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 ⇒ 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 ::= … 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

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

exp : **type**.

lam : (exp → exp) → exp.
app : exp → exp → exp.

Teaser Example: λ-calculus

exp : **type**.

lam : (exp → exp) → exp.
app : exp → exp → exp.

eval : exp → exp → **type**.

ev-lam : eval (lam $\lambda x.\ E\ x$) (lam $\lambda x.\ E$ x)
ev-app : eval (app $E_1\ E_2$) V
← eval E_1 (lam $\lambda x.\ E_1'\ x$)
← eval $E_2\ V_2$
← eval $(E_1'\ V_2)\ V$.

Teaser Example: λ-calculus

exp : **type**.   cmp ⊏ exp.   val ⊏ exp.

lam :: (val → cmp) → val.   val ≤ cmp.
app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam λ*x*. *E* *x*) (lam λ*x*. *E* x)
ev-app :: eval (app E_1 E_2) *V*
← eval E_1 (lam λ*x*. E_1' *x*)
← eval E_2 V_2
← eval (E_1' V_2) *V*.

Teaser Example: λ-calculus

exp : **type**.   cmp ⊏ exp.   val ⊏ exp.

lam :: (val → cmp) → val.   val ≤ cmp.

app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam λ*x*. *E* *x*) (lam λ*x*. *E* x)

ev-app :: eval (app E_1 E_2) *V*

← eval E_1 (lam λ*x*. E_1' *x*)

← eval E_2 V_2

← 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.

always even!

double : nat → nat → **type**.

dbl-z : double z z.
dbl-s : double (s *N*) (s (s (*N2*))
 ← double *N* *N2*.

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 → Π*N2*: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 → even.
s_o : even → odd.

Option 2: implicit proofs

even : **type**.
odd : **type**.

z_e : even.
s_e : odd → even.
s_o : even → odd.

double : nat → even → **type**.

dbl-z : double z z_e.
dbl-s : double *N* (s_e (s_o *N2*))
← 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

```
even2nat : even → nat → type.
odd2nat : odd → nat → type.

e2n-ze : even2nat ze z.
e2n-se : even2nat (se O) (s N)
           ← odd2nat O N.
o2n-so : odd2nat (so E) (s N)
           ← 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

$even \sqsubset nat.$

$odd \sqsubset nat.$

z :: even.

s :: even → odd ∧ odd → even.

Better option: refinements

even ⊏ nat.

odd ⊏ nat.

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

```
even ⊏ nat.
odd ⊏ nat.

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 :: 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 :: 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 :: 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 :: 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 :: 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 :: 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 :: 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

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

- Key rule:

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

Checking

$\Gamma \vdash N \Leftarrow A$

- Key rule:
  - base type, so atoms fully applied

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

Checking

$\Gamma \vdash N \Leftarrow A$

- Key rule:
  - base type, so atoms fully applied
  - the only appeal to type equality

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

Checking with subtyping

$\Gamma \vdash N \Leftarrow A$

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

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

Checking with subtyping

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

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

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

Intersections

- Similar to product types, but no proof term

$$\frac{\Gamma \vdash N \Leftarrow A_1 \qquad \Gamma \vdash N \Leftarrow A_2}{\Gamma \vdash N \Leftarrow A_1 \wedge A_2} \qquad \frac{}{\Gamma \vdash N \Leftarrow \top}$$

$$\frac{\Gamma \vdash R \Rightarrow A_1 \wedge A_2}{\Gamma \vdash R \Rightarrow A_1} \qquad \frac{\Gamma \vdash R \Rightarrow A_1 \wedge A_2}{\Gamma \vdash R \Rightarrow A_2}$$

(Refinement restriction)

- *Sorts*: more precise classifiers than *types*.
  - sub*sorting*, 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

$$\boxed{\Gamma \vdash N \Leftarrow A}$$

- Key rule:

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

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

Subtyping at higher types?

- Structural rules? e.g.

$$\frac{A_2 \leq A_1 \qquad B_1 \leq B_2}{A_1 \to B_1 \leq A_2 \to B_2}$$

- Distributivity?

$$\frac{}{A \to (B_1 \wedge B_2) \leq (A \to B_1) \wedge (A \to B_2)}$$

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

```
bool : type.
t ⊏ bool.
f ⊏ bool.

true :: t.
false :: f.
and :: t → t → t
     ∧ t → f → f
     ∧ f → t → f
     ∧ f → f → f.
```

Unification

- Suppose X::f

bool : type.
t ⊏ bool.
f ⊏ bool.

true :: t.
false :: f.
and :: t → t → t
∧ t → f → f
∧ f → t → f
∧ f → f → f.

Unification

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

bool : type.
t ⊏ bool.
f ⊏ bool.

true :: t.
false :: f.
and :: t → t → t
∧ t → f → f
∧ f → t → f
∧ f → f → f.

Unification

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

  $\qquad\qquad\quad \Vert\!\cdot \quad \Vert\!\cdot$

  - X := and X_1 X_2

bool : type.
t ⊏ bool.
f ⊏ bool.

true :: t.
false :: f.
and :: t → t → t
∧ t → f → f
∧ f → t → f
∧ f → f → f.

Unification

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

bool : type.
t ⊏ bool.
f ⊏ bool.

true :: t.
false :: f.
and :: t → t → t
∧ t → f → f
∧ f → t → f
∧ f → f → f.

Unification

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

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*.

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*.

Type reconstruction

$\text{eq} : \text{nat} \to \text{nat} \to \textbf{type}.$

$\text{smart-eq} \sqsubset \text{eq} :: \text{even} \to \text{even} \to \textbf{sort}$

$\wedge\ \text{odd} \to \text{odd} \to \textbf{sort}.$

$\text{dumb-eq} \sqsubset \text{eq} :: \top \to \top \to \textbf{sort}.$

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

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 → 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 → 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

Y::even
Y::odd

Type reconstruction

$\mathrm{eq} : \mathrm{nat} \to \mathrm{nat} \to \textbf{type}.$

$\mathrm{smart\text{-}eq} \sqsubset \mathrm{eq} :: \mathrm{even} \to \mathrm{even} \to \textbf{sort}$

$\wedge\ \mathrm{odd} \to \mathrm{odd} \to \textbf{sort}.$

$\mathrm{dumb\text{-}eq} \sqsubset \mathrm{eq} :: \top \to \top \to \textbf{sort}.$

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

X::even
X::odd

Y::even
Y::odd

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 → *Y*::even

X::odd → *Y*::odd

Type reconstruction

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

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

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

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

$\mathrm{coerce} :: \Pi X{::}\mathrm{even}.\ \Pi Y{::}\mathrm{even}.$

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

$\qquad \wedge\ \Pi X{::}\mathrm{odd}.\ \Pi Y{::}\mathrm{odd}.$

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

Type reconstruction

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

Coverage checking

exp : **type**. cmp ⊏ exp. val ⊏ exp. val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.

Coverage checking

exp : **type**. cmp ⊏ exp. val ⊏ exp. val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam $\lambda x.\, E\, x$) (lam $\lambda x.\, E\, x$).
ev-app :: eval (app E_1 E_2) V
← eval E_1 (lam $\lambda x.\, E_1'\, x$)
← eval E_2 V_2
← eval $(E_1'\, V_2)$ V.

Coverage checking

exp : **type**. cmp ⊏ exp. val ⊏ exp. val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam λx. E x) (lam λx. E x).
ev-app :: eval (app E_1 E_2) V
 ← eval E_1 (lam λx. E_1' x)
 ← eval E_2 V_2
 ← eval (E_1' V_2) V.

Coverage checking

exp : **type**. cmp ⊏ exp. val ⊏ exp. val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam λx. E x) (lam λx. E x).
ev-app :: eval (app E_1 E_2) V
← eval E_1 (lam λx. E_1' x)
← eval E_2 V_2
← eval (E_1' V_2) V.

Coverage checking

exp : **type**. cmp ⊏ exp. val ⊏ exp. val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam $\lambda x.\ E\ x$) (lam $\lambda x.\ E\ x$).
ev-app :: eval (app E_1 E_2) V
← eval E_1 (lam $\lambda x.\ E_1'\ x$)
← eval E_2 V_2
← eval (E_1' V_2) V.

Coverage checking

exp : **type**. cmp ⊏ exp. val ⊏ exp. val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam λx. E x) (lam λx. E x).
ev-app :: eval (app E_1 E_2) V
    ← eval E_1 (lam λx. E_1' x)
    ← eval E_2 V_2
    ← eval (E_1' V_2) V.

Coverage checking

exp : **type**. cmp ⊏ exp. val ⊏ exp. val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam λx. E x) (lam λx. E x).
ev-app :: eval (app E_1 E_2) V
← eval E_1 (lam λx. E_1' x)
← eval E_2 V_2
← eval (E_1' V_2) V.

Coverage checking

exp : **type**. cmp ⊏ exp. val ⊏ exp. val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.

eval :: cmp → val → **sort**.

ev-lam :: eval (lam λ*x*. *E x*) (lam λ*x*. *E x*).
ev-app :: eval (app E_1 E_2) *V*
← eval E_1 (lam λ*x*. E_1' *x*)
← eval E_2 V_2
← 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.