Refinement Types as Proof Irrelevance

William Lovas
with Frank Pfenning

Overview

- *Refinement types* sharpen existing type systems without complicating their metatheory
- *Subset interpretation* soundly and completely eliminates them
- Shows the expressive power of refinements

Overview

- *Refinement types* sharpen existing type systems without complicating their metatheory
- *Subset interpretation* soundly and completely eliminates them
- Shows the expressive power of refinements
  - Translation is quite complicated!

Refinement types

- New layer of "sorts" above usual "type" layer
  - subsorting, intersection sorts
  - every *sort* refines a *type*
- Refinement restriction
  - only sort-check well-typed terms
  - … and only at sorts refining their type
- Goal: *more precisely* classify *well-typed* terms

Example: natural numbers

nat : **type**.
z : nat.
s : nat → nat.

Example: natural numbers

```
nat : type.
z : nat.
s : nat → nat.

% double relation: [double N N2] iff N2 = N * 2
double : nat → nat → type.
dbl/z : double z z.
dbl/s : ΠN:nat. ΠN2:nat. double N N2
        → double (s N) (s (s N2)).
```

- LF methodology: *judgements as types*

Example: natural numbers

$\text{even} \sqsubset \text{nat}.\ \ \text{odd} \sqsubset \text{nat}.$

$\text{z} :: \text{even}.$

$\text{s} :: (\text{even} \to \text{odd}) \ \wedge \ (\text{odd} \to \text{even}).$

- LFR methodology: *properties as sorts*

Example: natural numbers

even ⊏ nat. odd ⊏ nat.

z :: even.

s :: (even → odd) ∧ (odd → even).

% double: just like double, but second arg even*

double* ⊏ double :: ⊤ → even → **sort**.

dbl/z :: double* z z.

dbl/s :: Π*N*::⊤. Π*N2*::even. double* *N* *N2*

→ double* (s *N*) (s (s *N2*)).

- LFR methodology: *properties as sorts*

Example: natural numbers

```
even ⊏ nat.  odd ⊏ nat.
z :: even.
s :: (even → odd)  ∧  (odd → even).

% double*: just like double, but second arg even
double* ⊏ double :: ⊤ → even → sort.
dbl/z :: double* z z.
dbl/s :: ΠN::⊤. ΠN2::even. double* N N2
        → double* (s N) (s (s N2)).
```

- LFR methodology: *properties as sorts*

(Aside: other examples)

- More precise types $\Rightarrow$ more precise judgements!
- Statically capture many interesting properties:
  - even and odd natural numbers
  - big-step evaluation returns a value
  - uniform yet stratified encoding of PTSes
  - normal natural deductions / cut-free sequent derivations
  - hereditary Harrop formulas for logic programming
  - ...

Example: sorts as predicates

```
nat : type.
z : nat.
s : nat → nat.
```

Example: sorts as predicates

nat : **type**.
z : nat.
s : nat → nat.

even ⊏ nat.
odd ⊏ nat.
z :: even.
s :: (even → odd)
  ∧ (odd → even).

Example: sorts as predicates

nat : **type**.
z : nat.
s : nat → nat.

even ⊏ nat.
odd ⊏ nat.
z :: even.
s :: (even → odd)
  ∧ (odd → even).

even : nat → **type**.
odd : nat → **type**.
pf-z : even z.
pf-s_1 : Πx:nat. even x → odd (s x)
pf-s_2 : Πx:nat. odd x → even (s x).

Example: sorts as predicates

nat : typ
z : nat.
s : nat →

- Translation follows this idea:
  - **refinements** become *predicates*
  - **sort declarations** become *proof constructors*

even ⊏ nat.
odd ⊏ nat.
z :: even.
s :: (even → odd)
  ∧ (odd → even).

even : nat → **type**.
odd : nat → **type**.
pf-z : even z.
$\text{pf-s}_1 : \Pi x{:}\text{nat. even } x \to \text{odd (s } x)$
$\text{pf-s}_2 : \Pi x{:}\text{nat. odd } x \to \text{even (s } x).$

Outline

- ✓ Overview
- ✓ Example: even and odd natural numbers
- Theorems: soundness, completeness, adequacy
- Technical detail: role of proof irrelevance
- Conclusions

Example: sorts as predicates

nat : **type**.
z : nat.
s : nat → nat.

even ⊏ nat.
odd ⊏ nat.
z :: even.
s :: (even → odd)
  ∧ (odd → even).

even : nat → **type**.
odd : nat → **type**.
pf-z : even z.
$\text{pf-s}_1 : \Pi x{:}\text{nat}.\ \text{even}\ x \to \text{odd}\ (\text{s}\ x)$
$\text{pf-s}_2 : \Pi x{:}\text{nat}.\ \text{odd}\ x \to \text{even}\ (\text{s}\ x).$

Example: sorts as predicates

nat : **type**.
z : nat.
s : nat → nat.

> N :: even iff P : even N
> (for some P)

even ⊏ nat.
odd ⊏ nat.
z :: even.
s :: (even → odd)
∧ (odd → even).

even : nat → **type**.
odd : nat → **type**.
pf-z : even z.
pf-s_1 : Πx:nat. even x → odd (s x)
pf-s_2 : Πx:nat. odd x → even (s x).

Soundness & Completeness

- Translation turns:
  - well-formed sorts *S* into predicates *S'*(-)
  - derivations of $N :: S$ into proofs *N'*

Soundness & Completeness

- Translation turns:
  - well-formed sorts S into predicates $S'(-)$
  - derivations of $N :: S$ into proofs N'

Theorem: if $S \rightsquigarrow S'$ and $N :: S \rightsquigarrow N'$, then $N' : S'(N)$

Theorem: if $S \rightsquigarrow S'$ and $N' : S'(N)$, then $N :: S$

Soundness & Completeness

- Translation turns:
  - well-formed sorts S into predicates $S'(\text{-})$
  - derivations of $N :: S$ into proofs N'

Theorem: if $S \rightsquigarrow S'$ and $N :: S \rightsquigarrow N'$, then $N' : S'(N)$

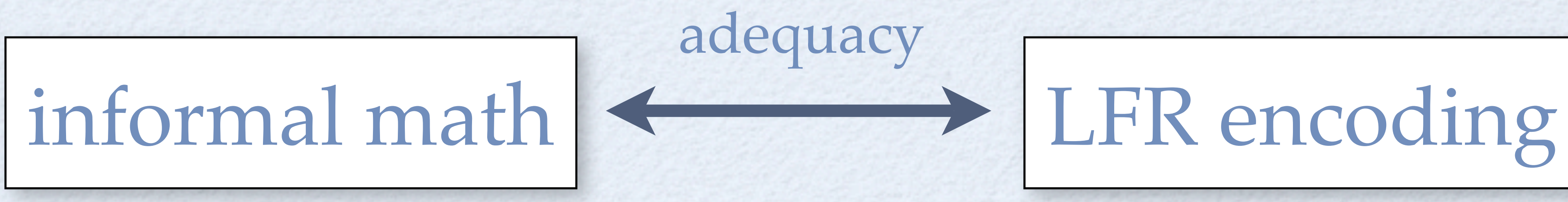
Theorem: if $S \rightsquigarrow S'$ and $N' : S'(N)$, then $N :: S$

Corollary: Preservation of Adequacy

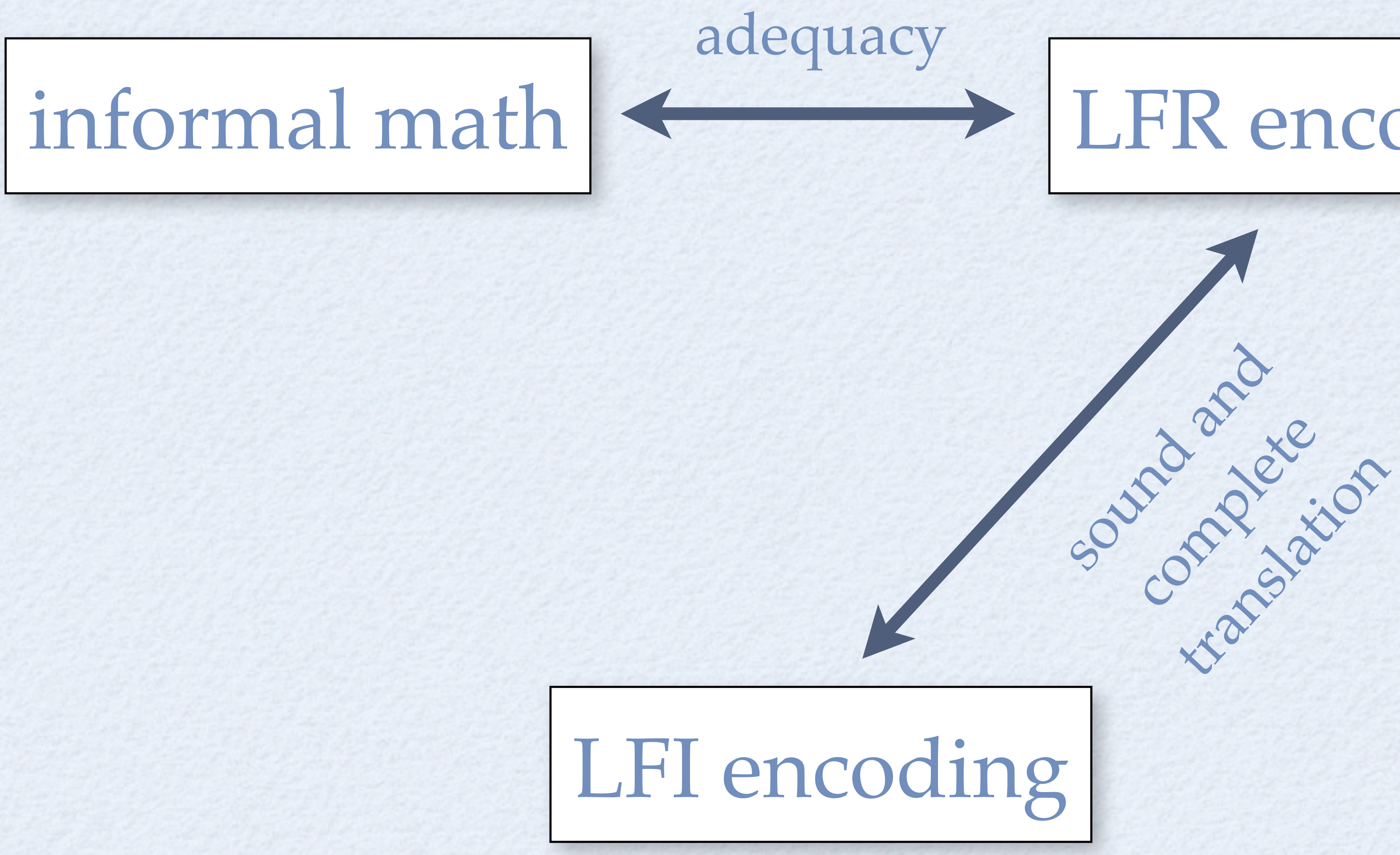
Adequacy

- LF enjoys a well-developed theory of adequate representations
    - **Adequacy**: compositional bijection between informal entities and canonical (β-normal η-long) terms

Adequacy

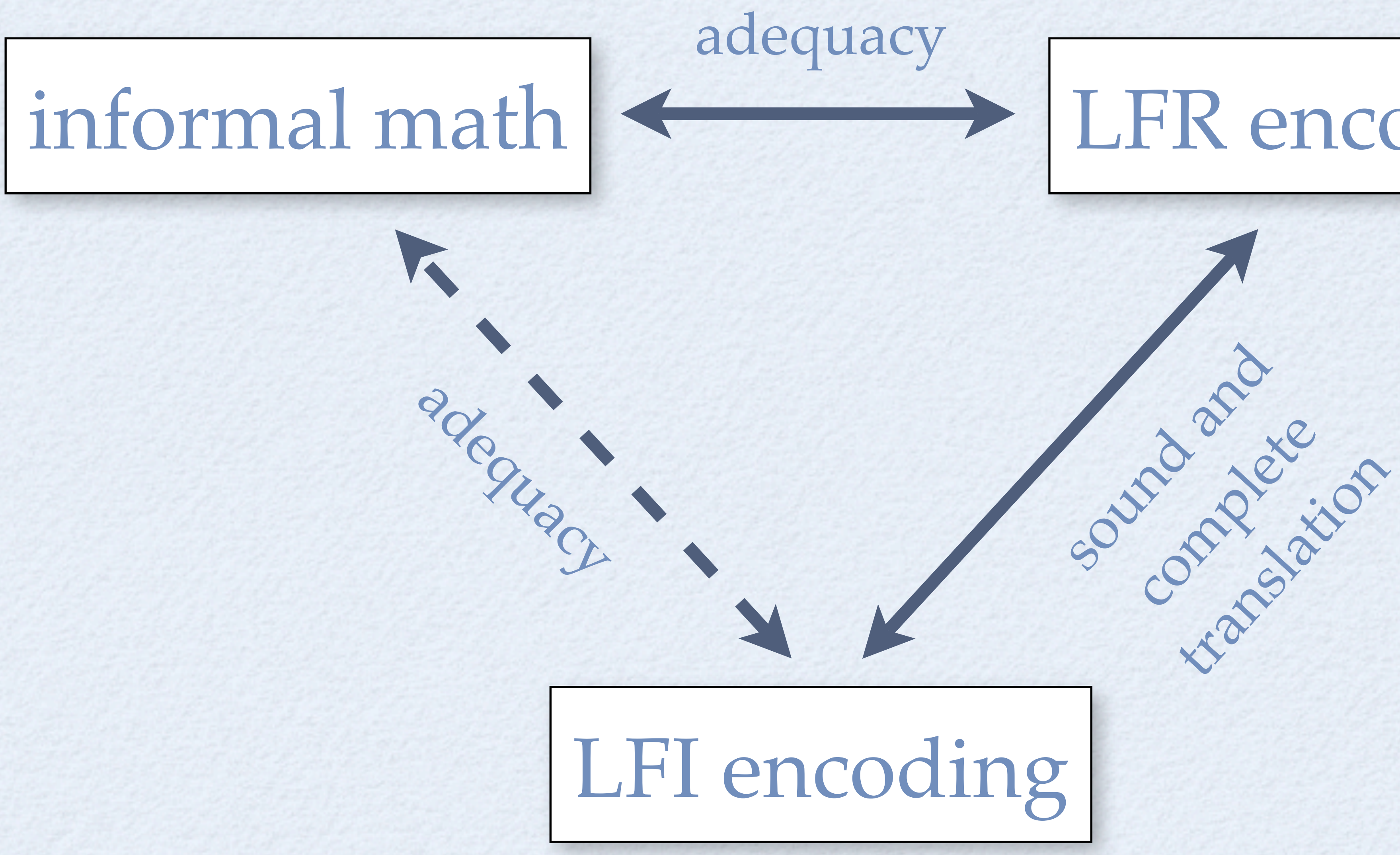


Adequacy



informal math
adequacy
LFR encoding
sound and
complete
translation
LFI encoding

Adequacy


adequacy
informal math
LFR encoding
adequacy
sound and complete translation
LFI encoding

Key lemmas

- Translation “derivation-directed”:

Key lemmas

- Translation "derivation-directed":

Lemma (Erasure)**:** If $N :: S \rightsquigarrow N'$, then $N :: S$

Lemma (Reconstruction)**:** If $N :: S$, then $N :: S \rightsquigarrow N'$

Key lemmas

- Translation "derivation-directed":

Lemma (Erasure)**:** If $N :: S \rightsquigarrow N'$, then $N :: S$

Lemma (Reconstruction)**:** If $N :: S$, then $N :: S \rightsquigarrow N'$

Lemma (Compositionality)**:** Let σ denote $[M/x]$.
If $S \rightsquigarrow S'$ and $\sigma S \rightsquigarrow S''$, then $\sigma S'(N) = S''(\sigma N)$

Outline

- ✓ Overview
- ✓ Example: even and odd natural numbers
- ✓ Theorems: soundness, completeness, adequacy
- Technical detail: role of proof irrelevance
- Conclusions

Translating judgements

- Simple sorts (like even and odd) are easy
- *Dependent* sorts (like double*) trickier!
- Crucial difference: *inhabitation* vs. *formation*

Translating judgements

- Simple sorts (like even and odd) are easy
- *Dependent* sorts (like double*) trickier!
- Crucial difference: *inhabitation* vs. *formation*
  - even and odd: *subsets* of nat

Translating judgements

- Simple sorts (like even and odd) are easy
- *Dependent* sorts (like double*) trickier!
- Crucial difference: *inhabitation* vs. *formation*
  - even and odd: *subsets* of nat
  - double*: same as double, but with *invariant*

Translating judgements

```
% double*: just like double, but second arg even
double* ⊏ double :: ⊤ → even → sort.
```

- Dependent sort families translate three ways:
  - **formation family:** inhabited when the sort is well-formed at all

Translating judgements

% double: just like double, but second arg even*
double* ⊏ double :: ⊤ → even → **sort**.

- Dependent sort families translate three ways:
  - **formation family:** inhabited when the sort is well-formed at all

% fm-double: formation family*
fm-double* : nat → nat → **type**.

Translating judgements

```
% double*: just like double, but second arg even
double* ⊏ double :: ⊤ → even → sort.
```

- Dependent sort families translate three ways:
  - **formation proof constructors:** ways of proving a sort family well-formed

Translating judgements

```
% double*: just like double, but second arg even
double* ⊏ double :: ⊤ → even → sort.
```

- Dependent sort families translate three ways:
  - **formation proof constructors:** ways of proving a sort family well-formed

```
% fm-double*/i: formation proof constructor
fm-double*/i : Πx:nat. Πy:nat. even y → fm-double* x y.
```

Translating judgements

```
% double*: just like double, but second arg even
double* ⊏ double :: ⊤ → even → sort.
```

- Dependent sort families translate three ways:
  - **formation proof constructors:** ways of proving a sort family well-formed

```
% fm-double*/i: formation proof constructor
fm-double*/i : Πx:nat. Πy:nat. even y → fm-double* x y.
```

Translating judgements

```
% double*: just like double, but second arg even
double* ⊏ double :: ⊤ → even → sort.
```

- Dependent sort families translate three ways:
  - **predicate family:** holds of terms that have the sort (provided the sort is well-formed)

Translating judgements

% double: just like double, but second arg even*
double* $\sqsubset$ double :: $\top \to$ even $\to$ **sort**.

- Dependent sort families translate three ways:
  - **predicate family:** holds of terms that have the sort (provided the sort is well-formed)

% double: predicate family*
double* : Πx:nat. Πy:nat. fm-double* x $y \to$
double x $y \to$ **type**.

Translating judgements

```
% double*: just like double, but second arg even
double* ⊏ double :: ⊤ → even → sort.
```

- Dependent sort families translate three ways:
  - **predicate family:** holds of terms that have the sort (provided the sort is well-formed)

```
% double*: predicate family
double* : Πx:nat. Πy:nat. fm-double* x y →
          double x y → type.
```

Translating double

% double: just like double, but second arg even*

double* ⊏ double :: ⊤ → even → **sort**.

⇝

% fm-double: formation family*

fm-double* : nat → nat → **type**.

% fm-double/i: formation proof constructor*

fm-double*/i : Π*x*:nat. Π*y*:nat. even *y* → fm-double* *x y*.

% double: predicate family*

double* : Π*x*:nat. Π*y*:nat. fm-double* *x y* →
double *x y* → **type**.

What about irrelevance?

What about irrelevance?

- Suppose double* M N could be well-formed for two reasons. Then:

What about irrelevance?

- Suppose double* M N could be well-formed for two reasons. Then:
  - two proofs of fm-double* M N, say P_1 and P_2

What about irrelevance?

- Suppose double* M N could be well-formed for two reasons. Then:
  - two proofs of fm-double* M N, say P_1 and P_2
  - two different translations of double* M N: double* M N P_1 **(-)** and double* M N P_2 **(-)**

What about irrelevance?

- Suppose double* M N could be well-formed for two reasons. Then:
  - two proofs of fm-double* M N, say P_1 and P_2
  - two different translations of double* M N: double* M N P_1 **(-)** and double* M N P_2 **(-)**
  - if D :: double* M N and $D \rightsquigarrow D'$, soundness requires D' : double* M N P_i **(**D**)**

Seeing double twice

$\text{zero} \sqsubset \text{nat}.$

$\text{z} :: \text{even} \ \wedge \ \text{zero}.$

$$\text{double*} \sqsubset \text{double} :: (\top \to \text{even} \to \textbf{sort}) \\ \wedge \ (\text{zero} \to \text{zero} \to \textbf{sort}).$$

Seeing double twice

zero ⊏ nat.

z :: even ∧ zero.

double* ⊏ double :: (⊤ → even → **sort**)
∧ (zero → zero → **sort**).

zero : nat → **type**.

$p\text{-}z_1$: even z. $p\text{-}z_2$: zero z.

Seeing double twice

zero ⊏ nat.

z :: even ∧ zero.

double* ⊏ double :: (⊤ → even → **sort**)

∧ (zero → zero → **sort**).

zero : nat → **type**.

p-z_1 : even z. p-z_2 : zero z.

Seeing double twice

zero ⊏ nat.

z :: even ∧ zero.

double* ⊏ double :: (⊤ → even → **sort**)
∧ (zero → zero → **sort**).

zero : nat → **type**.

p-z_1 : even z. p-z_2 : zero z.

% fm-double/i: formation proof constructor*

fm-double*/i_1 : Π*x*:nat. Π*y*:nat. even *y* → fm-double* *x* *y*.

fm-double*/i_2 : Π*x*:nat. Π*y*:nat. zero *x* → zero *y* →
fm-double* *x* *y*.

Seeing double twice

zero ⊏ nat.

z :: even ∧ zero.

double* z z ?

double* ⊏ double :: (⊤ → even → **sort**)
∧ (zero → zero → **sort**).

zero : nat → **type**.

p-z_1 : even z. p-z_2 : zero z.

% fm-double/i: formation proof constructor*

fm-double*/i_1 : Π*x*:nat. Π*y*:nat. even *y* → fm-double* *x y*.

fm-double*/i_2 : Π*x*:nat. Π*y*:nat. zero *x* → zero *y* →
fm-double* *x y*.

Proof irrelevance

- Hides the identity of certain terms
    - Proofs of propositions: identity is immaterial
    - Equivalence of terms ignores irrelevant items
- Particularly interesting in a *dependent* setting: term equalities affect type equalities!

Translating double

% double: just like double, but second arg even*

double* $\sqsubset$ double :: $\top \to$ even $\to$ **sort**.

$\rightsquigarrow$

% fm-double: formation family*

...

% fm-double/i_k: formation proof constructors*

...

% double: predicate family*

double* : Πx:nat. Πy:nat. fm-double* $x\ y \div\!\!>$
double $x\ y \to$ **type**.

Translating double

% double: just like double, but second arg even*

double* $\sqsubset$ double :: $\top \to$ even $\to$ **sort**.

$\rightsquigarrow$

% fm-double: formation family*

...

% fm-double/i_k: formation proof constructors*

...

% double: predicate family*

double* : Πx:nat. Πy:nat. fm-double* x y $\div\!\!\to$
double x y $\to$ **type**.

Conclusions

- Possible to translate away refinements, even in the rich, dependent setting of LF
- Uniform translation is quite complicated, proofs of theorems intricate
- Perhaps it's better to keep refinements primitive

secret/backup slides

Related work

- Matthieu Sozeau
  - subset types for CIC
  - refinement types for PVS

Other gotchas

- Intersections:

$$c :: S_1 \wedge S_2.$$

Other gotchas

- Intersections:

c :: $S_1 \wedge S_2$.

pf-c_1 : S_1'(c).
pf-c_2 : S_2'(c).

Other gotchas

- Intersections:

$c :: S_1 \wedge S_2.$

$\text{pf-}c_1 : S_1'(c).$
$\text{pf-}c_2 : S_2'(c).$

vs.

$\text{pf-}c : S_1'(c) \times S_2'(c).$

Other gotchas

- Intersections:

$c :: S_1 \wedge S_2.$

$\text{pf-}c_1 : S_1'(c).$
$\text{pf-}c_2 : S_2'(c).$

vs.

$\text{pf-}c : S_1'(c) \times S_2'(c).$

- Subsorting must generate proof coercions

LF: a logical framework

- Harper, Honsell, and Plotkin, 1987, 1993
- Dependently-typed lambda-calculus
- Encode deductive systems and metatheory, uniformly and machine-checkably
  - *e.g.* a programming language and its type safety theorem
  - *e.g.* a logic and its cut elimination theorem

Judgements as types

On paper	In LF
Syntax ▸ e ::= … τ ::= …	Simple type ▸ exp : **type**. tp : **type**.
Judgement ▸ $\Gamma \vdash e : \tau$	Type family ▸ of : exp → tp → **type**.
Derivation ▸ $\mathcal{D} :: \Gamma \vdash e : \tau$	Well-typed term ▸ M : of E T
Proof checking	Type checking