

Refinement Types for LF

William Lovas

Automating reason



We can look at the current field of problem solving by computers as a series of ideas about how to present a problem. If a problem can be cast into one of these representations in a natural way, then it is possible to manipulate it and stand some chance of solving it.

(Allen Newell, 1965)

- Better representations \Rightarrow better automation

Thesis

- Refinement types are a useful and practical addition to the logical framework LF.

Thesis

- Refinement types are a useful and practical addition to the logical framework LF.
 - ▶ in particular, lead to better representations

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

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
- Guiding principle: “*judgements as types*”

Judgements as types

On paper	In LF
Syntax ▶ $e ::= \dots \quad \tau ::= \dots$	Simple type ▶ $\text{exp} : \mathbf{type}. \quad \text{tp} : \mathbf{type}.$
Judgement ▶ $\Gamma \vdash e : \tau$	Type family ▶ $\text{of} : \text{exp} \rightarrow \text{tp} \rightarrow \mathbf{type}.$
Derivation ▶ $\mathcal{D} :: \Gamma \vdash e : \tau$	Well-typed term ▶ $M : \text{of } E T$
Proof checking	Type checking

Refinement types

- More precise layer of classification beyond -- but correlated with -- the usual types
- “Inclusion” or “implication” as subtyping, e.g.:
 - ▶ all *values* are *expressions*
 - ▶ all *odd natural numbers* are *positive*

Refinement types

- More precise layer of classification beyond -- but correlated with -- the usual types
- “Inclusion” or “implication” as subtyping, e.g.:
 - ▶ all *values* are *expressions*
 - ▶ all *odd natural numbers* are *positive*
- More interesting types means more interesting judgements!

Refinement types

- More precise layer of classification beyond -- but correlated with -- the usual types
- “Inclusion” or “implication” as subtyping, e.g.:
 - ▶ all *values* are *expressions*
 - ▶ all *odd natural numbers* are *positive*
- More interesting types means more interesting judgements!
 - ▶ ... and better representations!

Teaser Example: λ -calculus

$\text{exp} : \text{type.}$

$\text{lam} : (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp.}$

$\text{app} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp.}$

Teaser Example: λ -calculus

$\text{exp} : \mathbf{type}.$

$\text{lam} : (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}.$

$\text{app} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}.$

$\text{eval} : \text{exp} \rightarrow \text{exp} \rightarrow \mathbf{type}.$

$\text{ev-lam} : \text{eval} (\text{lam } \lambda x. E \ x) (\text{lam } \lambda x. E \ x)$

$\text{ev-app} : \text{eval} (\text{app } E_1 \ E_2) \ V$

$\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' \ x)$

$\leftarrow \text{eval } E_2 \ V_2$

$\leftarrow \text{eval } (E_1' \ V_2) \ V.$

Teaser Example: λ -calculus

$\text{exp} : \mathbf{type}$. $\text{cmp} \sqsubset \text{exp}$. $\text{val} \sqsubset \text{exp}$.

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}$. $\text{val} \leq \text{cmp}$.

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}$.

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}$.

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x)$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$

$\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$

$\leftarrow \text{eval } E_2 V_2$

$\leftarrow \text{eval } (E_1' V_2) V$.

Teaser Example: λ -calculus

$\text{exp} : \mathbf{type}$. $\text{cmp} \sqsubset \text{exp}$. $\text{val} \sqsubset \text{exp}$.

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}$. $\text{val} \leq \text{cmp}$.

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}$.

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}$.

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x)$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$

$\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$

$\leftarrow \text{eval } E_2 V_2$

$\leftarrow \text{eval } (E_1' V_2) V$.

Example: natural numbers

Example: natural numbers

`nat : type.`

`z : nat.`

`s : nat → nat.`

Example: natural numbers

`nat : type.`

`z : nat.`

`s : nat → nat.`

`double : nat → nat → type.`

`dbl-z : double z z.`

`dbl-s : double (s N) (s (s (N2)))
← double N N2.`

Example: natural numbers

nat : type.

z : nat.

s : nat → nat.

double : nat → nat → type.

dbl-z : double z z.

dbl-s : double (s N) (s (s (N2)))

← double N N2.



always even!

Option 1: explicit proofs

- Represent *evenness* and *oddness* as *judgements* on natural numbers.

Option 1: explicit proofs

- Represent *evenness* and *oddness* as *judgements* on natural numbers.
- **Cumbersome:** definitions must be “proof-carrying”, manipulate witnesses.

Option 1: explicit proofs

Option 1: explicit proofs

`even : nat → type.`

`odd : nat → type.`

`ev-z : even z.`

`ev-s : even (s N) ← odd N.`

`od-s : odd (s N) ← even N.`

Option 1: explicit proofs

`even : nat → type.`

`odd : nat → type.`

`ev-z : even z.`

`ev-s : even (s N) ← odd N.`

`od-s : odd (s N) ← even N.`

`double : nat → ΠN2:nat. even N2 → type.`

`dbl-z : double z z ev-z.`

`dbl-s : double N (s (s N2)) (ev-s (od-s Deven))
← double N N2 Deven.`

Option 2: implicit proofs

- Represent *even* and *odd* as new types, distinct from the natural numbers.

Option 2: implicit proofs

- Represent *even* and *odd* as new types, distinct from the natural numbers.
- **Heavyweight:** need conversions between various types.

Option 2: implicit proofs

Option 2: implicit proofs

even : type.

odd : type.

z_e : even.

s_e : odd \rightarrow even.

s_o : even \rightarrow odd.

Option 2: implicit proofs

even : type.

odd : type.

$z_e : \text{even}.$

$s_e : \text{odd} \rightarrow \text{even}.$

$s_o : \text{even} \rightarrow \text{odd}.$

double : nat \rightarrow even \rightarrow type.

$\text{dbl-z} : \text{double } z \ z_e.$

$\text{dbl-s} : \text{double } N \ (s_e \ (s_o \ N2))$

$\leftarrow \text{double } N \ N2.$

Option 2: intrinsic proofs

- But... need erasures from even and odd to nat

Option 2: intrinsic proofs

- But... need erasures from even and odd to nat

$\text{even2nat} : \text{even} \rightarrow \text{nat} \rightarrow \mathbf{type}.$

$\text{odd2nat} : \text{odd} \rightarrow \text{nat} \rightarrow \mathbf{type}.$

$e2n\text{-}z_e : \text{even2nat } z_e \text{ } z.$

$e2n\text{-}s_e : \text{even2nat } (s_e \ 0) \ (s \ N)$

$\leftarrow \text{odd2nat } 0 \ N.$

$o2n\text{-}s_o : \text{odd2nat } (s_o \ E) \ (s \ N)$

$\leftarrow \text{even2nat } E \ N.$

Option 3: metatheorem

- Represent *evenness* and *oddness* as *judgements* (as in Option 1 above).
- Prove a Twelf metatheorem: for every *doubling* derivation, there's an *evenness* derivation.

Option 3: metatheorem

- Represent *evenness* and *oddness* as *judgements* (as in Option 1 above).
- Prove a Twelf metatheorem: for every *doubling* derivation, there's an *evenness* derivation.
- **Problem:** less direct, and metatheorem checking is complex.

Option 3: metatheorem

Option 3: metatheorem

`even : nat → type.`

`odd : nat → type.`

`% ... ev-z, ev-s, od-s ...`

Option 3: metatheorem

```
even : nat → type.
```

```
odd : nat → type.
```

```
% ... ev-z, ev-s, od-s ...
```

```
double-even : double N N2 → even N2 → type.
```

```
%mode double-even +Ddbl -Deven
```

```
- : double-even dbl-z even-z
```

```
- : double-even (dbl-s Ddbl) (ev-s (od-s Deven))
```

```
← double-even Ddbl Deven.
```

```
%worlds () (double-even Ddbl Deven).
```

```
%total Ddbl (double-even Ddbl Deven).
```

Better option: refinements

- Represent *evenness* and *oddness* as *refinements* of the type of natural numbers.

Better option: refinements

- Represent *evenness* and *oddness* as *refinements* of the type of natural numbers.
- **Simple:** doubling judgement doesn't change.

Better option: refinements

- Represent *evenness* and *oddness* as *refinements* of the type of natural numbers.
- **Simple:** doubling judgement doesn't change.
- **Lightweight:** constructors remain the same.

Better option: refinements

- Represent *evenness* and *oddness* as *refinements* of the type of natural numbers.
- **Simple:** doubling judgement doesn't change.
- **Lightweight:** constructors remain the same.
- **Direct:** strong typing guarantee on derivations.

Better option: refinements

$\text{even} \sqsubseteq \text{nat.}$

$\text{odd} \sqsubseteq \text{nat.}$

$z :: \text{even.}$

$s :: \text{even} \rightarrow \text{odd} \wedge \text{odd} \rightarrow \text{even.}$

Better option: refinements

`even` \sqsubset `nat`.

`odd` \sqsubset `nat`.

`z` :: `even`.

`s` :: `even` \rightarrow `odd` \wedge `odd` \rightarrow `even`.

`double` :: `nat` \rightarrow `even` \rightarrow **`type`**.

`dbl-z` :: `double` `z` `z`.

`dbl-s` :: `double` (`s` `N`) (`s` (`s` (`N2`)))
 \leftarrow `double` `N` `N2`.

Better option: refinements

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

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

$z :: \text{even}.$

$s :: \text{even} \rightarrow \text{odd} \wedge \text{odd} \rightarrow \text{even}.$

$\text{double} :: \text{nat} \rightarrow \text{even} \rightarrow \text{type}.$

$\text{dbl-z} :: \text{double } z \ z.$

$\text{dbl-s} :: \text{double } (s \ N) \ (s \ (s \ (N2)))$
 $\leftarrow \text{double } N \ N2.$

Better option: refinements

$z :: \text{even.}$

$s :: \text{even} \rightarrow \text{odd} \wedge \text{odd} \rightarrow \text{even.}$

$\text{double} :: \text{nat} \rightarrow \text{even} \rightarrow \mathbf{\text{type.}}$

$\text{dbl-z} :: \text{double } z \ z.$

$\text{dbl-s} :: \text{double } (s \ N) \ (s \ (s \ N2))$
 $\leftarrow \text{double } N \ N2.$

Better option: refinements

$z :: \text{even.}$

$s :: \text{even} \rightarrow \text{odd} \wedge \text{odd} \rightarrow \text{even.}$

$\text{double} :: \text{nat} \rightarrow \text{even} \rightarrow \mathbf{\text{type.}}$

$\text{dbl-z} :: \text{double } z \ z.$

$\text{dbl-s} :: \text{double } (s \ N) \ (s \ (s \ N2))$
 $\quad \leftarrow \text{double } N \ N2.$

Better option: refinements

`z :: even.`

`s :: even → odd ∧ odd → even.`

`double :: nat → even → type.`

`dbl-z :: double z z.`

`dbl-s :: double (s N) (s (s N2))
← double N N2.`

Better option: refinements

$z :: \text{even.}$

$s :: \text{even} \rightarrow \text{odd} \wedge \text{odd} \rightarrow \text{even.}$

$\text{double} :: \text{nat} \rightarrow \text{even} \rightarrow \text{type.}$

$\text{dbl-z} :: \text{double } z \ z.$

$\text{dbl-s} :: \text{double } (s \ N) \ (s \ (s \ N2))$

$\leftarrow \text{double } N \ N2.$

Better option: refinements

`z :: even.`

`s :: even → odd` \wedge `odd → even.`

`double :: nat → even → type.`

`dbl-z :: double z z.`

`dbl-s :: double (s N) (s (s N2))`
`← double N N2.`

Better option: refinements

$z :: \text{even.}$

$s :: \text{even} \rightarrow \text{odd} \wedge \text{odd} \rightarrow \text{even.}$

$\text{double} :: \text{nat} \rightarrow \text{even} \rightarrow \text{type.}$

$\text{dbl-z} :: \text{double } z \ z.$

$\text{dbl-s} :: \text{double } (s \ N) \ (s \ (s \ N2))$
 $\leftarrow \text{double } N \ N2.$

Better option: refinements

`z :: even.`

`s :: even → odd ∧ odd → even.`

`double :: nat → even → type.`

`dbl-z :: double z z.`

`dbl-s :: double (s N) (s (s N2))`
`← double N N2.`

Outline

- ✓ Introduction: Motivation
- Completed work
 - ▶ LFR type theory and metatheory
 - ▶ Higher-sort subsorting
- Proposed work
 - ▶ Unification
 - ▶ Type reconstruction
 - ▶ Coverage checking
- Summary

Adequacy

- Does my encoding mean anything?
- Strategy: exhibit a *compositional bijection* between *mathematical objects* and *canonical forms* following *judgements as types*.
 - ▶ “*Canonical forms*” are β -normal and η -long.

Canonical forms method

- Represent *only* the canonical forms:
 - ▶ β -normal syntactically
 - ▶ η -long through typing
 - ▶ hereditary substitutions contract redexes
- Simplifies metatheory, emphasizes adequacy
- Concurrent LF (Watkins, *et al*, 2003)

LF typing

- Bidirectional typing
- Synthesis: $\Gamma \vdash R \Rightarrow A$
 - ▶ elims: $R ::= x \mid c \mid R N$
- Checking: $\Gamma \vdash N \Leftarrow A$
 - ▶ intros: $N ::= R \mid \lambda x. N$

Checking

- Key rule:

$$\Gamma \vdash N \Leftarrow A$$

$$\frac{\Gamma \vdash R \Rightarrow P' \quad P' = P}{\Gamma \vdash R \Leftarrow P}$$

Checking

$$\Gamma \vdash N \Leftarrow A$$

- Key rule:

- ▶ base type, so atoms fully applied

$$\frac{\Gamma \vdash R \Rightarrow P' \quad P' = P}{\Gamma \vdash R \Leftarrow P}$$

Checking

$$\Gamma \vdash N \Leftarrow A$$

- Key rule:

- ▶ base type, so atoms fully applied
- ▶ the only appeal to type equality

$$\frac{\Gamma \vdash R \Rightarrow P' \quad P' = P}{\Gamma \vdash R \Leftarrow P}$$

Checking with subtyping

$$\Gamma \vdash N \Leftarrow A$$

- Key change:

- ▶ equality becomes subtyping
- ▶ subtyping... only at base type?

$$\frac{\Gamma \vdash R \Rightarrow P' \quad P' \leq P}{\Gamma \vdash R \Leftarrow P}$$

Checking with subtyping

$$\Gamma \vdash N \Leftarrow A$$

- Key change:
 - ▶ equality becomes subtyping
 - ▶ subtyping... only at base type?

$$\frac{\Gamma \vdash R \Rightarrow P' \quad P' \leq P}{\Gamma \vdash R \Leftarrow P}$$

Intersections

- Similar to product types, but no proof term

$$\Gamma \vdash N \Leftarrow A_1 \quad \Gamma \vdash N \Leftarrow A_2$$

$$\Gamma \vdash N \Leftarrow A_1 \wedge A_2$$

$$\Gamma \vdash N \Leftarrow \top$$

$$\Gamma \vdash R \Rightarrow A_1 \wedge A_2$$

$$\Gamma \vdash R \Rightarrow A_1$$

$$\Gamma \vdash R \Rightarrow A_1 \wedge A_2$$

$$\Gamma \vdash R \Rightarrow A_2$$

(Refinement restriction)

- *Sorts*: more precise classifiers than *types*.
 - ▶ *subsorting*, *intersection sorts*
- Refinement relation: $\Gamma \vdash S \sqsubset A$
- Only *sort-check* *well-typed* terms:
 - ▶ e.g. $\Gamma \vdash N \Leftarrow S$ only sensible if $\Gamma \vdash N \Leftarrow A$
for some A such that $\Gamma \vdash S \sqsubset A$

Important principles

- **Substitution:** if $\Gamma, x:A \vdash N \Leftarrow B$ and $\Gamma \vdash M \Leftarrow A$, then $\Gamma \vdash [M/x]_A N \Leftarrow B$.
- **Identity:** for all A : $\Gamma, x:A \vdash \eta_A(x) \Leftarrow A$.

Subtyping

$$\Gamma \vdash N \Leftarrow A$$

- Key rule:

$$\frac{\Gamma \vdash R \Rightarrow P' \quad P' \leq P}{\Gamma \vdash R \Leftarrow P}$$

- ▶ Bidirectional: subtyping only at mode switch
- ▶ Canonical: mode switch only at base type

Subtyping at higher types?

- Structural rules? e.g.

$$\frac{A_2 \leq A_1 \quad B_1 \leq B_2}{A_1 \rightarrow B_1 \leq A_2 \rightarrow B_2}$$

- Distributivity?

$$A \rightarrow (B_1 \wedge B_2) \leq (A \rightarrow B_1) \wedge (A \rightarrow B_2)$$

Subtyping at higher types!

- **Intrinsic subtyping:** if $A \leq B$ and $\Gamma \vdash N \Leftarrow A$, then $\Gamma \vdash N \Leftarrow B$.

Subtyping at higher types!

- **Intrinsic subtyping:** if $A \leq B$ and $\Gamma \vdash N \Leftarrow A$, then $\Gamma \vdash N \Leftarrow B$.
- **Equivalently:** if $A \leq B$, then $\Gamma, x:A \vdash \eta_A(x) \Leftarrow B$.

Subtyping at higher types!

- **Intrinsic subtyping:** if $A \leq B$ and $\Gamma \vdash N \Leftarrow A$, then $\Gamma \vdash N \Leftarrow B$.
- **Equivalently:** if $A \leq B$, then $\Gamma, x:A \vdash \eta_A(x) \Leftarrow B$.
 - ▶ just like the Identity principle!

Subtyping at higher types!

- **Intrinsic subtyping:** if $A \leq B$ and $\Gamma \vdash N \Leftarrow A$, then $\Gamma \vdash N \Leftarrow B$.
- **Equivalently:** if $A \leq B$, then $\Gamma, x:A \vdash \eta_A(x) \Leftarrow B$.
 - ▶ just like the Identity principle!
 - ▶ ... also the Substitution principle ...

Subtyping at higher types!

- **Intrinsic subtyping:** if $A \leq B$ and $\Gamma \vdash N \Leftarrow A$, then $\Gamma \vdash N \Leftarrow B$.
- **Equivalently:** if $A \leq B$, then $\Gamma, x:A \vdash \eta_A(x) \Leftarrow B$.
 - ▶ just like the Identity principle!
 - ▶ ... also the Substitution principle ...
- Usual rules all *sound* in this sense.

Subtyping at higher types?

- ... and also *complete*!
- **Theorem:** if $\Gamma, x:A \vdash \eta_A(x) \Leftarrow B$, then $A \leq B$.
- **Or:** if $\Gamma \vdash N \Leftarrow A$ implies $\Gamma \vdash N \Leftarrow B$, then $A \leq B$.

Subtyping at higher types?

- ... and also *complete*!
- **Theorem:** if $\Gamma, x:A \vdash \eta_A(x) \Leftarrow B$, then $A \leq B$.
- **Or:** if $\Gamma \vdash N \Leftarrow A$ implies $\Gamma \vdash N \Leftarrow B$, then $A \leq B$.
- There are no new subtyping principles.

Outline

- ✓ Introduction: Motivation
- ✓ Completed work
 - ▶ LFR type theory and metatheory
 - ▶ Higher-sort subsorting
- Proposed work
 - ▶ Unification
 - ▶ Type reconstruction
 - ▶ Coverage checking
- Summary

Proposed work

- *Want:* a usable framework!

Proposed work

- *Want*: a usable framework!
- *Need*: type reconstruction...

Proposed work

- *Want*: a usable framework!
- *Need*: type reconstruction...
- *Depends on*: unification...

Proposed work

- *Want:* a usable framework!
- *Need:* type reconstruction...
- *Depends on:* unification...
- *Useful for:* coverage checking...

Unification

$\text{bool} : \text{type}.$
 $t \sqsubset \text{bool}.$
 $f \sqsubset \text{bool}.$

$\text{true} :: t.$
 $\text{false} :: f.$

$\text{and} :: t \rightarrow t \rightarrow t$
 $\quad \wedge t \rightarrow f \rightarrow f$
 $\quad \wedge f \rightarrow t \rightarrow f$
 $\quad \wedge f \rightarrow f \rightarrow f.$

Unification

- Suppose $X::f$

$\text{bool} : \text{type}.$

$t \sqsubset \text{bool}.$

$f \sqsubset \text{bool}.$

$\text{true} :: t.$

$\text{false} :: f.$

$\text{and} :: t \rightarrow t \rightarrow t$

$\wedge t \rightarrow f \rightarrow f$

$\wedge f \rightarrow t \rightarrow f$

$\wedge f \rightarrow f \rightarrow f.$

Unification

- Suppose $X::f$
- $X \doteq$ and $M N$?

$\text{bool} : \text{type.}$

$t \sqsubset \text{bool.}$

$f \sqsubset \text{bool.}$

$\text{true} :: t.$

$\text{false} :: f.$

$\text{and} :: t \rightarrow t \rightarrow t$

$\wedge t \rightarrow f \rightarrow f$

$\wedge f \rightarrow t \rightarrow f$

$\wedge f \rightarrow f \rightarrow f.$

Unification

- Suppose $X::f$
- $X \doteq$ and $M N$?
 - $X :=$ and $X_1 X_2$

$\text{bool} : \text{type.}$

$t \sqsubset \text{bool.}$

$f \sqsubset \text{bool.}$

$\text{true} :: t.$

$\text{false} :: f.$

$\text{and} :: t \rightarrow t \rightarrow t$

$\wedge t \rightarrow f \rightarrow f$

$\wedge f \rightarrow t \rightarrow f$

$\wedge f \rightarrow f \rightarrow f.$

Unification

- Suppose $X::f$
- $X \doteq$ and $M N$?
 - $\parallel \cdot \parallel$
 - ▶ $X :=$ and $X_1 X_2$
 - ▶ but $X_1 :: ?$, $X_2 :: ?$...
three incomparable possibilities!

$\text{bool} : \text{type.}$

$t \sqsubset \text{bool.}$

$f \sqsubset \text{bool.}$

$\text{true} :: t.$

$\text{false} :: f.$

$\text{and} :: t \rightarrow t \rightarrow t$

$\wedge t \rightarrow f \rightarrow f$

$\wedge f \rightarrow t \rightarrow f$

$\wedge f \rightarrow f \rightarrow f.$

Unification

- **Solution?** unify as usual, but maintain typing constraints.
 - ▶ Kohlhase and Pfenning (1993) used *subtyping* constraints.

Type reconstruction

$eq : nat \rightarrow nat \rightarrow \mathbf{type}.$

$smart\text{-}eq \sqsubset eq :: even \rightarrow even \rightarrow \mathbf{sort}$

$\wedge odd \rightarrow odd \rightarrow \mathbf{sort}.$

$dumb\text{-}eq \sqsubset eq :: \top \rightarrow \top \rightarrow \mathbf{sort}.$

$coerce :: smart\text{-}eq X Y \rightarrow dumb\text{-}eq X Y.$

Type reconstruction

$eq : nat \rightarrow nat \rightarrow \mathbf{type}.$

$smart\text{-}eq \sqsubset eq :: even \rightarrow even \rightarrow \mathbf{sort}$

$\wedge odd \rightarrow odd \rightarrow \mathbf{sort}.$

$dumb\text{-}eq \sqsubset eq :: \top \rightarrow \top \rightarrow \mathbf{sort}.$

$coerce :: \mathbf{smart\text{-}eq } X Y \rightarrow dumb\text{-}eq } X Y.$

Type reconstruction

$eq : nat \rightarrow nat \rightarrow \mathbf{type}.$

$smart\text{-}eq \sqsubset eq :: even \rightarrow even \rightarrow \mathbf{sort}$

$\wedge odd \rightarrow odd \rightarrow \mathbf{sort}.$

$dumb\text{-}eq \sqsubset eq :: \top \rightarrow \top \rightarrow \mathbf{sort}.$

$coerce :: smart\text{-}eq(X)Y \rightarrow dumb\text{-}eq X Y.$

Type reconstruction

```
eq : nat → nat → type.
smart-eq ⊆ eq :: even → even → sort
                ∧ odd → odd → sort.
dumb-eq ⊆ eq :: ⊤ → ⊤ → sort.

coerce :: smart-eq X Y → dumb-eq X Y.
```

```
X::even
X::odd
```

Type reconstruction

$eq : nat \rightarrow nat \rightarrow \mathbf{type}.$

$smart\text{-}eq \sqsubset eq :: even \rightarrow even \rightarrow \mathbf{sort}$

$\wedge odd \rightarrow odd \rightarrow \mathbf{sort}.$

$dumb\text{-}eq \sqsubset eq :: \top \rightarrow \top \rightarrow \mathbf{sort}.$

$coerce :: smart\text{-}eq X \mathbf{Y} \rightarrow dumb\text{-}eq X Y.$

$X :: even$

$X :: odd$

Type reconstruction

$eq : nat \rightarrow nat \rightarrow \mathbf{type}.$

$smart\text{-}eq \sqsubset eq :: \text{even} \rightarrow \text{even} \rightarrow \mathbf{sort}$

$\wedge \text{odd} \rightarrow \text{odd} \rightarrow \mathbf{sort}.$

$dumb\text{-}eq \sqsubset eq :: \top \rightarrow \top \rightarrow \mathbf{sort}.$

$coerce :: smart\text{-}eq X Y \rightarrow dumb\text{-}eq X Y.$

$X::\text{even}$

$X::\text{odd}$

$Y::\text{even}$

$Y::\text{odd}$

Type reconstruction

$eq : nat \rightarrow nat \rightarrow \mathbf{type}.$

$smart\text{-}eq \sqsubset eq :: even \rightarrow even \rightarrow \mathbf{sort}$

$\wedge odd \rightarrow odd \rightarrow \mathbf{sort}.$

$dumb\text{-}eq \sqsubset eq :: \top \rightarrow \top \rightarrow \mathbf{sort}.$

$coerce :: smart\text{-}eq X Y \rightarrow dumb\text{-}eq X Y.$

$X::even$

$X::odd$

$Y::even$

$Y::odd$

Type reconstruction

$eq : nat \rightarrow nat \rightarrow \mathbf{type}.$

$smart\text{-}eq \sqsubset eq :: \mathbf{even} \rightarrow \mathbf{even} \rightarrow \mathbf{sort}$
 $\wedge \mathbf{odd} \rightarrow \mathbf{odd} \rightarrow \mathbf{sort}.$

$dumb\text{-}eq \sqsubset eq :: \top \rightarrow \top \rightarrow \mathbf{sort}.$

$coerce :: smart\text{-}eq\ X\ Y \rightarrow dumb\text{-}eq\ X\ Y.$

$X::\mathbf{even}$	\Rightarrow	$Y::\mathbf{even}$
$X::\mathbf{odd}$	\Rightarrow	$Y::\mathbf{odd}$

Type reconstruction

$eq : nat \rightarrow nat \rightarrow \mathbf{type}.$

$smart\text{-}eq \sqsubset eq :: even \rightarrow even \rightarrow \mathbf{sort}$

$\wedge odd \rightarrow odd \rightarrow \mathbf{sort}.$

$dumb\text{-}eq \sqsubset eq :: \top \rightarrow \top \rightarrow \mathbf{sort}.$

$coerce :: \Pi X::even. \Pi Y::even.$

$smart\text{-}eq X Y \rightarrow dumb\text{-}eq X Y$

$\wedge \Pi X::odd. \Pi Y::odd.$

$smart\text{-}eq X Y \rightarrow dumb\text{-}eq X Y.$

Type reconstruction

- **Typical strategy:** consider all possibilities, prune along the way.
- Does this always work?

Coverage checking

$\text{exp} : \mathbf{type}. \text{cmp} \sqsubset \text{exp}. \text{val} \sqsubset \text{exp}. \text{val} \leq \text{cmp}.$

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}.$

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}.$

Coverage checking

$\text{exp} : \mathbf{type}. \text{cmp} \sqsubset \text{exp}. \text{val} \sqsubset \text{exp}. \text{val} \leq \text{cmp}.$

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}.$

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}.$

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}.$

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x).$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$
 $\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$
 $\leftarrow \text{eval } E_2 V_2$
 $\leftarrow \text{eval } (E_1' V_2) V.$

Coverage checking

$\text{exp} : \mathbf{type}. \text{cmp} \sqsubset \text{exp}. \text{val} \sqsubset \text{exp}. \text{val} \leq \text{cmp}.$

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}.$

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}.$

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}.$

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x).$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$

$\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$

$\leftarrow \text{eval } E_2 V_2$

$\leftarrow \text{eval } (E_1' V_2) V.$

Coverage checking

$\text{exp} : \mathbf{type}. \text{cmp} \sqsubset \text{exp}. \text{val} \sqsubset \text{exp}. \text{val} \leq \text{cmp}.$

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}.$

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}.$

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}.$

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x).$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$

$\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$

$\leftarrow \text{eval } E_2 V_2$

$\leftarrow \text{eval } (E_1' V_2) V.$

Coverage checking

$\text{exp} : \mathbf{type}. \text{cmp} \sqsubset \text{exp}. \text{val} \sqsubset \text{exp}. \text{val} \leq \text{cmp}.$

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}.$

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}.$

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}.$

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x).$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$
 $\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$
 $\leftarrow \text{eval } E_2 V_2$
 $\leftarrow \text{eval } (E_1' V_2) V.$

Coverage checking

$\text{exp} : \mathbf{type}. \text{cmp} \sqsubset \text{exp}. \text{val} \sqsubset \text{exp}. \text{val} \leq \text{cmp}.$

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}.$

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}.$

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}.$

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x).$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$

$\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$

$\leftarrow \text{eval } E_2 V_2$

$\leftarrow \text{eval } (E_1' V_2) V.$

Coverage checking

$\text{exp} : \mathbf{type}. \text{cmp} \sqsubset \text{exp}. \text{val} \sqsubset \text{exp}. \text{val} \leq \text{cmp}.$

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}.$

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}.$

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}.$

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x).$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$

$\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$

$\leftarrow \text{eval } E_2 V_2$

$\leftarrow \text{eval } (E_1' V_2) V.$

Coverage checking

$\text{exp} : \mathbf{type}. \text{cmp} \sqsubset \text{exp}. \text{val} \sqsubset \text{exp}. \text{val} \leq \text{cmp}.$

$\text{lam} :: (\text{val} \rightarrow \text{cmp}) \rightarrow \text{val}.$

$\text{app} :: \text{cmp} \rightarrow \text{cmp} \rightarrow \text{cmp}.$

$\text{eval} :: \text{cmp} \rightarrow \text{val} \rightarrow \mathbf{sort}.$

$\text{ev-lam} :: \text{eval} (\text{lam } \lambda x. E x) (\text{lam } \lambda x. E x).$

$\text{ev-app} :: \text{eval} (\text{app } E_1 E_2) V$
 $\leftarrow \text{eval } E_1 (\text{lam } \lambda x. E_1' x)$
 $\leftarrow \text{eval } E_2 V_2$
 $\leftarrow \text{eval } (E_1' V_2) V.$

Coverage checking

- **Key idea:** leverage precise sort information.
- Interesting interactions with type reconstruction...

Summary

- **Completed:**
 - ▶ LFR type theory and metatheory
 - ▶ Interpretations of higher-type subtyping
- **Proposed:**
 - ▶ Make a *usable framework* by specifying *unification* and *type reconstruction*.
 - ▶ Head start on *metatheorem proving* with *coverage checking*.

Summary



(brilliant!)

- **Thesis:** Refinement types are a *useful* and *practical* addition to the logical framework LF.
- Better representations will make LF a better tool.