Refinement Types for Logical Frameworks

William Lovas

Thesis

Refinement types are a **useful** and **practical** extension to the LF logical framework.

Thesis

"judgments as types"

Refinement types are a **useful** and **practical** extension to the LF logical framework.

Thesis

Refinement types are a **useful** and **practical** extension to the LF logical framework.

subtyping (≤), *intersections* (∧)

Shoulders of Giants

Shoulders of Giants

deBruijn

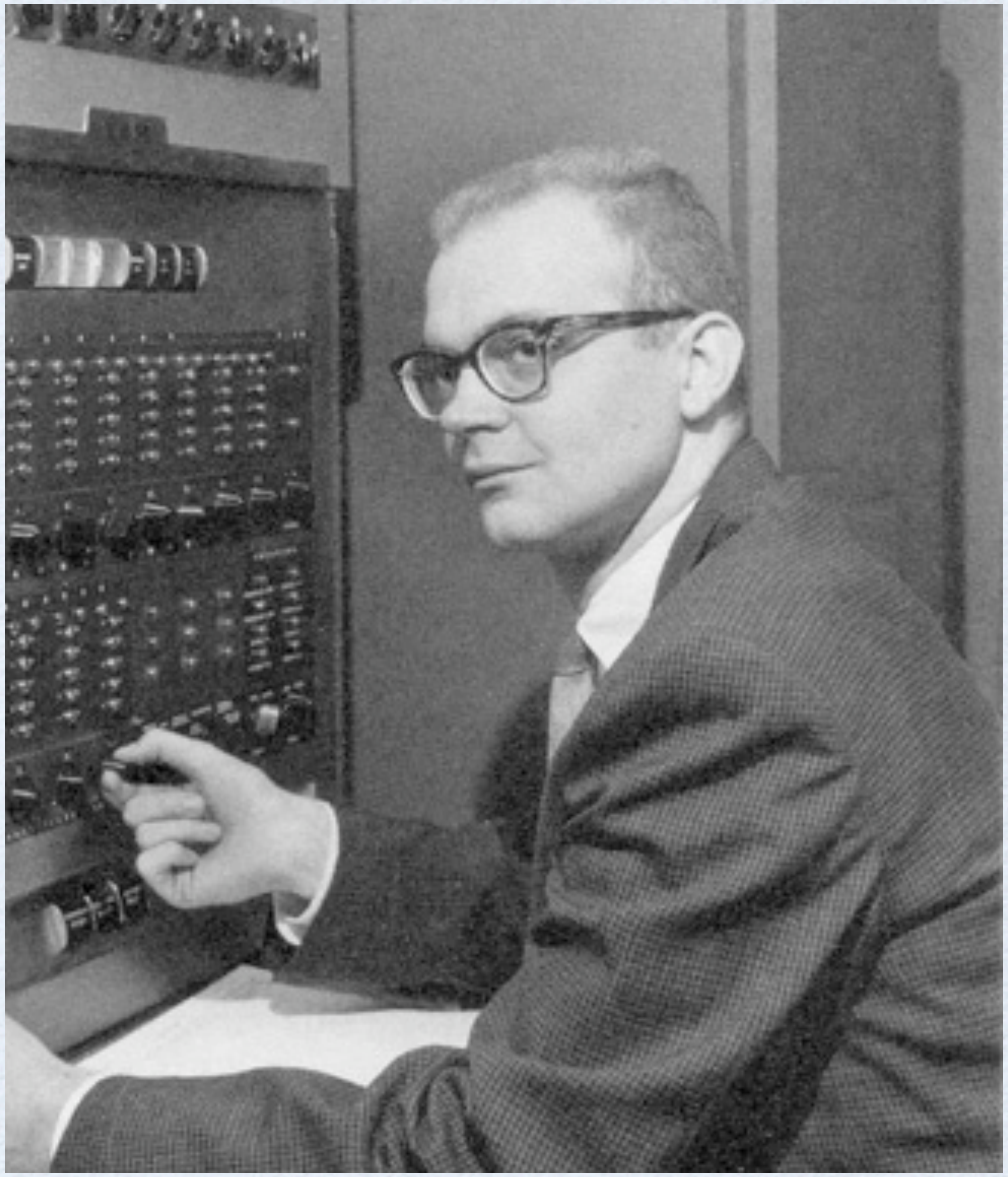
Shoulders of Giants

deBruijn

Shoulders of Giants

deBruijn

Knuth



Shoulders of Giants

deBruijn

May 20, 1968

Professor N. G. deBruijn
Combinatorial Conference
c/o Faculty of Mathematics
University of Waterloo
Waterloo, Ontario
Canada

Dear Dick:

I think I can demonstrate the usefulness of that idea of "subsorts" which I mentioned to you last week.

Enclosed is a proof that equivalence relations determine a partition, written in the extension of your language which I am proposing. The proof has three parts: Chapter 1 introduces the boolean operations and quantifiers; Chapter 2 introduces some aspects of set theory; and Chapter 3 is the proof itself.

When I write

$$\alpha := \text{PN} \quad \underline{\text{sort}}\ (\xi)$$

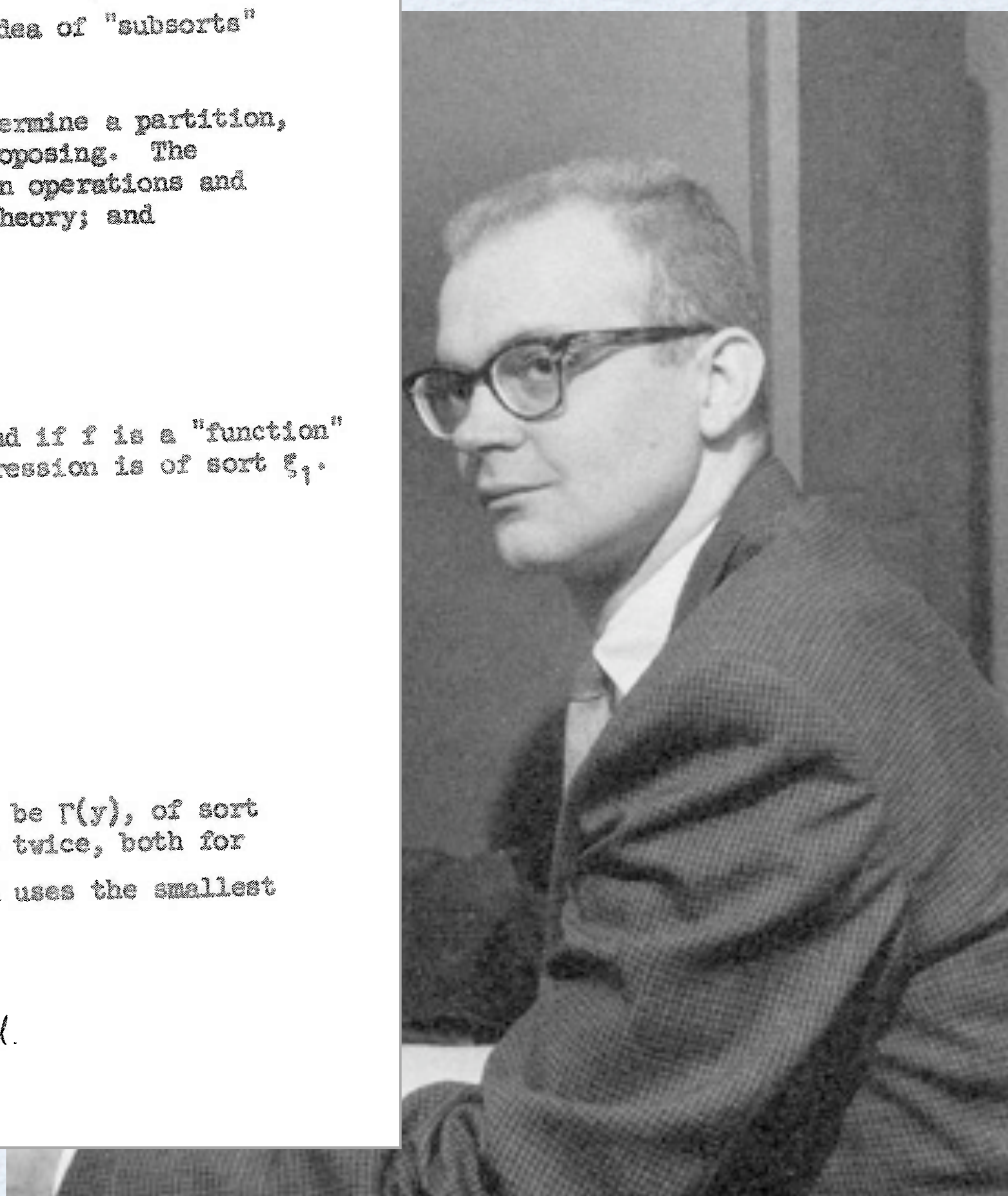
I mean α is a subsort of ξ. Then if y is of sort α, and if f is a "function" $[x\ \xi]\xi_1$, I am allowed to write $\{y\}f$ and the latter expression is of sort ξ_1. Furthermore constructions such as

$$\begin{array}{l|ll} x & & \xi \\ & \theta := \Sigma(x) & \xi_1 \\ \hline x & & \alpha \\ & \theta := \Gamma(x) & \xi_2 \end{array}$$

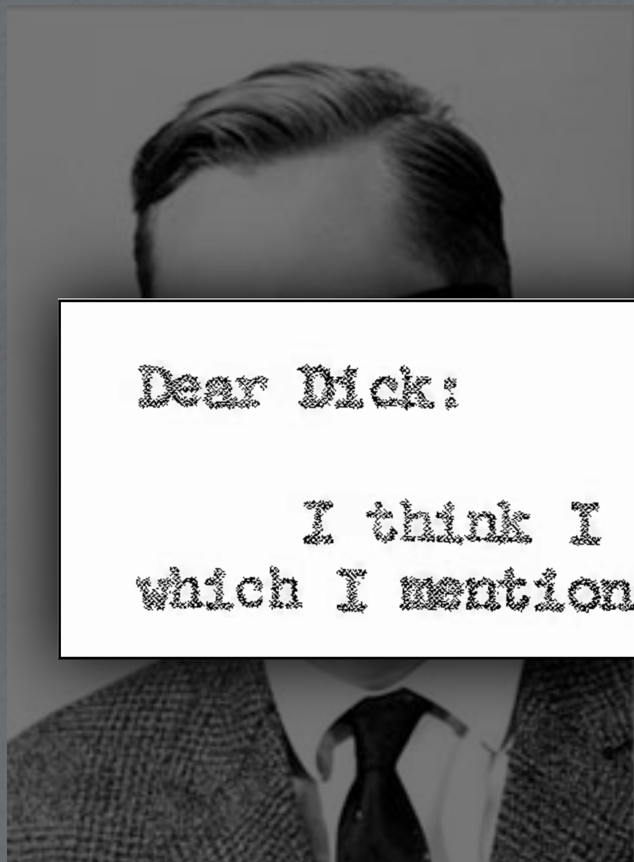
may be used; in these circumstances $\theta(y)$ is defined to be $\Gamma(y)$, of sort ξ_2. In other words I allow the symbol θ to be defined twice, both for sort ξ and its subsort α; the definition of $\theta(y)$ which uses the smallest sort containing y is always used.*

* Or maybe it is better to let either definition be used.

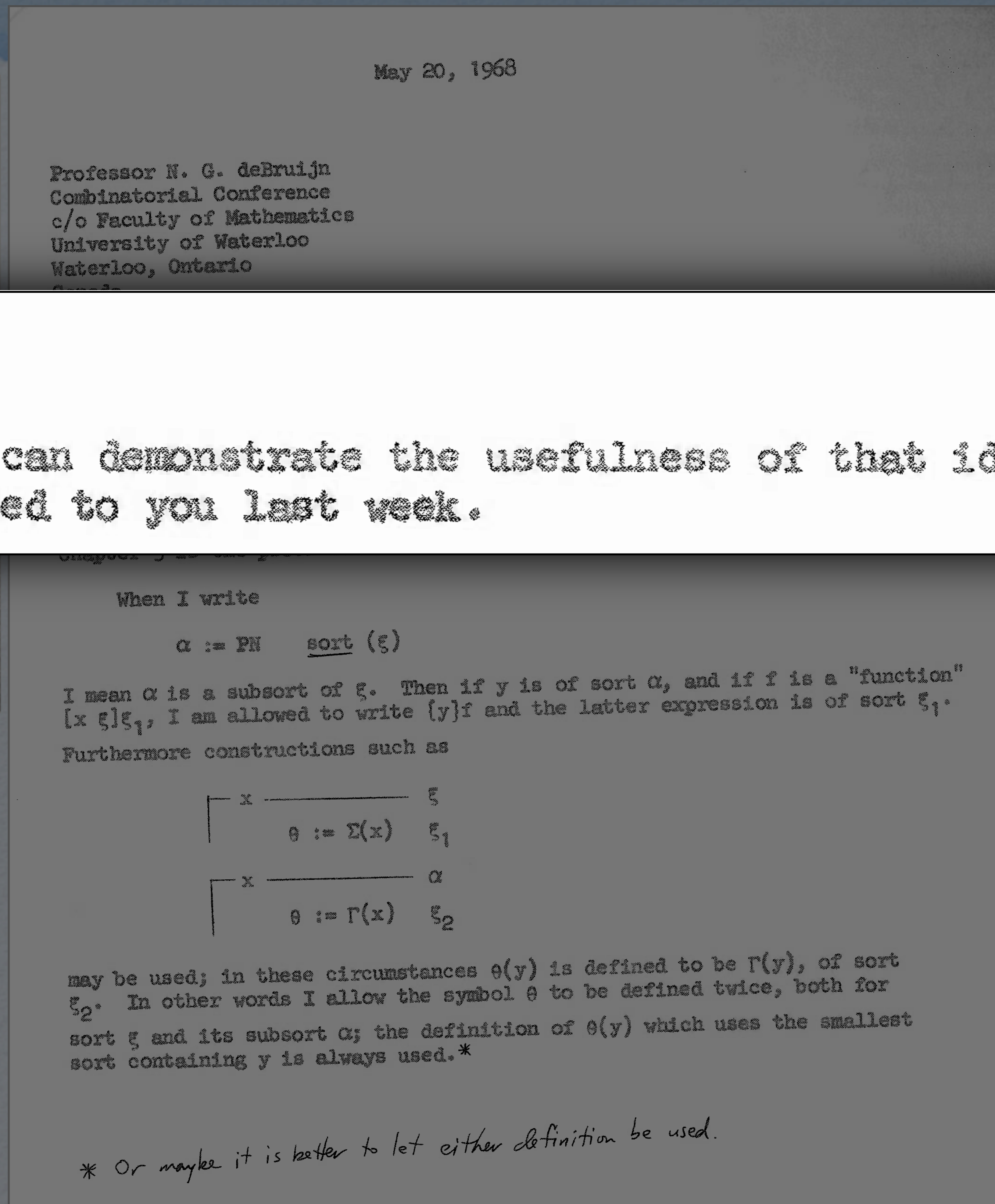
Knuth



Shoulders of Giants



deBruijn



May 20, 1968

Professor N. G. deBruijn
Combinatorial Conference
c/o Faculty of Mathematics
University of Waterloo
Waterloo, Ontario

Dear Dick:

I think I can demonstrate the usefulness of that idea of "subsorts" which I mentioned to you last week.

When I write

α := PN $\underline{\text{sort}}$ (ξ)

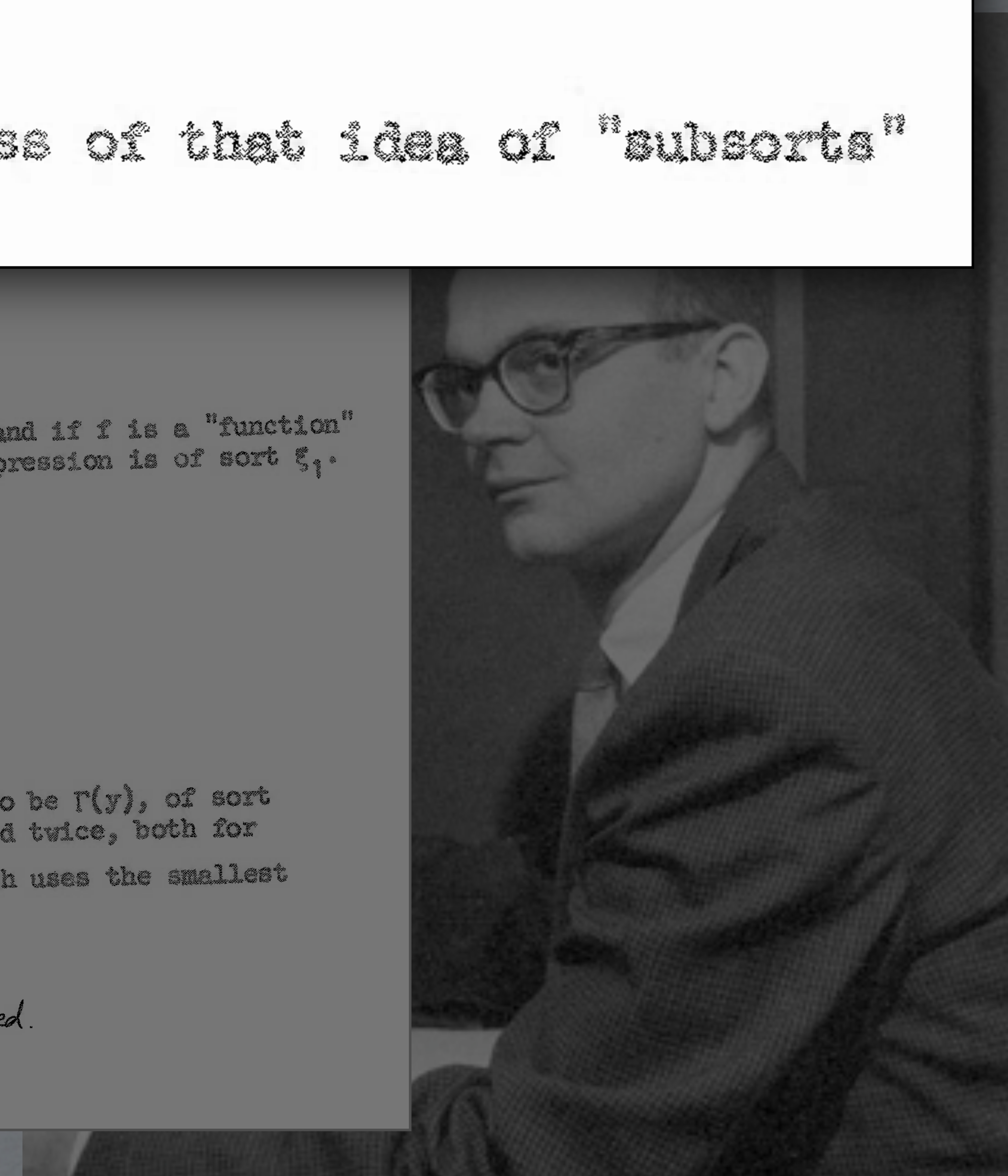
I mean α is a subsort of ξ. Then if y is of sort α, and if f is a "function" $[x\ \xi]\xi_1$, I am allowed to write $\{y\}f$ and the latter expression is of sort ξ_1. Furthermore constructions such as

$$\begin{array}{l|ll} x & \rule{4em}{0.4pt} & \xi \\ & \theta := \Sigma(x) & \xi_1 \end{array}$$

$$\begin{array}{l|ll} x & \rule{4em}{0.4pt} & \alpha \\ & \theta := \Gamma(x) & \xi_2 \end{array}$$

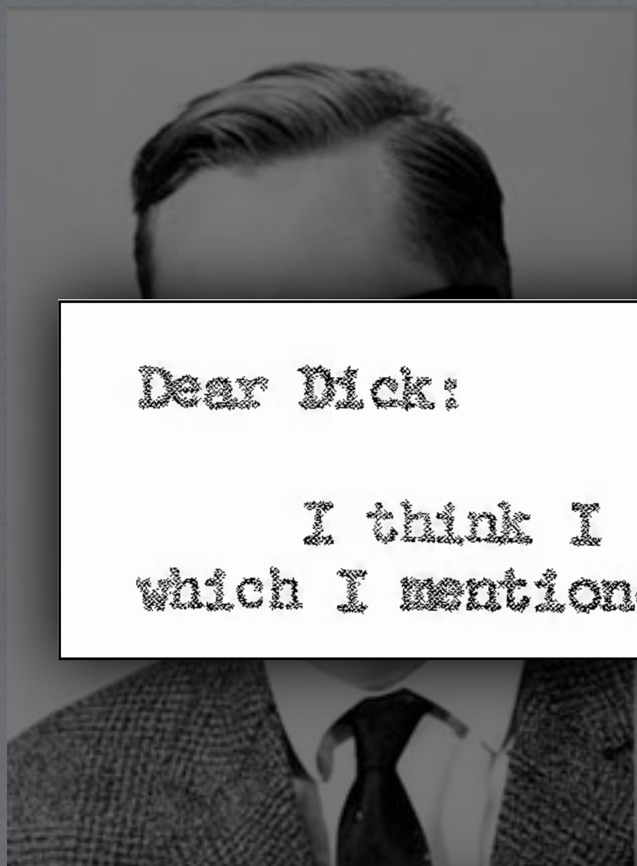
may be used; in these circumstances $\theta(y)$ is defined to be $\Gamma(y)$, of sort ξ_2. In other words I allow the symbol θ to be defined twice, both for sort ξ and its subsort α; the definition of $\theta(y)$ which uses the smallest sort containing y is always used.*

* Or maybe it is better to let either definition be used.



Knuth

Shoulders of Giants



May 20, 1968

Professor N. G. deBruijn
Combinatorial Conference
c/o Faculty of Mathematics
University of Waterloo
Waterloo, Ontario

Knuth

Dear Dick:

I think I can demonstrate the usefulness of that idea of "subsorts" which I mentioned to you last week.

When I write

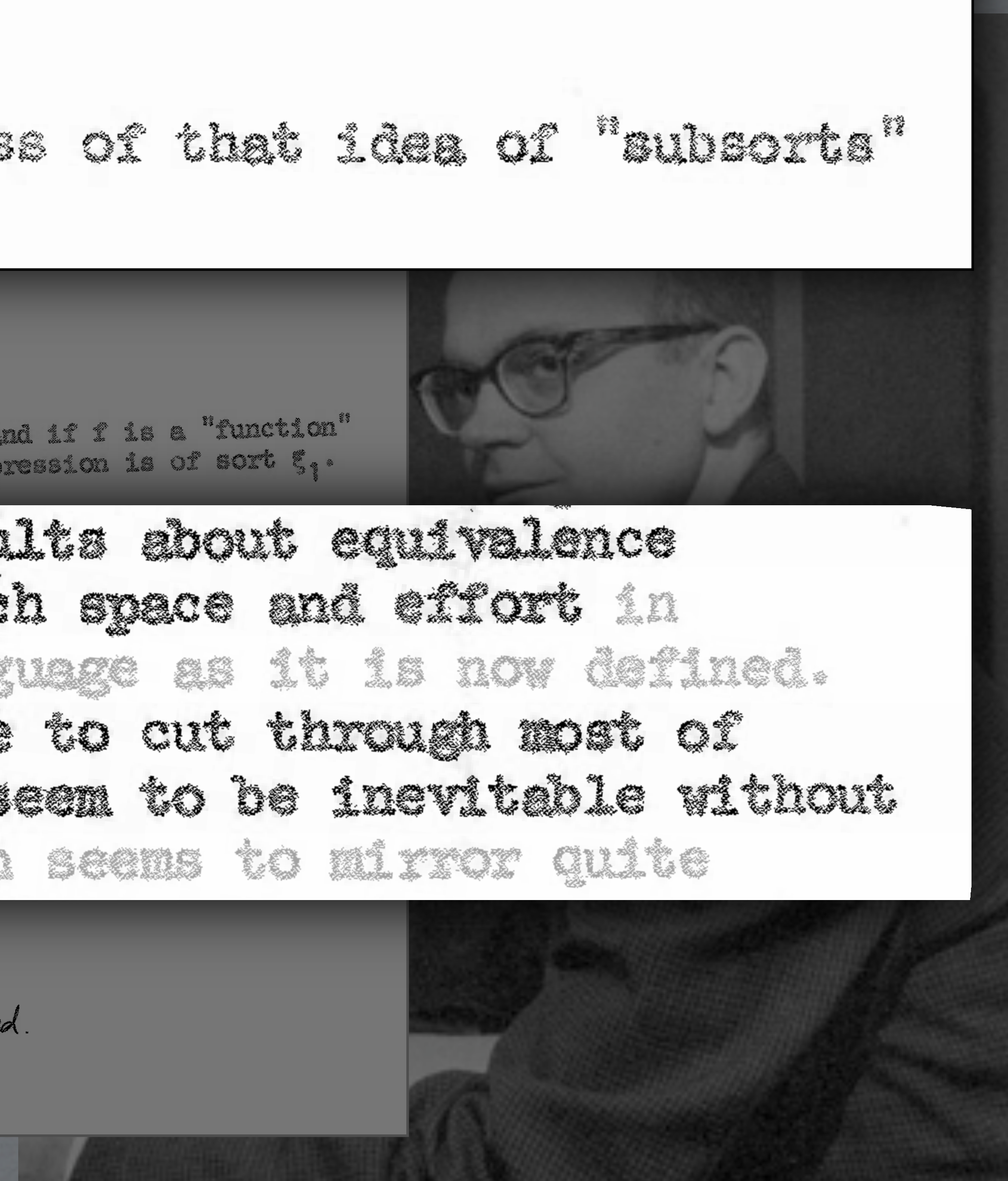
α := PN sort (ξ)

I mean α is a subsort of ξ. Then if y is of sort α, and if f is a "function" [x ξ]ξ_1, I am allowed to write {y}f and the latter expression is of sort ξ_1.

and 2 you will not be able to prove the results about equivalence relations without using about 5 times as much space and effort in Chapter 3, if you work entirely in your language as it is now defined. The use of subsorts makes it possible for me to cut through most of the red tape and the circumlocutions which seem to be inevitable without subsorts. Furthermore the enclosed solution seems to mirror quite

sort containing y is always used.*

* Or maybe it is better to let either definition be used.



Thesis

Refinement types are a **useful** and **practical** extension to the LF logical framework.

Contributions

- Refinements are **useful**:
  - many case studies
  - subset interpretation
- Refinements are **practical**:
  - rich yet simple metatheory
  - sort reconstruction

Outline

- Overview and motivation
- Basic formalism
  - LFR type theory and metatheory
  - Higher-sort subsorting
- Rest of the story
  - Subset interpretation
  - Sort reconstruction
  - Case studies
- Summary

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

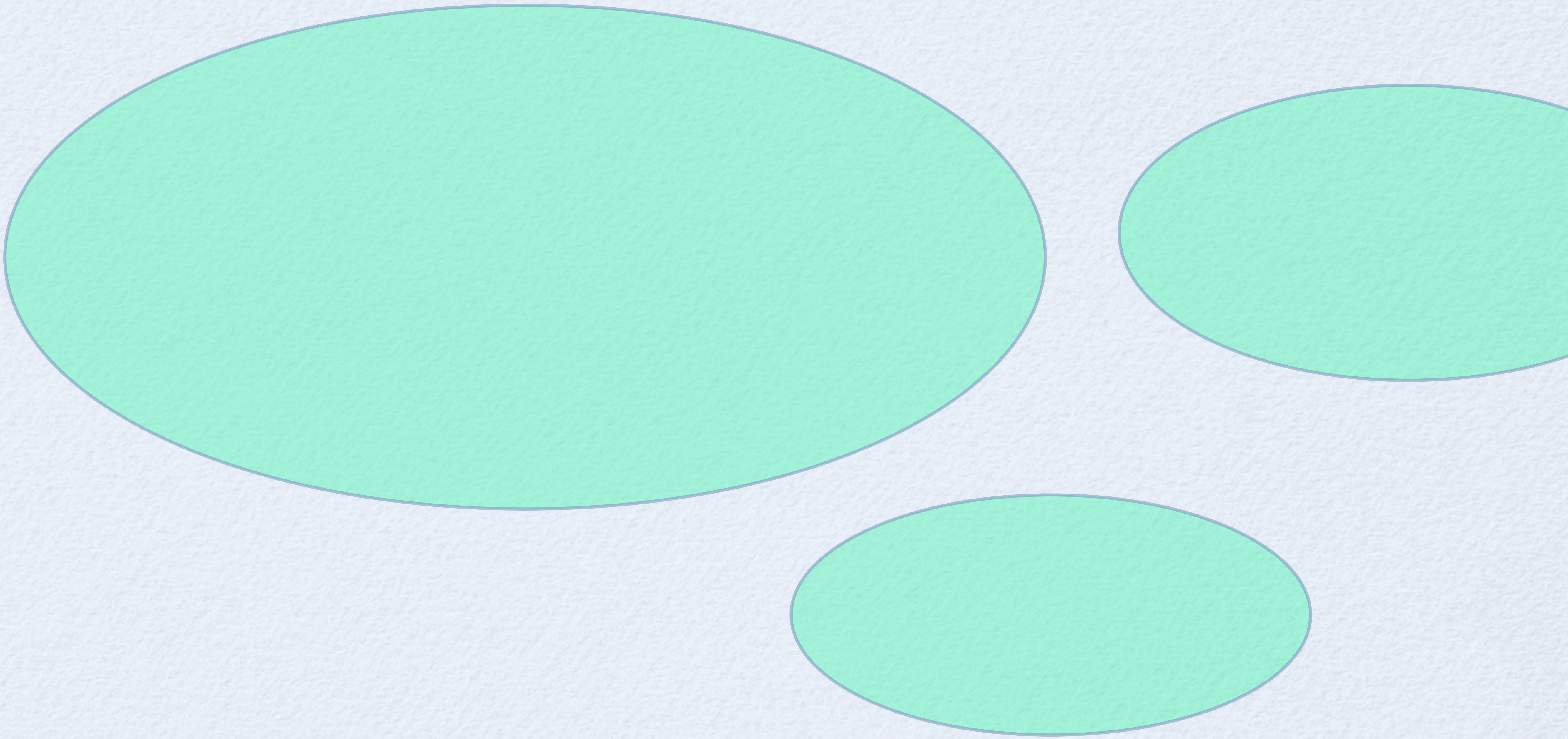
- Harper, Honsell, and Plotkin, 1987, 1993
- Dependently-typed lambda-calculus
- Encode deductive systems and metatheory, uniformly, and machine-checkably
  - *e.g.* a programming language and its type safety theorem
- Guiding principle: *"judgements as types"*

Judgements as types

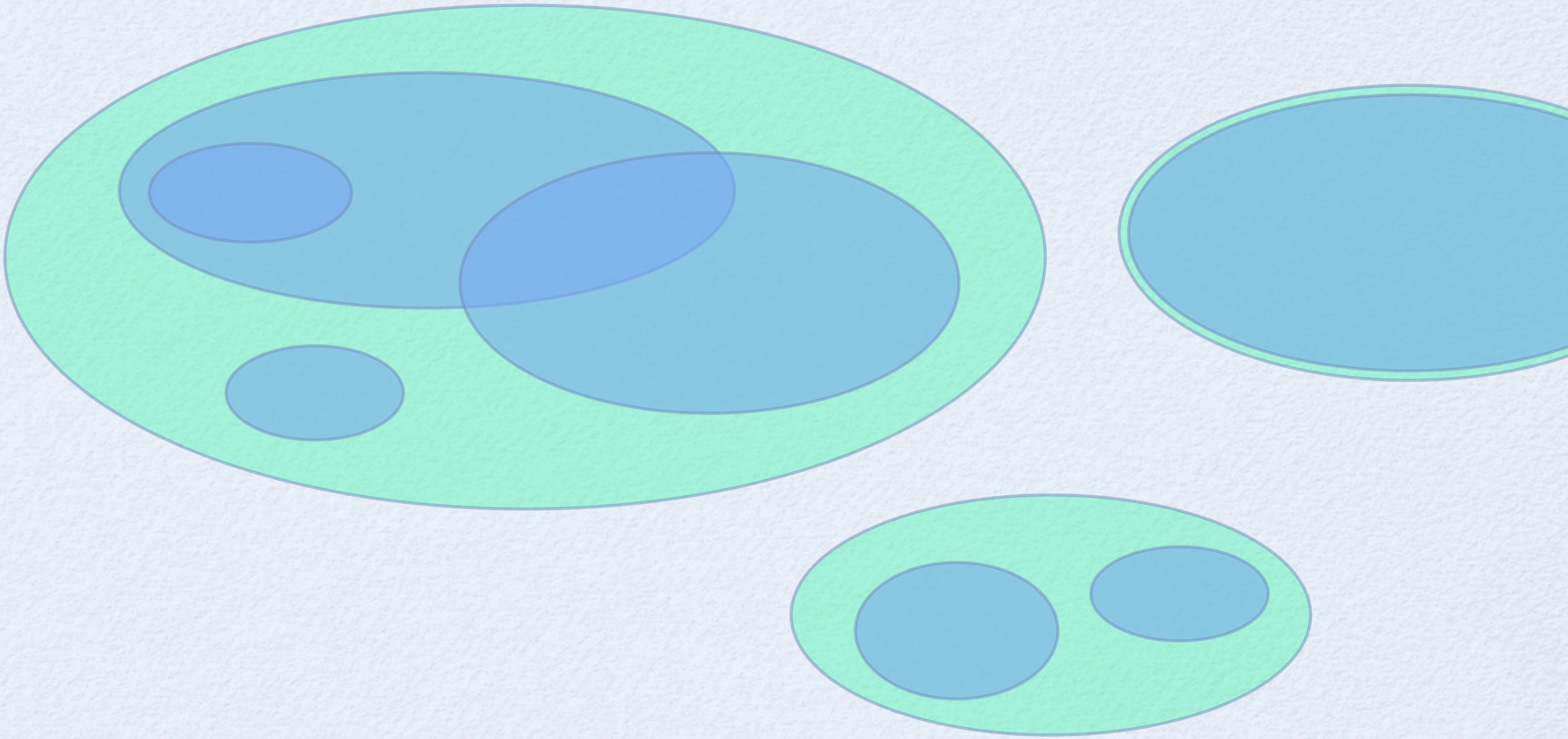
On paper	In LF
Syntax ▸ e ::= … τ ::= …	Simple type ▸ exp : **type**. tp : **type**.
Judgement ▸ $\Gamma \vdash e : \tau$	Type family ▸ of : exp → tp → **type**.
Derivation ▸ $\mathcal{D} :: \Gamma \vdash e : \tau$	Well-typed term ▸ M : of E T
Proof checking	Type checking

Refinement types / Sorts

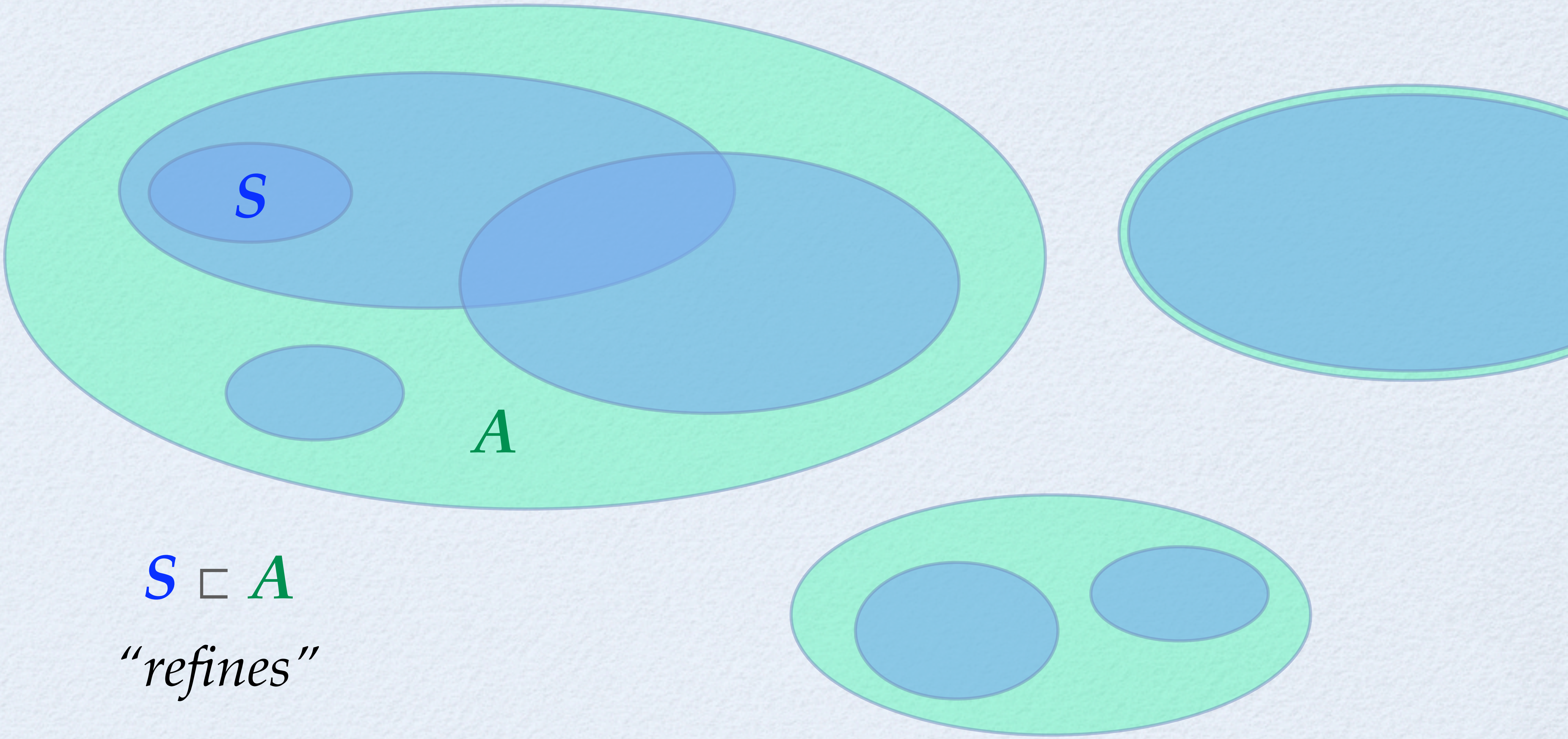
Refinement types / Sorts



Refinement types / Sorts



Refinement types / Sorts



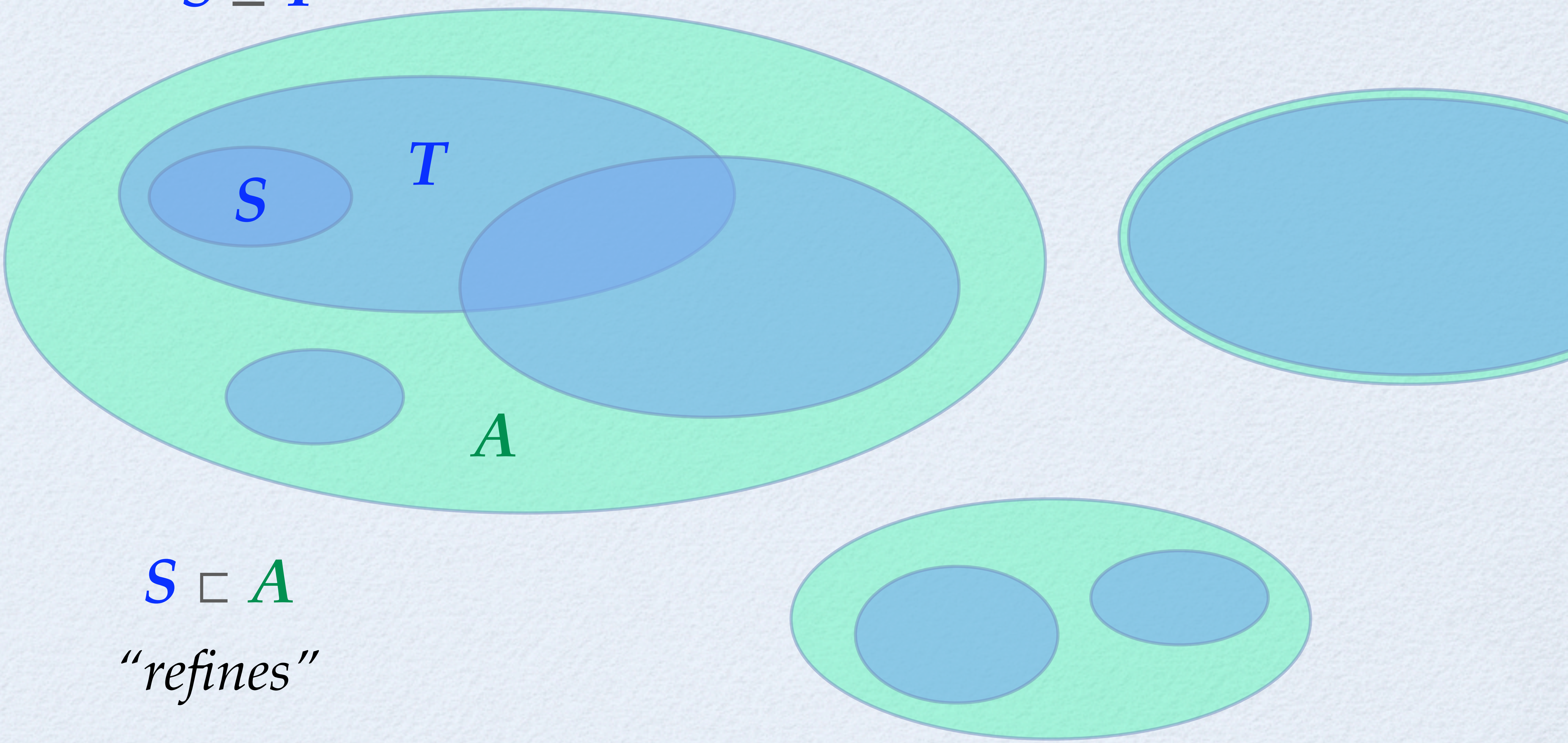
$S \sqsubset A$

"refines"

Refinement types / Sorts

"subsort"

$S \leq T$



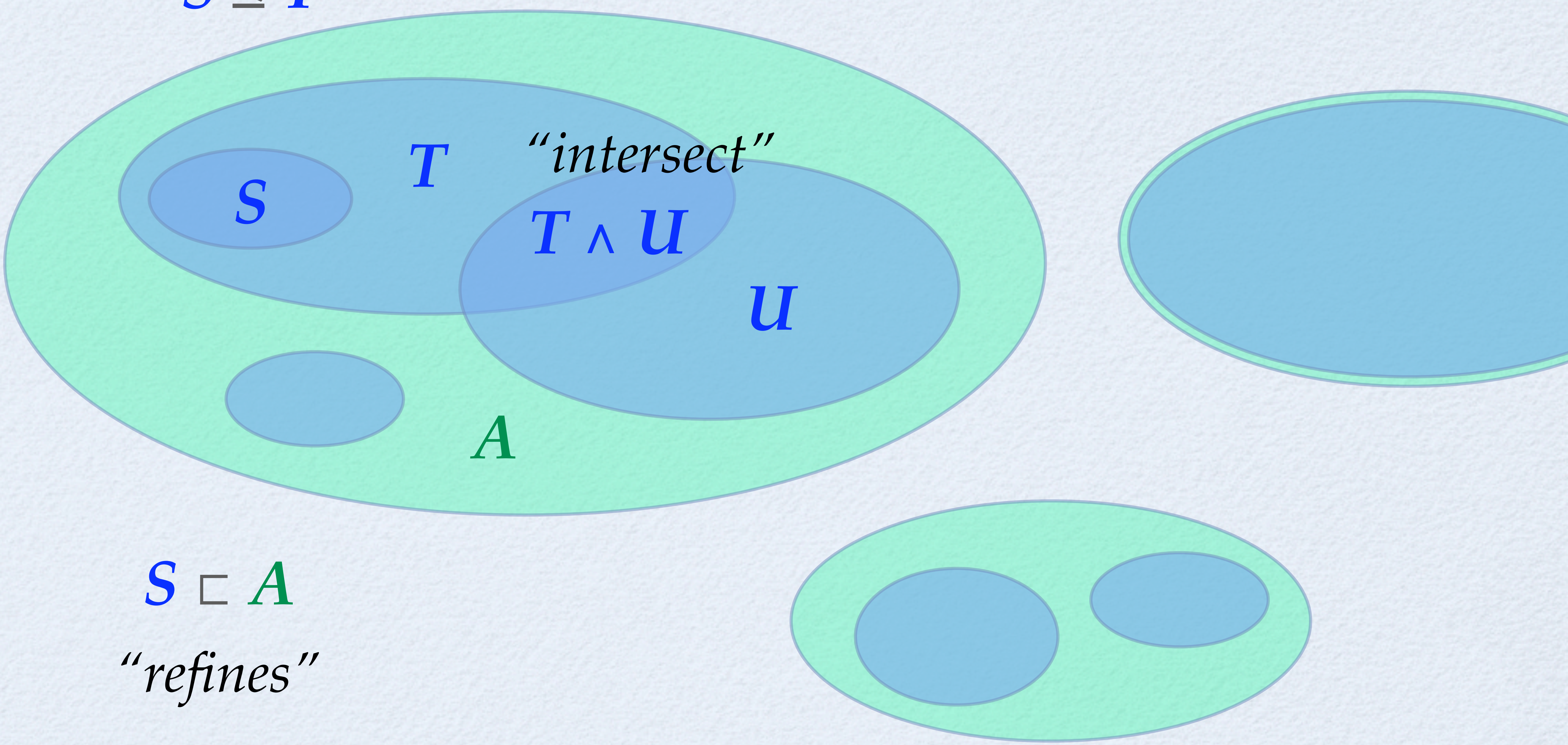
$S \sqsubset A$

"refines"

Refinement types / Sorts

"subsort"

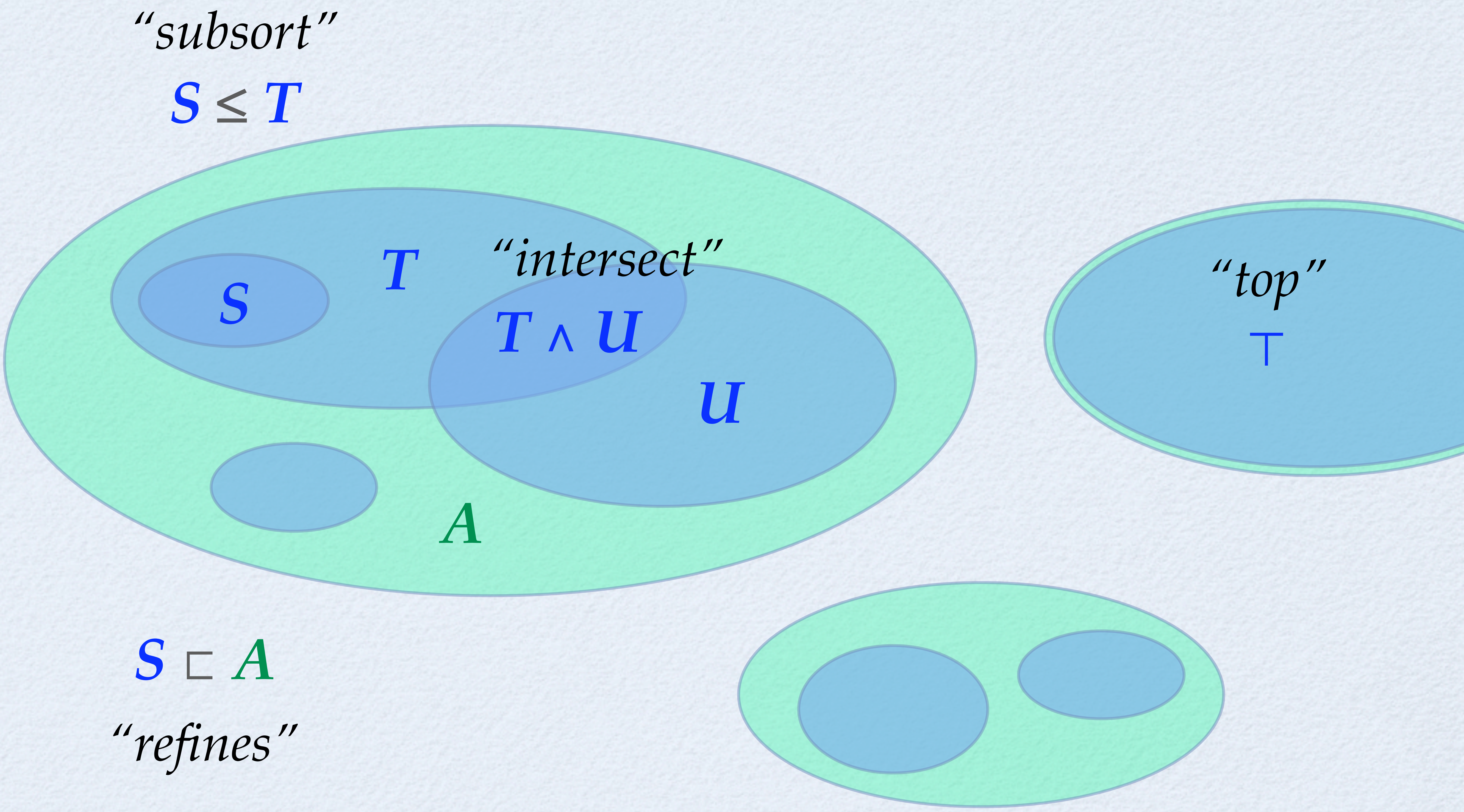
$S \leq T$



$S \sqsubset A$

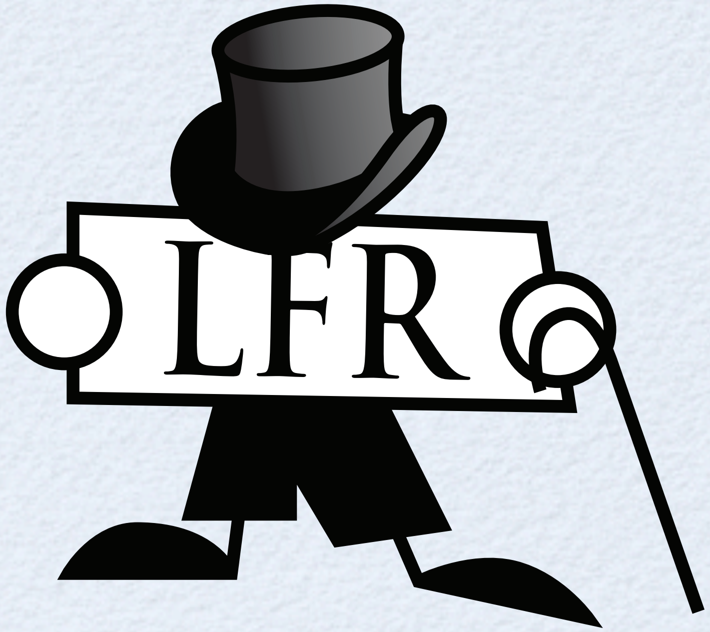
"refines"

Refinement types / Sorts



Properties as sorts

- Even and odd natural numbers,
- Expressions that are values,
- Normal natural deductions,
- Cut-free sequent proofs,
- Derivations without a particular rule,
- Prenex and rank-2 polymorphism,
- …



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**.

always even!

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 *judgments* on natural numbers.

Option 1: explicit proofs

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

(*properties as judgments + judgments as types)*

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 1: explicit proofs

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

Option 2: implicit proofs

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

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 : 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: implicit proofs

- But… need erasures from even and odd to nat

Option 2: implicit proofs

- But… need erasures from even and odd to nat

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

e2n-z_e : even2nat z_e z.
e2n-s_e : even2nat (s_e O) (s N)
    ← odd2nat O N.
o2n-s_o : odd2nat (s_o E) (s N)
    ← even2nat E N.

Option 2: implicit proofs

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

Option 3: metatheorem

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

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).
```

Option 3: metatheorem

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

Better option: refinements

even $\sqsubset$ nat.

odd $\sqsubset$ nat.

z :: even.

s :: even $\rightarrow$ odd $\wedge$ odd $\rightarrow$ 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

	simple	lightweight	direct
1. implicit proofs	✗	✓	✓
2. explicit proofs	✓	✗	✓
3. metatheorem	✓	✓	✗
4. refinements	✓	✓	✓

- **Simple:** doubling judgment doesn't change.
- **Lightweight:** constructors remain the same.
- **Direct:** strong typing guarantee on derivations.

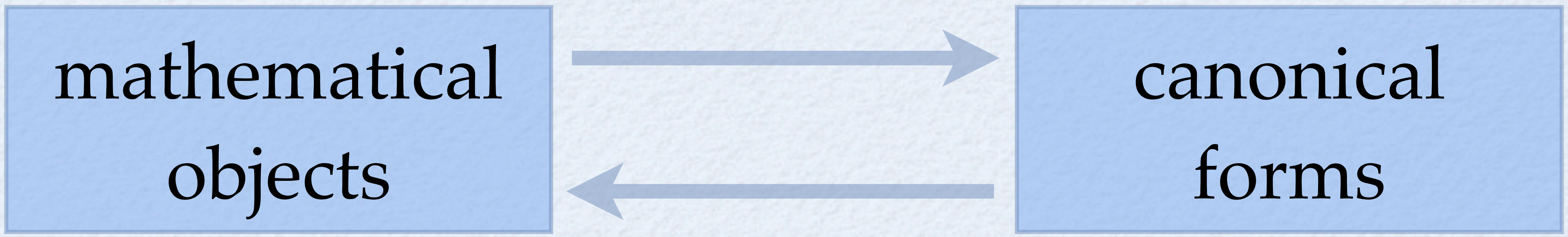
Outline

✓ Overview and Motivation

- Basic formalism
  - LFR type theory and metatheory
  - Higher-sort subsorting
- Rest of the story
  - Subset interpretation
  - Sort reconstruction
  - Case studies
- Summary

Adequacy

- Does my encoding mean anything?
- Strategy: exhibit a *compositional bijection* that *preserves properties.*



  - *"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:

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

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

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 subsorting

$$\Gamma \vdash N \Leftarrow S$$

- Key change:
  - equality becomes subsorting
  - subsorting… only at base sorts?

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

Checking with subsorting

$$\Gamma \vdash N \Leftarrow S$$

- Key change:
  - equality becomes subsorting
  - subsorting… only at base sorts?

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

Intersections

- Similar to product types, but no proof term

$$\frac{\Gamma \vdash N \Leftarrow S_1 \qquad \Gamma \vdash N \Leftarrow S_2}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2} \qquad \frac{}{\Gamma \vdash N \Leftarrow \top}$$

$$\frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_1} \qquad \frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_2}$$

Important principles

- **Substitution:**

  if $\Gamma, x{::}S \sqsubset A \vdash N \Leftarrow T$ and $\Gamma \vdash M \Leftarrow S$,

  then $\Gamma \vdash [M/x]_A\, N \Leftarrow T$.

- **Identity:** for all A: $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow S$.

Subsorting

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

- Key rule:

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

  - Bidirectional: subsorting only at mode switch
  - Canonical: mode switch only at base sort

Subsorting at higher sorts?

- Structural rules? e.g.

$$\frac{S_2 \leq S_1 \qquad T_1 \leq T_2}{S_1 \rightarrow T_1 \;\leq\; S_2 \rightarrow T_2}$$

- Distributivity?

$$\frac{}{(S \rightarrow T_1) \wedge (S \rightarrow T_2) \;\leq\; S \rightarrow (T_1 \wedge T_2)}$$

Subsorting at higher sorts!

- **Intrinsic subsorting:**
  if $S \leq T$ and $\Gamma \vdash N \Leftarrow S$, then $\Gamma \vdash N \Leftarrow T$.

Subsorting at higher sorts!

- **Intrinsic subsorting:**
  if $S \leq T$ and $\Gamma \vdash N \Leftarrow S$, then $\Gamma \vdash N \Leftarrow T$.

- **Equivalently:**
  if $S \leq T$, then $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$.

Subsorting at higher sorts!

- **Intrinsic subsorting:**
  if $S \leq T$ and $\Gamma \vdash N \Leftarrow S$, then $\Gamma \vdash N \Leftarrow T$.

- **Equivalently:**
  if $S \leq T$, then $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$.
  - just like the Identity principle!

Subsorting at higher sorts!

- **Intrinsic subsorting:**
  if $S \leq T$ and $\Gamma \vdash N \Leftarrow S$, then $\Gamma \vdash N \Leftarrow T$.

- **Equivalently:**
  if $S \leq T$, then $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$.
  - just like the Identity principle!
  - … also the Substitution principle …

Subsorting at higher sorts!

- **Intrinsic subsorting:**
  if $S \leq T$ and $\Gamma \vdash N \Leftarrow S$, then $\Gamma \vdash N \Leftarrow T$.

- **Equivalently:**
  if $S \leq T$, then $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$.
  - just like the Identity principle!
  - … also the Substitution principle …

- Usual rules all *sound* in this sense.

Subsorting at higher sorts?

- … and also *complete*!
- **Theorem:** if $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$, then $S \leq T$.
- **Or:** if $\Gamma \vdash N \Leftarrow S$ implies $\Gamma \vdash N \Leftarrow T$, then $S \leq T$.

Subsorting at higher sorts?

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

Outline

- ✓ Introduction: Motivation
- ✓ Basic formalism
  - LFR type theory and metatheory
  - Higher-sort subsorting
- Rest of the story
  - Subset interpretation
  - Sort reconstruction
  - Case studies
- Summary

Subset Interpretation

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

Subset Interpretation

- *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!

Sorts as predicates

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

Sorts as predicates

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

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

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).

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

Sorts as predicates

nat : **typ**
z : nat.
s : nat →

- Translation follows th
  - **refineme** … *…icates*
  - … …ns become *proof constructors*

(One twist: *proof irrelevance*)

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

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

Sorts as predicates

$\text{nat} : \textbf{type}.$
$\text{z} : \text{nat}.$
$\text{s} : \text{nat} \rightarrow \text{nat}.$

$\text{even} \sqsubset \text{nat}.$
$\text{odd} \sqsubset \text{nat}.$
$\text{z} :: \text{even}.$
$\text{s} :: \text{even} \rightarrow \text{odd}$
$\quad \wedge\ \text{odd} \rightarrow \text{even}.$

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

Sorts as predicates

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

N :: even iff *M* : even *N*
(for some *M*)

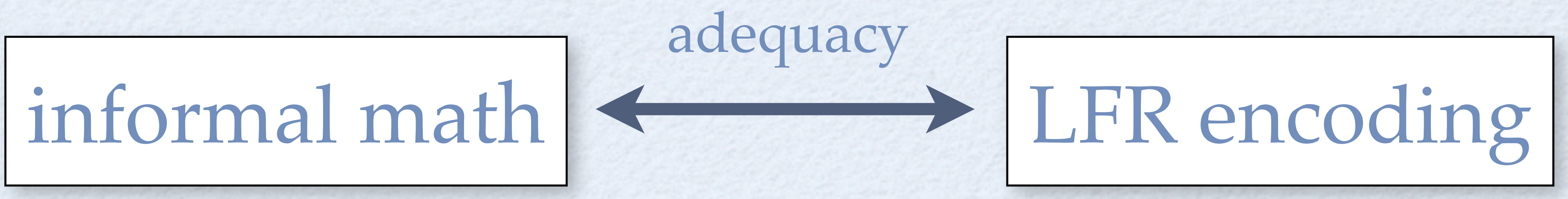
even ⊏ nat.
odd ⊏ nat.
z :: even.
s :: even → odd
∧ odd → even.

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

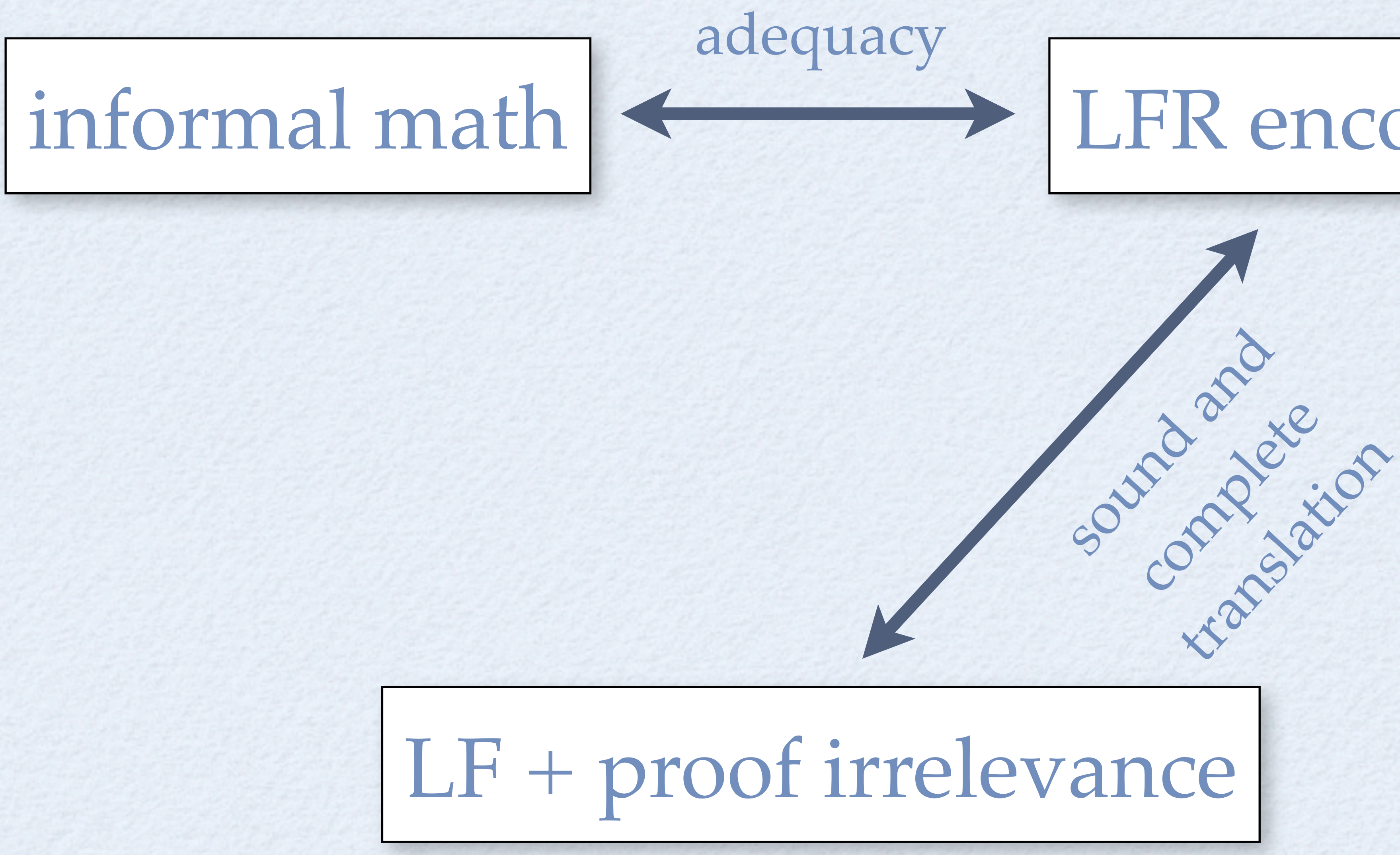
Adequacy

- LF enjoys a well-developed theory of adequate representations
    - **Adequacy**: compositional, property-preserving bijection between informal entities and canonical terms

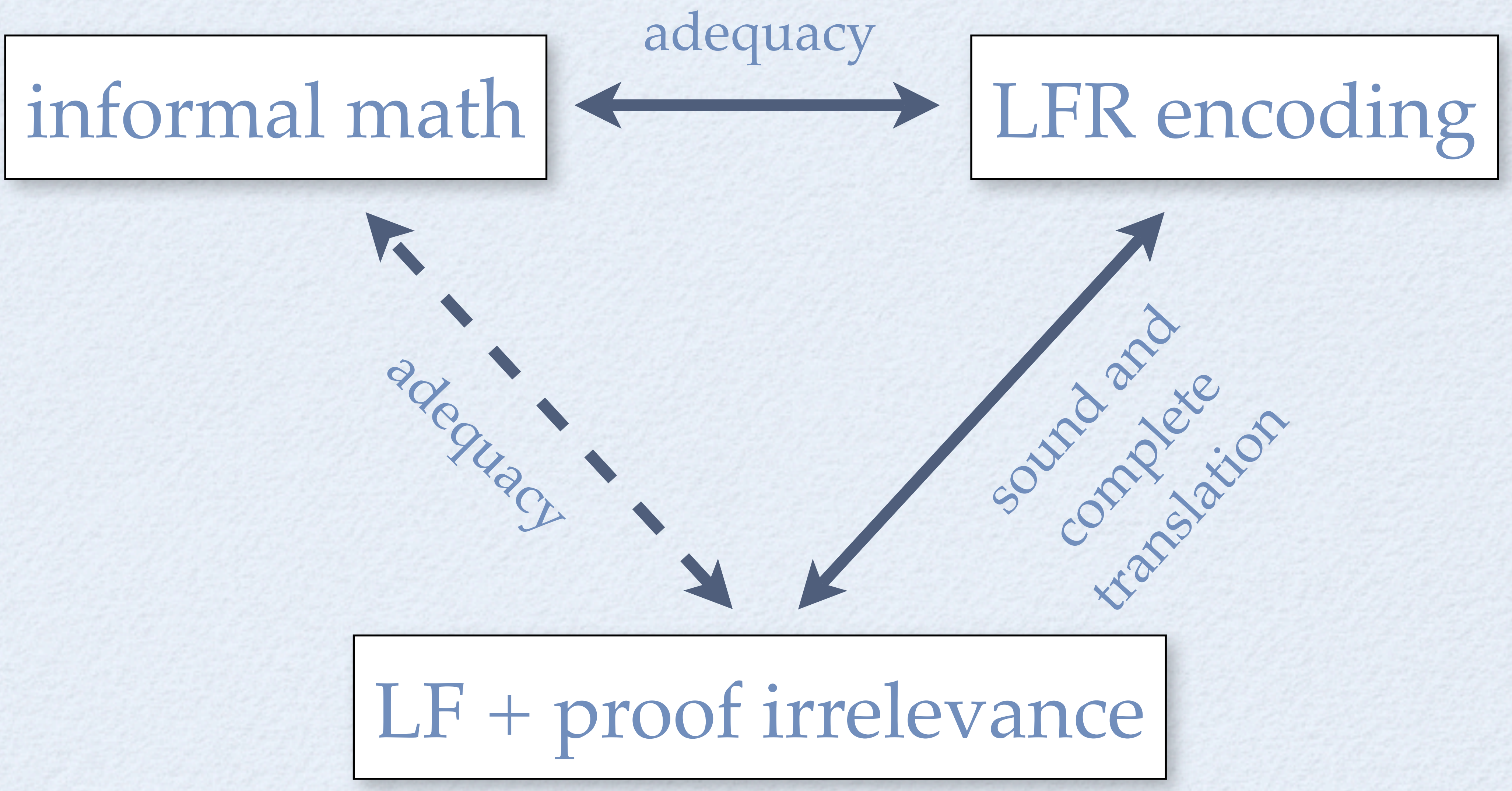
Adequacy



Adequacy


informal math
adequacy
LFR encoding
sound and complete translation
LF + proof irrelevance

Adequacy


informal math
adequacy
LFR encoding
adequacy
sound and
complete
translation
LF + proof irrelevance

Sort Reconstruction

- Three phases:
  - **LFR Type Reconstruction**: reconstruct implicit arguments and types of subterms by matching.
  - **Constraint generation**: reduce a sort-checking problem to a constraint.
  - **Constraint solving**: solve that constraint.

Sort Reconstruction

double : nat → nat → **type**.
dbl/s : double *N* *N2* → double (s *N*) (s (s *N2*)).

Sort Reconstruction

double : nat → nat → **type**.
dbl/s : double *N* *N2* → double (s *N*) (s (s *N2*)).

↓ LF type reconstruction

double : nat → nat → **type**.
dbl/s : Π*N*:nat. Π*N2*:nat.
double *N* *N2* → double (s *N*) (s (s *N2*)).

Sort Reconstruction

double : nat → nat → **type**.
dbl/s : Π*N*:nat. Π*N2*:nat.
double *N* *N2* → double (s *N*) (s (s *N2*)).

Sort Reconstruction

double : nat → nat → **type**.
dbl/s : Π*N*:nat. Π*N2*:nat.
    double *N* *N2* → double (s *N*) (s (s *N2*)).

Sort Reconstruction

double : nat → nat → **type**.

dbl/s : ΠN:nat. Π$N2$:nat.

double N $N2$ → double (s N) (s (s $N2$)).

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

dbl/s :: double* N $N2$ → double* (s N) (s (s $N2$)).

Sort Reconstruction

double : nat → nat → **type**.
dbl/s : ΠN:nat. Π$N2$:nat.
    double N $N2$ → double (s N) (s (s $N2$)).

double* ⊏ double :: ⊤ → even → **type**.
dbl/s :: double* N $N2$ → double* (s N) (s (s $N2$)).

↓ LFR type reconstruction

double* ⊏ double :: ⊤ → even → **type**.
dbl/s :: ΠN::σ⊏nat. Π$N2$::σ_2⊏nat.
    double* N $N2$ → double* (s N) (s (s $N2$)).

Sort Reconstruction

$\text{double*} \sqsubset \text{double} :: \top \to \text{even} \to \textbf{type}.$

$\text{dbl/s} :: \Pi N{::}\sigma{\sqsubset}\text{nat}.\ \Pi N2{::}\sigma_2{\sqsubset}\text{nat}.$

$\quad\quad \text{double*}\ N\ N2 \to \text{double*}\ (\text{s}\ N)\ (\text{s}\ (\text{s}\ N2)).$

Sort Reconstruction

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

dbl/s :: Π*N*::σ⊏nat. Π*N2*::σ_2⊏nat.

double* *N* *N2* → double* (s *N*) (s (s *N2*)).

Sort Reconstruction

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

dbl/s :: Π*N*::σ⊏nat. Π*N2*::σ_2⊏nat.

double* *N* *N2* → double* (s *N*) (s (s *N2*)).

↓ Constraint generation

$\sigma_2 \leq$ even

Sort Reconstruction

double* $\sqsubset$ double :: $\top$ → even → **type**.

dbl/s :: Π*N*::σ⊏nat. Π*N2*::σ_2⊏nat.

    double* *N* *N2* → double* (s *N*) (s (s *N2*)).

↓ Constraint generation

$\sigma_2 \leq$ even

↓ Constraint solving

double* $\sqsubset$ double :: $\top$ → even → **type**.

dbl/s :: Π*N*::⊤⊏nat. Π*N2*::even⊏nat.

    double* *N* *N2* → double* (s *N*) (s (s *N2*)).

Sort Reconstruction

- **Theorem (Soundness):** result of reconstruction is well-formed
- **Theorem (Principality):** any other possible reconstruction is less general

Case Studies

Case Study 1: normal forms

Definition 3.1 (*Weak-Head Normal*)

$\mathrm{T}_\star$, $A_1 \rightarrow A_2$, $\forall X \leq A{:}K.B$, and $\Lambda X \leq A{:}K.B$ are weak head normal.

$X(A_1, \ldots, A_n)$ is weak head normal if $A_1, \ldots, A_n$ are in normal form.

Case Study 1: normal forms

DEFINITION 3.1 (*Weak-Head Normal*)

$\mathrm{T}_\star$, $A_1 {\rightarrow} A_2$, $\forall X {\leq} A{:}K.B$, and $\Lambda X {\leq} A{:}K.B$ are weak head normal.

$X(A_1, \ldots, A_n)$ is weak head normal if $A_1, \ldots, A_n$ are in normal form.

(Compagnoni and Goguen, *Typed Operational Semantics for Higher-Order* Subtyping, Inf. & Comput. 2003)

Case Study 1: normal forms

DEFINITION 3.1 (*Weak-Head Normal*)

$\mathrm{T}_\star$, $A_1 \rightarrow A_2$, $\forall X \leq A{:}K.B$, and $\Lambda X \leq A{:}K.B$ are weak head normal.

$X(A_1, \ldots, A_n)$ is weak head normal if $A_1, \ldots, A_n$ are in normal form.

(Compagnoni and Goguen, *Typed Operational Semantics for Higher-Order* Subtyping, Inf. & Comput. 2003)

- 3 steps:
  - translate grammar of types
  - characterize normal types
  - characterize weak head normal types

Case Study 1: normal forms

- $A ::= X \mid A \rightarrow A \mid \forall X{\leq}A{:}K.\ A \mid \Lambda X{\leq}A{:}K.\ A \mid A\ A \mid \mathrm{T}_\star$

```
kd : type.
tp : type.

T* : tp.
arrow : tp → tp → tp.
all : tp → kd → (tp → tp) → tp.
Lam : tp → kd → (tp → tp) → tp.
App : tp → tp → tp.
```

Case Study 1: normal forms

- $A ::= P \mid A \to A \mid \forall X{\leq}A{:}K.\ A \mid \Lambda X{\leq}A{:}K.\ A \mid T_{\star}$

  $P ::= X \mid P\ A$

```
btp ⊏ tp.  ntp ⊏ tp.

T* :: ntp.
arrow :: ntp → ntp → ntp.
all :: ntp → ⊤ → (btp → ntp) → ntp.
Lam :: ntp → ⊤ → (btp → ntp) → ntp.
App :: btp → ntp → btp.
btp ≤ ntp.
```

Case Study 1: normal forms

Definition 3.1 (*Weak-Head Normal*)

$T_\star$, $A_1 \to A_2$, $\forall X \leq A{:}K.B$, and $\Lambda X \leq A{:}K.B$ are weak head normal.

$X(A_1, \ldots, A_n)$ is weak head normal if $A_1, \ldots, A_n$ are in normal form.

```
whntp ⊏ tp.

T* :: whntp.
arrow :: ⊤ → ⊤ → whntp.
all :: ⊤ → ⊤ → (btp → ⊤) → whntp.
Lam :: ⊤ → ⊤ → (btp → ⊤) → whntp.
btp ≤ whntp.
```

Case Study 2: CBV/CBN

3.2.1 Call-By-Value (CBV) Strategy

The standard call-by-value strategy is defined as follows:

$$
\begin{array}{lcl}
V & ::= & x \mid \lambda x{:}A.M \mid \Lambda u{:}K.M \\
R & ::= & (\lambda x{:}A.M)\,V \mid (\Lambda u{:}K.M)\{A\} \mid \\
 & & abort_A(M) \mid callcc_A(M) \\
E & ::= & [] \mid E\,M \mid V\,E \mid E\{A\}
\end{array}
$$

3.2.2 Call-By-Name (CBN) Strategy

The standard call-by-name strategy is defined as follows:

$$
\begin{array}{lcl}
V & ::= & \lambda x{:}A.M \mid \Lambda u{:}K.M \\
R & ::= & (\lambda x{:}A.M_1)\,M_2 \mid (\Lambda u{:}K.M)\{A\} \mid \\
 & & abort_A(M) \mid callcc_A(M) \\
E & ::= & [] \mid E\,M \mid E\{A\}
\end{array}
$$

Case Study 2: CBV/CBN

3.2.1 Call-By-Value (CBV) Strategy

The standard call-by-value strategy is defined as follows:

$$\begin{array}{lcl} V & ::= & x \mid \lambda x{:}A.M \mid \Lambda u{:}K.M \\ R & ::= & (\lambda x{:}A.M)\,V \mid (\Lambda u{:}K.M)\{A\} \mid \\ & & \mathit{abort}_A(M) \mid \mathit{callcc}_A(M) \\ E & ::= & [] \mid E\,M \mid V\,E \mid E\{A\} \end{array}$$

3.2.2 Call-By-Name (CBN) Strategy

The standard call-by-name strategy is defined as follows:

$$\begin{array}{lcl} V & ::= & \lambda x{:}A.M \mid \Lambda u{:}K.M \\ R & ::= & (\lambda x{:}A.M_1)\,M_2 \mid (\Lambda u{:}K.M)\{A\} \mid \\ & & \mathit{abort}_A(M) \mid \mathit{callcc}_A(M) \\ E & ::= & [] \mid E\,M \mid E\{A\} \end{array}$$

(Harper and Lillibridge, *Explicit Polymorphism and CPS Conversion*, POPL 1993)

Case Study 2: CBV/CBN

Definition 2.1 (Syntax)

Kinds	K	$::=$	$\Omega \mid K_1 \Rightarrow K_2$
Constructors	A	$::=$	$\alpha \mid u \mid A_1 \rightarrow A_2 \mid \forall u{:}K.A \mid$
			$\lambda u{:}K.A \mid A_1\ A_2$
Terms	M	$::=$	$x \mid \lambda x{:}A.M \mid M_1\ M_2 \mid$
			$\Lambda u{:}K.M \mid M\{A\} \mid$
			$callcc_A(M) \mid abort_A(M)$

kd : **type**. tp : **type**. tm : **type**.

lam : tp → (tm → tm) → tm.

app : tm → tm → tm.

Lam : kd → (tp → tm) → tm.

App : tm → tp → tm.

callcc : tp → tm → tm.

abort : tp → tm → tm.

evctx : **type**.

<> : evctx.

capp1 : evctx → tm → evctx.

capp2 : tm → evctx → evctx.

cLam : kd → (tp → evctx) → evctx.

cApp : evctx → tp → evctx.

Case Study 2: CBV/CBN

- Useful observations about redexes:
  - need to recognize lambdas for redexes
  - control operators are always redexes

```
lambda ⊏ tm.  Lambda ⊏ tm.          control ⊏ tm.

lam :: ⊤ → (⊤ → ⊤) → lambda.        abort :: ⊤ → ⊤ → control.
Lam :: ⊤ → (⊤ → ⊤) → Lambda.        callcc :: ⊤ → ⊤ → control.
```

Case Study 2: CBV/CBN

3.2.2 Call-By-Name (CBN) Strategy

The standard call-by-name strategy is defined as follows:

$$
\begin{array}{lcl}
V & ::= & \lambda x{:}A.M \mid \Lambda u{:}K.M \\
R & ::= & (\lambda x{:}A.M_1)\; M_2 \mid (\Lambda u{:}K.M)\{A\} \mid \\
 & & abort_A(M) \mid callcc_A(M) \\
E & ::= & [] \mid E\; M \mid E\{A\}
\end{array}
$$

n/val, n/red ⊏ tm.

lambda ≤ n/val.
Lambda ≤ n/val.
app :: lambda → ⊤ → n/red.
App :: Lambda → ⊤ → n/red.
control ≤ n/red.

n/evctx ⊏ evctx.

<> :: n/evctx.
capp1 :: n/evctx → ⊤ → n/evctx.
cApp :: n/evctx → ⊤ → n/evctx.

Case Study 2: CBV/CBN

3.2.1 Call-By-Value (CBV) Strategy

The standard call-by-value strategy is defined as follows:

$$
\begin{array}{lll}
V & ::= & x \mid \lambda x{:}A.M \mid \Lambda u{:}K.M \\
R & ::= & (\lambda x{:}A.M)\,V \mid (\Lambda u{:}K.M)\{A\} \mid \\
 & & abort_A(M) \mid callcc_A(M) \\
E & ::= & [] \mid E\,M \mid V\,E \mid E\{A\}
\end{array}
$$

v/val, v/red, v/lambda ⊏ tm.

lam :: ⊤ → (v/val → ⊤) → v/lambda.
v/lambda ≤ v/val.
Lambda ≤ v/val.
app :: v/lambda → v/val → v/red.
App :: Lambda → ⊤ → v/red.
control ≤ v/red.

v/evctx ⊏ evctx.

<> :: v/evctx.
capp1 :: v/evctx → ⊤ → v/evctx.
capp2 :: v/val → v/evctx → v/evctx.
cApp :: v/evctx → ⊤ → v/evctx.

Case Study 3: singletons

2.2 A Singleton-Free System

To formalize our results, we also require a singleton-free target language into which to translate expressions from the singleton calculus. We will define the singleton-free system in terms of its differences from the singleton calculus.

We will say that a constructor c (not necessarily well-formed) syntactically belongs to the singleton-free calculus provided that c contains no singleton kinds. Note that as a consequence of containing no singleton kinds, all product and sum kinds may be written in non-dependent form. Also, all kinds in the singleton-free calculus are well-formed.

The inference rules for the singleton-free system are obtained by removing from the singleton calculus all the rules dealing with subkinding (Rules 9–13, 28 and 45) and all the rules dealing with singleton kinds (Rules 6, 15, 25, 34 and 35). Note that derivable judgements in the singleton-free system must be built using only expressions syntactically belonging to the singleton-free calculus. When a judgement is derivable in the singleton-free system, we will note this fact by marking the turnstile $\vdash_{sf}$.

Case Study 3: singletons

2.2 A Singleton-Free System

To formalize our results, we also require a singleton-free target language into which to translate expressions from the singleton calculus. We will define the singleton-free system in terms of its differences from the singleton calculus.

We will say that a constructor c (not necessarily well-formed) syntactically belongs to the singleton-free calculus provided that c contains no singleton kinds. Note that as a consequence of containing no singleton kinds, all product and sum kinds may be written in non-dependent form. Also, all kinds in the singleton-free calculus are well-formed.

The inference rules for the singleton-free system are obtained by removing from the singleton calculus all the rules dealing with subkinding (Rules 9–13, 28 and 45) and all the rules dealing with singleton kinds (Rules 6, 15, 25, 34 and 35). Note that derivable judgements in the singleton-free system must be built using only expressions syntactically belonging to the singleton-free calculus. When a judgement is derivable in the singleton-free system, we will note this fact by marking the turnstile $\vdash_{sf}$.

(Crary, *Sound and Complete Elimination of Singleton Kinds,* ACM TOCL 2007)

Case Study 3: singletons

kinds	K	$::=$	$T \mid S(c) \mid \Pi\alpha{:}K_1.K_2 \mid \Sigma\alpha{:}K_1.K_2$
constructors	c	$::=$	$\alpha \mid b \mid \lambda\alpha{:}K.c \mid c_1 c_2 \mid \langle c_1, c_2 \rangle \mid \pi_1 c \mid \pi_2 c$
assignments	Γ	$::=$	$\epsilon \mid \Gamma, \alpha{:}K$

Fig. 1. Syntax

kd : **type**. tp : **type**.

t : kd.
sing : tp → kd.
pi : kd → (tp → kd) → kd.
sigma : kd → (tp → kd) → kd.

Case Study 3: singletons

kinds	K ::=	$T \mid S(c) \mid \Pi\alpha{:}K_1.K_2 \mid \Sigma\alpha{:}K_1.K_2$
constructors	c ::=	$\alpha \mid b \mid \lambda\alpha{:}K.c \mid c_1 c_2 \mid \langle c_1, c_2\rangle \mid \pi_1 c \mid \pi_2 c$
assignments	Γ ::=	$\epsilon \mid \Gamma, \alpha{:}K$

Fig. 1. Syntax

```
sf/kd ⊏ kd.  sf/tp ⊏ tp.

t : sf/kd.
% no: sing : sf/tp → sf/kd.
pi : sf/kd → (sf/tp → sf/kd) → sf/kd.
sigma : sf/kd → (sf/tp → sf/kd) → sf/kd.
```

Case Study 3: singletons

Kind Equivalence $\boxed{\Gamma \vdash K_1 = K_2}$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash T = T} \quad (14)$$

$$\frac{\Gamma \vdash c_1 = c_2 : T}{\Gamma \vdash S(c_1) = S(c_2)} \quad (15)$$

$$\frac{\Gamma \vdash K_2' = K_1' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' = K_2''}{\Gamma \vdash \Pi\alpha{:}K_1'.K_1'' = \Pi\alpha{:}K_2'.K_2''} \quad (16)$$

$$\frac{\Gamma \vdash K_1' = K_2' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' = K_2''}{\Gamma \vdash \Sigma\alpha{:}K_1'.K_1'' = \Sigma\alpha{:}K_2'.K_2''} \quad (17)$$

```
keq : kd → kd → type.
eq : tp → tp → kd → type.
kof : tp → kd → type.

r14 : keq t t.

r15 : keq (sing C1) (sing C2)
    ← eq C1 C2 t.

r16 : keq (pi K1’ [a] K1” a)
          (pi K2’ [a] K2” a)
    ← keq K2’ K1’
    ← ({a} kof a K1’
        → keq (K1” a) (K2” a)).
```

Case Study 3: singletons

Kind Equivalence $\boxed{\Gamma \vdash K_1 = K_2}$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash T = T} \quad (14)$$

$$\frac{\Gamma \vdash c_1 = c_2 : T}{\Gamma \vdash S(c_1) = S(c_2)} \quad (15)$$

$$\frac{\Gamma \vdash K_2' = K_1' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' = K_2''}{\Gamma \vdash \Pi\alpha{:}K_1'.K_1'' = \Pi\alpha{:}K_2'.K_2''} \quad (16)$$

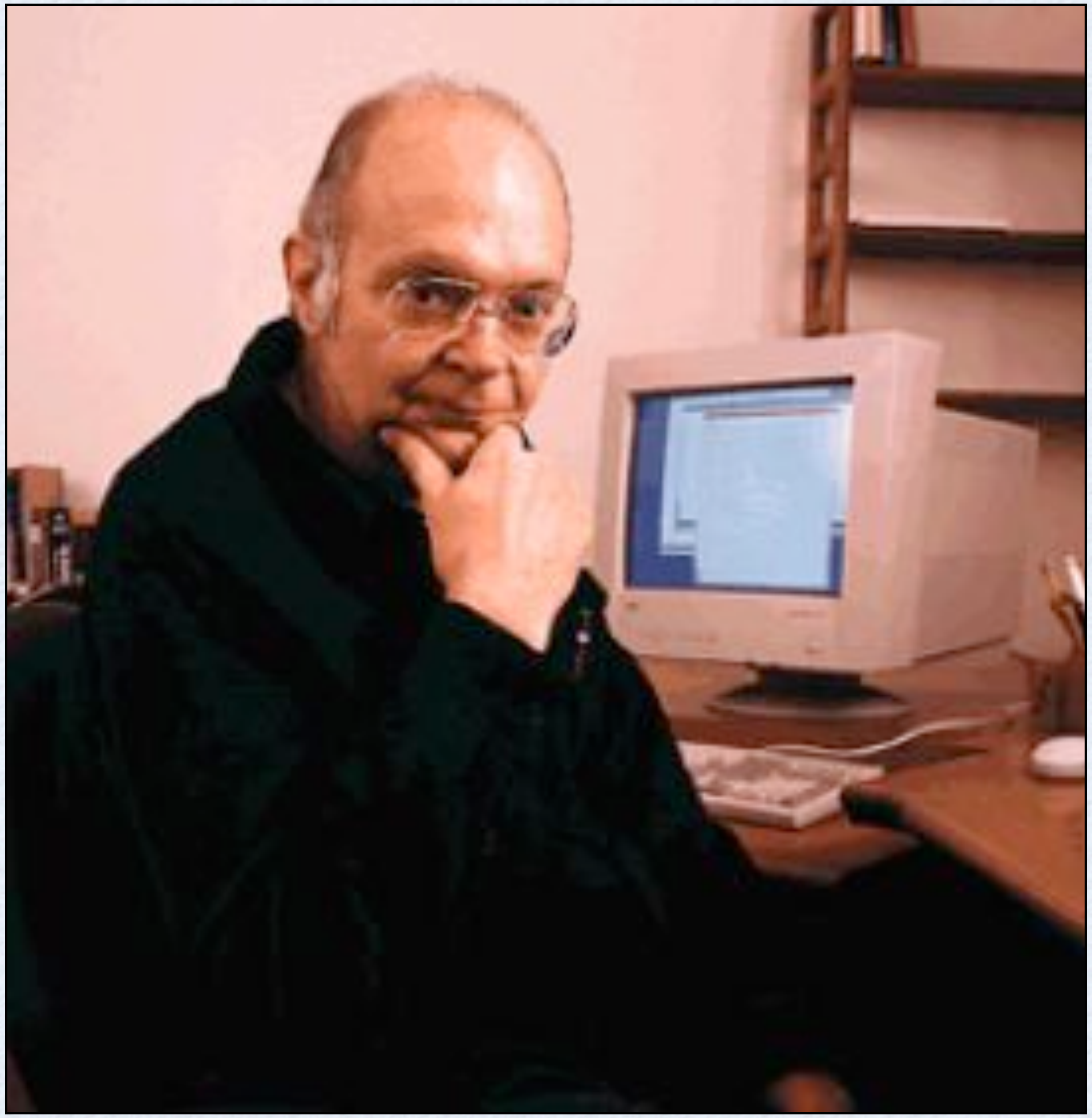
$$\frac{\Gamma \vdash K_1' = K_2' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' = K_2''}{\Gamma \vdash \Sigma\alpha{:}K_1'.K_1'' = \Sigma\alpha{:}K_2'.K_2''} \quad (17)$$

```
sf/keq ⊏ keq :: sf/kd → sf/kd → sort.
sf/eq ⊏ eq :: sf/tp → sf/tp → sf/kd
            → sort.
sf/kof : sf/tp → sf/kd → type.

r14 :: sf/keq t t.

% no r15

r16 :: sf/keq (pi K1’ [a] K1” a)
              (pi K2’ [a] K2” a)
    ← sf/keq K2’ K1’
    ← ({a} sf/kof a K1’
          → sf/keq (K1” a) (K2” a)).
```

Contributions

- Refinements are **useful**:
  - many case studies
  - subset interpretation
- Refinements are **practical**:
  - simple yet rich metatheory
  - sort reconstruction

Summary



- LFR: an expressive and practical logical framework
- (I think Knuth would be intrigued!)