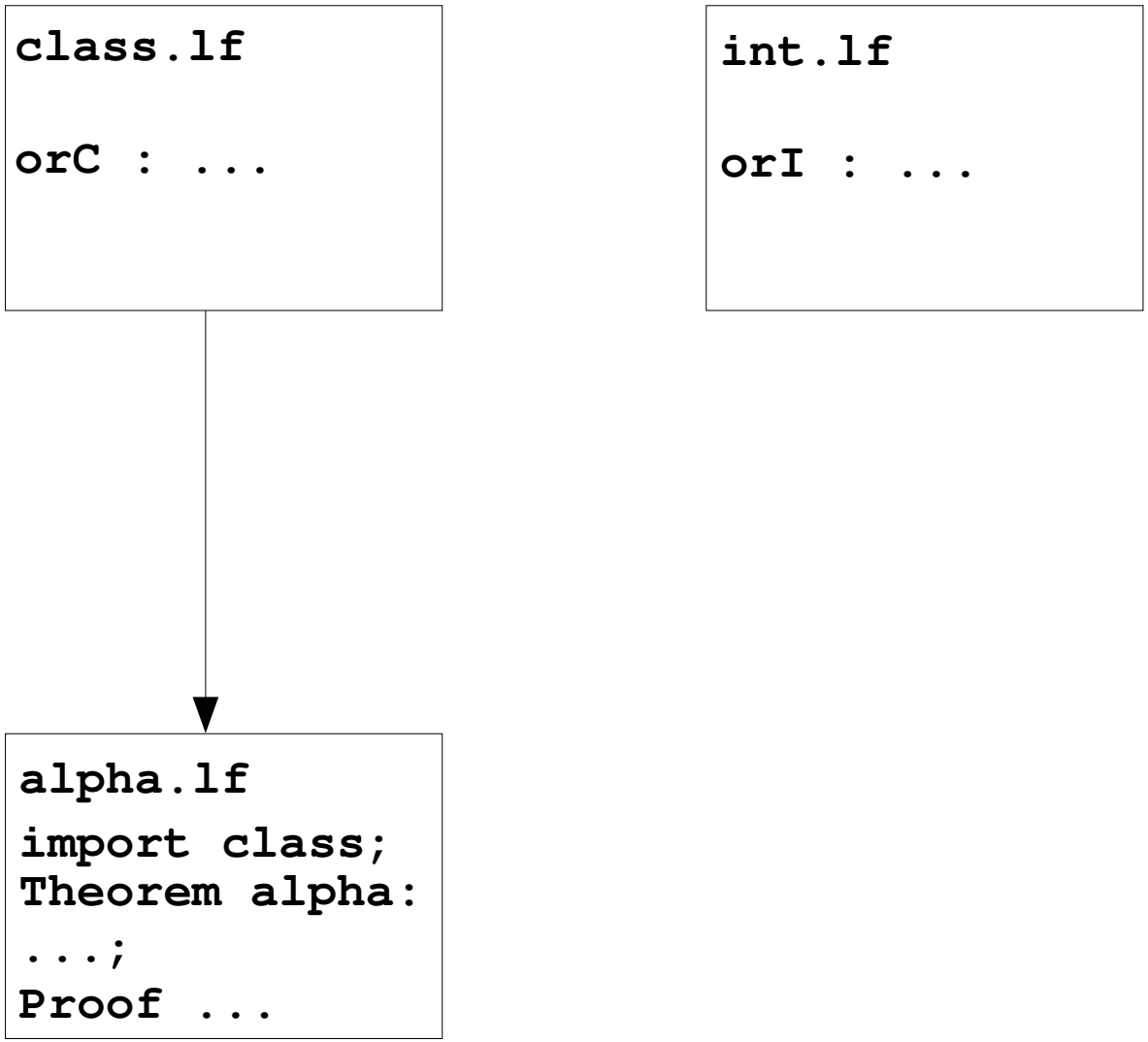
Declare the intuitionistic system.

How It Works



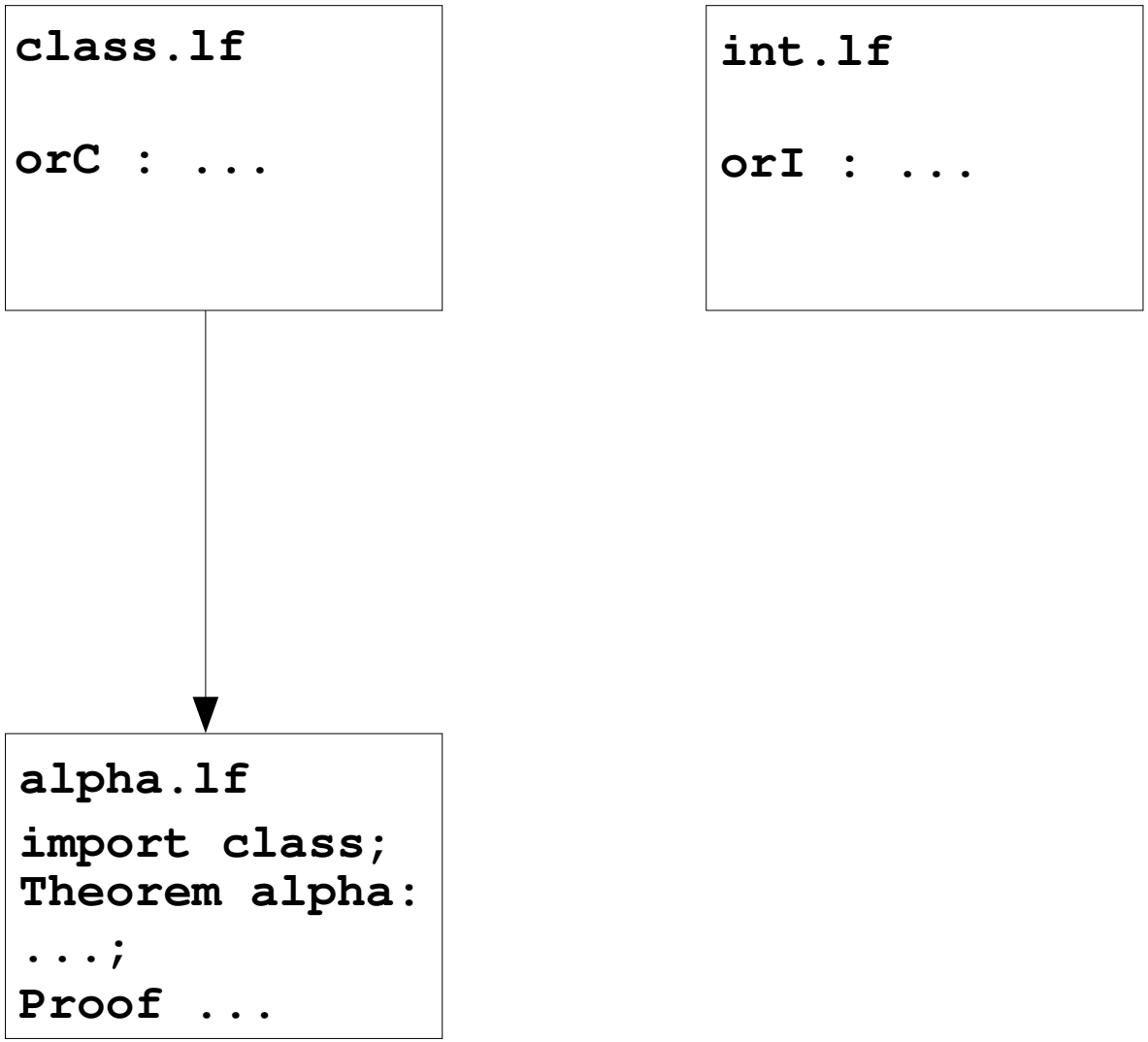
Prove α in the classical system.

How It Works



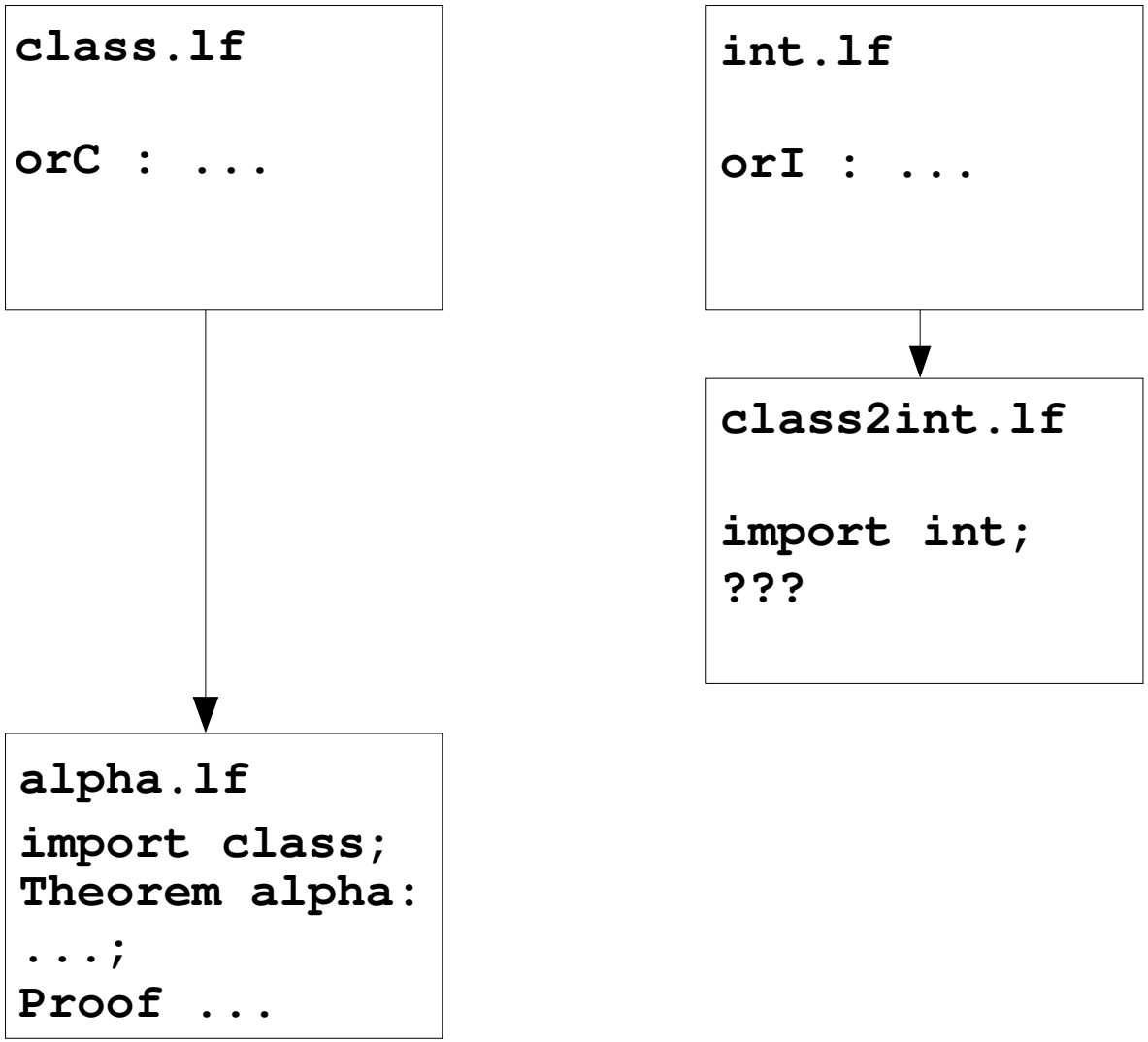
I don't want to do much work now,

How It Works



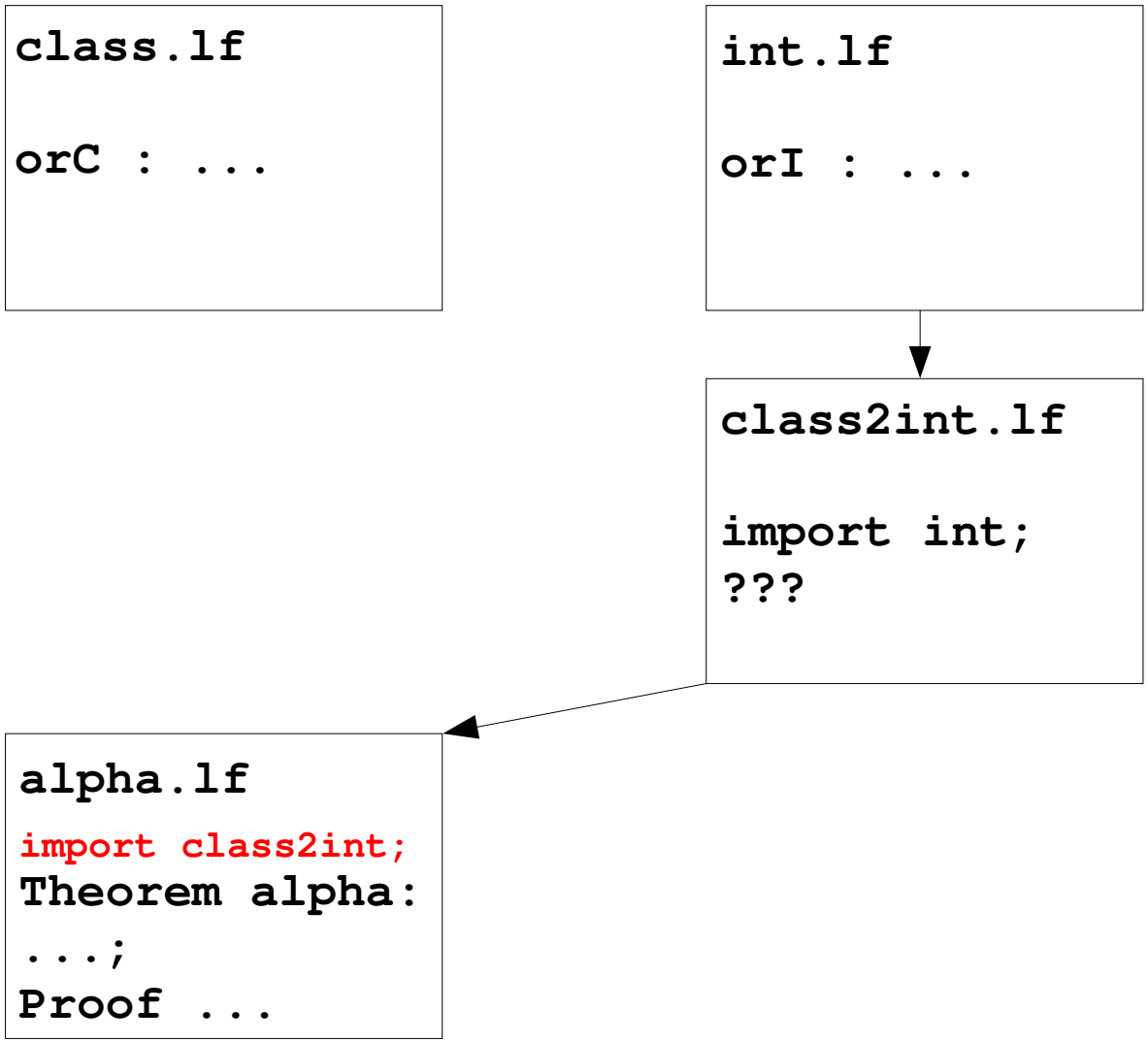
I don't want to do much work now, because I'm lazy.

How It Works



Write a module such that . . .

How It Works



. . . we have a proof of $\alpha^{\neg\neg}$ in the intuitionistic system.

First Attempt

`class.lf`

$$\vdots$$

$$\neg_C \quad : \quad \mathbf{Prop}_C \to \mathbf{Prop}_C$$

$$\vee_C \quad : \quad \mathbf{Prop}_C \to \mathbf{Prop}_C \to \mathbf{Prop}_C$$

$$\exists_C \quad : \quad (\mathbf{Term} \to \mathbf{Prop}_C) \to \mathbf{Prop}_C$$

$$\vdots$$

`alpha.lf`

```
import class;
```

Theorem alpha : $A \vee_C \neg_C A$

First Attempt

`class2int.lf`

$$
\begin{array}{rcl}
\mathbf{Prop}_C & = & \mathbf{Prop}_I \\
 & \vdots & \\
\neg_C & = & \neg_I \\
 & : & \mathbf{Prop}_C \to \mathbf{Prop}_C \\
\vee_C & = & [P, Q : \mathbf{Prop}_C] \neg_I (\neg_I P \wedge_I \neg_I Q) \\
 & : & \mathbf{Prop}_C \to \mathbf{Prop}_C \to \mathbf{Prop}_C \\
\exists_C & = & [P : \mathbf{Term} \to \mathbf{Prop}_C] \neg_I \forall_I [x : \mathbf{Term}] \neg_I (Px) \\
 & : & (\mathbf{Term} \to \mathbf{Prop}_C) \to \mathbf{Prop}_C \\
 & \vdots &
\end{array}
$$

`alpha.lf`

```
import class2int;
```

Theorem alpha : $A \vee_C \neg_C A$

First Attempt

`class2int.lf`

$$
\begin{array}{rcl}
\mathbf{Prop}_C & = & \mathbf{Prop}_I \\
 & \vdots & \\
\neg_C & = & \neg_I \\
 & : & \mathbf{Prop}_C \to \mathbf{Prop}_C \\
\vee_C & = & [P, Q : \mathbf{Prop}_C] \neg_I (\neg_I P \wedge_I \neg_I Q) \\
 & : & \mathbf{Prop}_C \to \mathbf{Prop}_C \to \mathbf{Prop}_C \\
\exists_C & = & [P : \mathbf{Term} \to \mathbf{Prop}_C] \neg_I \forall_I [x : \mathbf{Term}] \neg_I (Px) \\
 & : & (\mathbf{Term} \to \mathbf{Prop}_C) \to \mathbf{Prop}_C \\
 & \vdots &
\end{array}
$$

`alpha.lf`

```
import class2int;
```

Theorem alpha : $A \vee_C \neg_C A \quad \simeq \quad \neg_I(\neg_I A \wedge_I \neg_I \neg_I A)$

The Problem

We need to define an object

$$\begin{aligned} \vee_C E \quad : \quad & (P, Q, R : \mathbf{Prop}) \\ & (\mathrm{Prf}\,(P) \to \mathrm{Prf}\,(R)) \to \\ & (\mathrm{Prf}\,(Q) \to \mathrm{Prf}\,(R)) \to \\ & \mathrm{Prf}\,(P \vee_C Q) \to \mathrm{Prf}\,(R) \end{aligned}$$

The Problem

We need to prove this rule of deduction is derivable:

$$\dfrac{\begin{array}{ccc}[\phi] & [\psi] & \\ \vdots & \vdots & \\ \chi & \chi & \neg(\neg\phi \wedge \neg\psi)\end{array}}{\chi}$$

The Problem

We need to prove this rule of deduction is derivable:

$$\dfrac{\begin{array}{ccc}[\phi] & [\psi] & \\ \vdots & \vdots & \\ \chi & \chi & \neg(\neg\phi \wedge \neg\psi)\end{array}}{\chi}$$

. . . but this is not derivable in intuitionistic logic!

What Would Gödel Do?

What Would Gödel Do?

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

What Would Gödel Do?

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

Lemma

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

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

What Would Gödel Do?

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

Lemma

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

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

Idea: Define $\mathbf{Prop}_C$ to be the kind of all *stable* formulas.

What Would Gödel Do?

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

Lemma

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

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

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

$$\mathbf{Prop}_C = \Sigma p : \mathbf{Prop}_I . \neg_I \neg_I p \to p$$

but LF does not have Σ-kinds.

What Would Gödel Do?

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

Lemma

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

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

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

$$\begin{aligned} \mathit{pair} &: (p : \mathbf{Prop}_I)(\neg_I\neg_I p \to p) \to \mathbf{Prop}_C \\ \pi_1 &: \mathbf{Prop}_C \to \mathbf{Prop}_I \\ \pi_2 &: (p : \mathbf{Prop}_C)\neg_I\neg_I \pi_1(p) \to \pi_1(p) \end{aligned}$$

and the computation rule

$$\pi_1(\mathit{pair}\ p\ q) = p : \mathbf{Prop}_I$$

Other Translations and Applications

This method copes with:

- Friedman's A-translation
- The Russell-Prawitz modality

$$\begin{aligned} FOL(\neg, \to, \wedge, \vee, \forall, \exists) &\to SOL(\forall, \to) \\ \text{System T} &\to \text{System F} \end{aligned}$$

It does not quite work with:

- The Dialectica interpretation

$$HA \to \text{System T}$$

Conclusion

I have shown a method for *proof reuse*:

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.*

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, ...

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.

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.
- Formalise a piece of pluralist mathematics (e.g. *Metamathematics of First-Order Arithmetic*).

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.
- Formalise a piece of pluralist mathematics (e.g. *Metamathematics of First-Order Arithmetic*).
- Use a logical framework to investigate translations between LTTs.