Refinement Types for Logical Frameworks

William Lovas
Thesis

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

“judgments as types”
Refinement types are a useful and practical extension to the LF logical framework.

subtyping ($\leq$), intersections ($\wedge$)
Shoulders of Giants
Shoulders of Giants

deBruijn
Shoulders of Giants

deBruijn

AUTOMATH
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

\[ \alpha := \text{Fun} \text{ sort}(g) \]

I mean \( \alpha \) is a subsort of \( g \). Then if \( y \) is of sort \( \alpha \), and if \( f \) is a "function" \([x \in g]_\alpha \), I am allowed to write \( \{y\}f \) and the latter expression is of sort \( \overline{\epsilon}_1 \).

Furthermore constructions such as

\[
\begin{align*}
  x & \quad \text{sort} \quad g \\
  \theta & := \Gamma(x) \quad \overline{\epsilon}_1 \\
  x & \quad \text{sort} \quad \alpha \\
  \theta & := \Gamma(x) \quad \overline{\epsilon}_2
\end{align*}
\]

may be used; in these circumstances \( \theta(y) \) is defined to be \( \Gamma(y) \), of sort \( \overline{\epsilon}_2 \). In other words I allow the symbol \( \theta \) to be defined twice, both for sort \( g \) and its subsort \( \alpha \); 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.
Dear Dick:

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

When I write

\[ \alpha := \Phi \quad \text{sort} \quad (\xi) \]

I mean \( \alpha \) is a subsort of \( \xi \). Then if \( y \) is of sort \( \alpha \), and if \( f \) is a "function" \( [x \in \xi_1] \), I am allowed to write \( (y)f \) and the latter expression is of sort \( \xi_1 \).

Furthermore constructions such as

\[
\begin{align*}
  x & : \xi \\
  \theta & := \tau(x) & \xi & \xi_1 \\
  x & : \alpha \\
  \theta & := \tau(x) & \alpha & \xi_2
\end{align*}
\]

may be used; in these circumstances \( \theta(y) \) is defined to be \( \tau(y) \), of sort \( \xi_2 \). In other words I allow the symbol \( \theta \) to be defined twice, both for sort \( \xi \) and its subsort \( \alpha \); 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.
Dear Dick:

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

When I write

\[ \alpha := \text{FM sort } (\xi) \]

I mean \( \alpha \) is a subsort of \( \xi \). Then if \( y \) is of sort \( \alpha \), and if \( f \) is a "function"

\[ [x \xi]_\xi, \]

I am allowed to write \( [y] f \) and the latter expression is of sort \( \xi_1 \).

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.
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**

- 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

- 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

<table>
<thead>
<tr>
<th>On paper</th>
<th>In LF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Simple type</td>
</tr>
<tr>
<td>▶ $e ::= ...$ $\tau ::= ...$</td>
<td>$\text{exp : type. tp : type.}$</td>
</tr>
<tr>
<td>Judgement</td>
<td>Type family</td>
</tr>
<tr>
<td>▶ $\Gamma \vdash e : \tau$</td>
<td>$\text{of : exp } \rightarrow \text{tp } \rightarrow \text{type.}$</td>
</tr>
<tr>
<td>Derivation</td>
<td>Well-typed term</td>
</tr>
<tr>
<td>▶ $\mathcal{D} :: \Gamma \vdash e : \tau$</td>
<td>$M : \text{of ET}$</td>
</tr>
<tr>
<td>Proof checking</td>
<td>Type checking</td>
</tr>
</tbody>
</table>
Refinement types / Sorts

S \leq T

"subsort"
Refinement types / Sorts

S ≤ T

"subsort"
Refinement types / Sorts

\[ S \leq T \]
Refinement types / Sorts

S ⊑ A

"refines"
Refinement types / Sorts

“subsort”

\[ S \leq T \]

\[ S \subset A \]

“refines”
Refinement types / Sorts

"subsort"  
\[ S \leq T \]

"intersect"  
\[ T \wedge U \]

"refines"  
\[ S \preceq A \]
Refinement types / Sorts

"subsort"
\[ S \leq T \]

"intersect"
\[ T \land U \]

"top"
\[ T \]

"refines"
\[ S \sqsubseteq A \]
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

\text{nat} : \text{type}.

\text{z} : \text{nat}.

\text{s} : \text{nat} \rightarrow \text{nat}.
Example: natural numbers

\[
\begin{align*}
\text{nat} & : \text{type}. \\
\text{z} & : \text{nat}. \\
\text{s} & : \text{nat} \rightarrow \text{nat}.
\end{align*}
\]

\[
\begin{align*}
\text{double} & : \text{nat} \rightarrow \text{nat} \rightarrow \text{type}. \\
\text{dbl-z} & : \text{double z z}.
\end{align*}
\]

\[
\begin{align*}
\text{dbl-s} & : \text{double (s N) (s (s (N2)))} \\
& \leftarrow \text{double N N2}.
\end{align*}
\]
Example: natural numbers

\textbf{nat} : type.
\textbf{z} : nat.
\textbf{s} : nat \rightarrow nat.
\textbf{double} : nat \rightarrow \textbf{nat} \rightarrow \textbf{type}.
\textbf{dbl-z} : double \textbf{z} \textbf{z} \textbf{z}.
\textbf{dbl-s} : double (s \textbf{N}) (s (s (N2))).

always even!
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

\[
\begin{align*}
\text{even} & : \text{nat} \rightarrow \text{type}. \\
\text{odd} & : \text{nat} \rightarrow \text{type}. \\
\text{ev-z} & : \text{even}\ z. \\
\text{ev-s} & : \text{even}\ (s\ N) \leftarrow \text{odd}\ N. \\
\text{od-s} & : \text{odd}\ (s\ N) \leftarrow \text{even}\ N.
\end{align*}
\]
Option 1: explicit proofs

\[
\begin{align*}
even : & \text{nat} \to \text{type.} \\
odd : & \text{nat} \to \text{type.} \\
even-z : & \text{even z.} \\
ev-s : & \text{even} (s \, N) \leftarrow \text{odd} \, N. \\
od-s : & \text{odd} (s \, N) \leftarrow \text{even} \, N. \\
double : & \text{nat} \to \Pi_{N2:\text{nat}}. \ \text{even} \, N2 \to \text{type.} \\
\text{dbl-z} : & \text{double} \, z \, z \, \text{ev-z.} \\
\text{dbl-s} : & \text{double} \, N \, (s \,(s \, N2)) \,(\text{ev-s} \,(\text{od-s} \, \text{Deven})) \leftarrow \text{double} \, N \, N2 \, \text{Deven.}
\end{align*}
\]
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
even : type.
odd : type.

ze : even.
se : odd \rightarrow even.
s_o : even \rightarrow odd.
Option 2: implicit proofs

\[\text{even} : \text{type}.
\text{odd} : \text{type}.
\text{ze} : \text{even}.
\text{se} : \text{odd} \rightarrow \text{even}.
\text{so} : \text{even} \rightarrow \text{odd}.
\]

\[\text{double} : \text{nat} \rightarrow \text{even} \rightarrow \text{type}.
\text{dbl-z} : \text{double} \ z \ z \text{ze}.
\text{dbl-s} : \text{double} \ N \ (\text{se} \ (\text{so} \ N2)) \\
\leftarrow \text{double} \ N \ N2.\]
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

```plaintext
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 ...


even : nat \rightarrow type.
odd : nat \rightarrow type.

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

double-even : double N N2 \rightarrow even N2 \rightarrow 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

\[
\begin{align*}
\text{even} & \sqsubseteq \text{nat}.
\text{odd} & \sqsubseteq \text{nat}.
\text{z} & :: \text{even}.
\text{s} & :: \text{even} \rightarrow \text{odd} \land \text{odd} \rightarrow \text{even}.
\end{align*}
\]
Better option: refinements

\begin{align*}
\text{even} & \sqsubseteq \text{nat}. \\
\text{odd} & \sqsubseteq \text{nat}. \\
z & :: \text{even}. \\
s & :: \text{even} \to \text{odd} \land \text{odd} \to \text{even}. \\
\text{double} & :: \text{nat} \to \text{even} \to \text{type}. \\
\text{dbl-z} & :: \text{double} \ z \ z. \\
\text{dbl-s} & :: \text{double} \ (s \ N) \ (s \ (s \ (N2))) \\
& \leftarrow \text{double} \ N \ N2.
\end{align*}
Better option: refinements

\begin{align*}
even \sqsubseteq \text{nat}. \\
\text{odd} \sqsubseteq \text{nat}.
\end{align*}

\begin{align*}
z :: & \text{even}. \\
s :: & \text{even} \to \text{odd} \land \text{odd} \to \text{even}.
\end{align*}

\begin{align*}
\text{double} :: & \text{nat} \to \text{even} \to \text{type}.
\end{align*}

\begin{align*}
\text{dbl-z} :: & \text{double } z \ z. \\
\text{dbl-s} :: & \text{double } (s \ N) \ (s \ (s \ (N2))) \\
& \leftarrow \text{double } N \ N2.
\end{align*}
**Better option: refinements**

<table>
<thead>
<tr>
<th></th>
<th>simple</th>
<th>lightweight</th>
<th>direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. implicit proofs</td>
<td>red</td>
<td>green</td>
<td>green</td>
</tr>
<tr>
<td>2. explicit proofs</td>
<td>green</td>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>3. metatheorem</td>
<td>green</td>
<td>green</td>
<td>red</td>
</tr>
<tr>
<td><strong>4. refinements</strong></td>
<td>green</td>
<td>green</td>
<td>green</td>
</tr>
</tbody>
</table>

- **Simple**: doubling judgment doesn’t change.
- **Lightweight**: constructors remain the same.
- **Direct**: strong typing guarantee on derivations.
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 $\beta$-normal and $\eta$-long.
Canonical forms method

- Represent *only* the canonical forms:
  - $\beta$-normal syntactically
  - $\eta$-long through typing
  - hereditary substitutions contract redexes
- Simplifies metatheory, emphasizes adequacy
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:

\[
\Gamma \vdash R \Rightarrow P' \quad P' = P
\]

\[
\Gamma \vdash R \Leftarrow P
\]
Key rule:

- base type, so atoms fully applied

\[
\Gamma \vdash R \Rightarrow P' \quad P' = P
\]

\[
\Gamma \vdash R \Leftarrow P
\]
Key rule:

- base type, so atoms fully applied
- the only appeal to type equality
Checking with subsorting

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

\[
\Gamma \vdash N \leftrightarrow S
\]

\[
\Gamma \vdash R \Rightarrow Q' \quad Q' \leq Q
\]

\[
\Gamma \vdash R \leftarrow Q
\]
Checking with subsorting

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

\[
\Gamma \vdash R \Rightarrow Q' \quad Q' \leq Q
\]

\[
\Gamma \vdash R \Leftarrow Q
\]
Intersections

- Similar to product types, but no proof term

\[
\begin{align*}
\Gamma \vdash N &\iff S_1 \\
\Gamma \vdash N &\iff S_2 \\
\hline
\Gamma \vdash N &\iff S_1 \land S_2 \\
\hline
\Gamma \vdash R &\Rightarrow S_1 \\
\Gamma \vdash R &\Rightarrow S_2
\end{align*}
\]
Important principles

- **Substitution:**
  
  if $\Gamma, x::S \sqsubseteq A \vdash N \Leftarrow T$ and $\Gamma \vdash M \Leftarrow S$,
  
  then $\Gamma \vdash [M/x]_A N \Leftarrow T$.

- **Identity:** for all $A$: $\Gamma, x::S \sqsubseteq A \vdash \eta_A(x) \Leftarrow S$. 
Subsorting

- Key rule:

\[
\Gamma \vdash R \Rightarrow Q' \quad Q' \leq Q
\]

\[
\Gamma \vdash R \Leftarrow Q
\]

- Bidirectional: subsorting only at mode switch
- Canonical: mode switch only at base sort
Subsorting at higher sorts?

- Structural rules? e.g.

\[
\frac{S_2 \leq S_1 \quad T_1 \leq T_2}{S_1 \rightarrow T_1 \leq S_2 \rightarrow T_2}
\]

- Distributivity?

\[
(S \rightarrow T_1) \land (S \rightarrow T_2) \leq S \rightarrow (T_1 \land T_2)
\]
Subsorting at higher sorts!

- Intrinsic subsorting:
  \[ S \leq T \text{ and } \Gamma \vdash N \iff S, \text{ then } \Gamma \vdash N \iff T. \]
Subsorting at higher sorts!

- **Intrinsic subsorting:**
  \[ S \leq T \text{ and } \Gamma \vdash N \equiv S, \text{ then } \Gamma \vdash N \equiv T. \]

- **Equivalently:**
  \[ S \leq T, \text{ then } \Gamma, x :: S \sqsubseteq A \vdash \eta_A(x) \equiv T. \]
Subsorting at higher sorts!

- **Intrinsic subsorting:**
  \[
  \text{if } S \leq T \text{ and } \Gamma \vdash N \iff S, \text{ then } \Gamma \vdash N \iff T.
  \]

- **Equivalently:**
  \[
  \text{if } S \leq T, \text{ then } \Gamma, x::S \sqsubseteq A \vdash \eta_A(x) \iff T.
  \]
  ▶ just like the Identity principle!
Subsorting at higher sorts!

- **Intrinsic subsorting:**
  \[
  \text{if } S \leq T \text{ and } \Gamma \vdash N \Leftarrow S, \text{ then } \Gamma \vdash N \Leftarrow T.
  \]

- **Equivalently:**
  \[
  \text{if } S \leq T, \text{ then } \Gamma, x::S \sqsubseteq A \vdash \eta_A(x) \Leftarrow T.
  \]

  - just like the Identity principle!
  - … also the Substitution principle …
Subsorting at higher sorts!

- **Intrinsic subsorting:**
  
  \[
  \text{if } S \leq T \text{ and } \Gamma \vdash N \iff S, \text{ then } \Gamma \vdash N \iff T.
  \]

- **Equivalently:**
  
  \[
  \text{if } S \leq T, \text{ then } \Gamma, x::S \sqsubseteq A \vdash \eta_A(x) \iff T.
  \]

  - 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 $\Gamma, x::S \sqsubseteq A \vdash \eta_A(x) \iff T$, then $S \leq T$.

- **Or:** if $\Gamma \vdash N \iff S$ implies $\Gamma \vdash N \iff T$, then $S \leq T$. 
Subsorting at higher sorts?

- ... and also complete!
- **Theorem**: if $\Gamma, x :: S \sqsubseteq A \vdash \eta_A(x) \leftarrow T$, 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.
✓ 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

\textbf{nat} : \textbf{type}.
\text{z} : \text{nat}.
\text{s} : \text{nat} \to \text{nat}.

\textbf{even} \sqsubseteq \text{nat}.
\textbf{odd} \sqsubseteq \text{nat}.
\text{z} :: \text{even}.
\text{s} :: \text{even} \to \text{odd}
\wedge \text{odd} \to \text{even}.
Sorts as predicates

\( \text{nat} : \text{type.} \)
\( z : \text{nat.} \)
\( s : \text{nat} \rightarrow \text{nat.} \)

\( \text{even} \sqsubseteq \text{nat.} \)
\( \text{odd} \sqsubseteq \text{nat.} \)
\( z :: \text{even.} \)
\( s :: \text{even} \rightarrow \text{odd} \)
\( \land \text{odd} \rightarrow \text{even.} \)

\( \text{even} : \text{nat} \rightarrow \text{type.} \)
\( \text{odd} : \text{nat} \rightarrow \text{type.} \)
\( \text{pf-z} : \text{even z.} \)
\( \text{pf-s}_1 : \Pi x: \text{nat. even } x \rightarrow \text{odd } (s \ x) \)
\( \text{pf-s}_2 : \Pi x: \text{nat. odd } x \rightarrow \text{even } (s \ x). \)
<table>
<thead>
<tr>
<th>Sorts as predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nat : type</strong></td>
</tr>
<tr>
<td><strong>z : nat.</strong></td>
</tr>
<tr>
<td><strong>s : nat →</strong></td>
</tr>
<tr>
<td><strong>even ⊑ nat.</strong></td>
</tr>
<tr>
<td><strong>odd ⊑ nat.</strong></td>
</tr>
<tr>
<td><strong>z :: even.</strong></td>
</tr>
<tr>
<td><strong>s :: even → odd</strong></td>
</tr>
<tr>
<td><strong>∧ odd → even.</strong></td>
</tr>
</tbody>
</table>

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


sorts as predicates

- Translation follows the following rules:
  - refinements become predicates
  - sort declarations become proof constructors

(One twist: proof irrelevance)

\[
\begin{align*}
\text{even} & : \text{nat} \rightarrow \text{type}. \\
\text{odd} & : \text{nat} \rightarrow \text{type}. \\
\text{pf-}z & : \text{even } z. \\
\text{pf-s}_1 & : \Pi x : \text{nat}. \text{even } x \rightarrow \text{odd } (s \ x) \\
\text{pf-s}_2 & : \Pi x : \text{nat}. \text{odd } x \rightarrow \text{even } (s \ x).
\end{align*}
\]
Sorts as predicates

<table>
<thead>
<tr>
<th>nat : <strong>type</strong>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>z : nat.</td>
</tr>
<tr>
<td>s : nat (\rightarrow) nat.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>even (\subseteq) nat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>odd (\subseteq) nat.</td>
</tr>
<tr>
<td>z :: even.</td>
</tr>
<tr>
<td>s :: even (\rightarrow) odd</td>
</tr>
<tr>
<td>(\land) odd (\rightarrow) even.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>even : nat (\rightarrow) <strong>type</strong>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>odd : nat (\rightarrow) <strong>type</strong>.</td>
</tr>
<tr>
<td>pf-z : even z.</td>
</tr>
<tr>
<td>pf-s_1 : (\Pi x:\text{nat.} \text{even } x \rightarrow \text{odd } (s \ x))</td>
</tr>
<tr>
<td>pf-s_2 : (\Pi x:\text{nat.} \text{odd } x \rightarrow \text{even } (s \ x)).</td>
</tr>
</tbody>
</table>
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-s₁ : Πx:nat. even x → odd (s x)
pf-s₂ : Πx:nat. odd x → even (s x).

N :: even iff M : even N
(for some M)
LF enjoys a well-developed theory of adequate representations

- **Adequacy**: compositional, property-preserving bijection between informal entities and canonical terms

![Informal Math to LFR Encoding](image-url)
Adequacy

informal math  adequacy  LFR encoding
Adequacy

informal math \rightarrow \text{adequacy} \rightarrow \text{LFR encoding}

LF + proof irrelevance

sound and complete translation
Adequacy

informal math

LF + proof irrelevance

LFR encoding

LF + proof irrelevance

sound and complete translation

adequacy
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.
double : nat → nat → type.
dbl/s : double N N2 → double (s N) (s (s N2)).
double : nat → nat → type.
dbl/s : double N N2 → double (s N) (s (s N2)).
double : nat → nat → type.
    double N N2 → double (s N) (s (s N2)).
sort reconstruction

double : \texttt{nat} \rightarrow \texttt{nat} \rightarrow \texttt{type}.
dbl/s : \Pi N: \texttt{nat}. \Pi N2: \texttt{nat}.
  \texttt{double } N \ N2 \rightarrow \texttt{double } (s \ N) \ (s \ (s \ N2)).
double : \( \text{nat} \rightarrow \text{nat} \rightarrow \text{type} \).

\( \text{dbl/s} : \Pi N:\text{nat}. \Pi N2:\text{nat}. \)
\[
\quad \text{double } N \ N2 \rightarrow \text{double } (s \ N) \ (s \ (s \ N2)).
\]

double* \sqsubseteq \text{double} :: \top \rightarrow \text{even} \rightarrow \text{type}.

\( \text{dbl/s} :: \text{double* } N \ N2 \rightarrow \text{double* } (s \ N) \ (s \ (s \ N2)).\)
Sort Reconstruction

double : nat → nat → type.
     double N N2 → double (s N) (s (s N2)).

\[
double^* \sqsubseteq \text{double} :: \top \rightarrow \text{even} \rightarrow \text{type}.
\]
dbl/s :: double* N N2 → double* (s N) (s (s N2)).
double* ⊑ double :: ∀ T → even → type.

dbl/s :: Π N :: σ ⊑nat. Π N2 :: σ2 ⊑nat.

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

double* ▼ double :: \( \top \rightarrow \text{even} \rightarrow \text{type} \).
dbl/s :: \( \Pi N::\sigma \sqsubseteq \text{nat}. \Pi N2::\sigma_2 \sqsubseteq \text{nat} \).

double* \( N \ N2 \rightarrow \text{double}^* \ (s \ N) \ (s \ (s \ N2)) \).
Sort Reconstruction

double* ⊑ double :: T → even → type.
dbl/s :: ΠN::σ ⊑ nat. ΠN2::σ2 ⊑ nat.
\quad double* N N2 → double* (s N) (s (s N2)).

Constraint generation

σ2 ≤ even
Sort Reconstruction

double* ⊑ double :: ⊤ → even → type.
dbl/s :: ΠN::σ ⊑ nat. ΠN2::σ2 ⊑ nat.
   double* N N2 → double* (s N) (s (s N2)).

Constraint generation

σ2 ≤ even

Constraint solving

double* ⊑ double :: ⊤ → even → type.
dbl/s :: ΠN::⊤ ⊑ nat. ΠN2::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 Study 1: normal forms

**Definition 3.1 (Weak-Head Normal)**

\( T_*, A_1 \rightarrow A_2, \forall X \leq A:K.B \), 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)

\[ T_*, A_1 \rightarrow A_2, \forall X \leq A:K.B, \text{ and } \Lambda X \leq A:K.B \text{ are weak head normal.} \]

\[ X(A_1, \ldots, A_n) \text{ is weak head normal if } A_1, \ldots, A_n \text{ are in normal form.} \]

Case Study 1: normal forms

**Definition 3.1 (Weak-Head Normal)**

\[ T_\star, A_1 \rightarrow A_2, \forall X \leq A:K.B, \text{ and } \Lambda X \leq A:K.B \text{ are weak head normal.} \]

\[ X(A_1, \ldots, A_n) \text{ is weak head normal if } A_1, \ldots, A_n \text{ are in normal form.} \]


- 3 steps:
  - translate grammar of types
  - characterize normal types
  - characterize weak head normal types
Case Study 1: normal forms

- \( A ::= X \mid A \rightarrow A \mid \forall X \leq A : K. A \mid \Lambda X \leq A : K. A \mid A A \mid T^* \)

kd : type.
tp : type.

T* : tp.
arrow : tp \rightarrow tp \rightarrow tp.
all : tp \rightarrow kd \rightarrow (tp \rightarrow tp) \rightarrow tp.
Lam : tp \rightarrow kd \rightarrow (tp \rightarrow tp) \rightarrow tp.
App : tp \rightarrow tp \rightarrow tp.
Case Study 1: normal forms

- $A ::= P \mid A \to A \mid \forall X \leq A : K. A \mid \Lambda X \leq A : K. A \mid T^*$

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

- $btp \sqsubseteq tp. \ ntp \sqsubseteq tp.$

- $T^* :: ntp.$
- $arrow :: ntp \to ntp \to ntp.$
- $all :: ntp \to \top \to (btp \to ntp) \to ntp.$
- $Lam :: ntp \to \top \to (btp \to ntp) \to ntp.$
- $App :: btp \to ntp \to btp.$
- $btp \leq ntp.$
Case Study 1: normal forms

**Definition 3.1 (Weak-Head Normal)**

\[ T_*, A_1 \rightarrow A_2, \forall X \leq A:K.B, \text{ and } \Lambda X \leq A:K.B \] are weak head normal.

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

\[
\text{whntp} \sqsubseteq \text{tp}.
\]

\[
T^* :: \text{whntp}.
\]

\[
\text{arrow} :: \top \rightarrow \top \rightarrow \text{whntp}.
\]

\[
\text{all} :: \top \rightarrow \top \rightarrow (\text{btp} \rightarrow \top) \rightarrow \text{whntp}.
\]

\[
\text{Lam} :: \top \rightarrow \top \rightarrow (\text{btp} \rightarrow \top) \rightarrow \text{whntp}.
\]

\[
\text{btp} \leq \text{whntp}.
\]
3.2.1 Call-By-Value (CBV) Strategy

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

\[
V ::= x | \lambda x: A. M | \Lambda u: K. M
\]
\[
R ::= (\lambda x: A. M) V | (\Lambda u: K. M)\{A\} | \text{abort}_A(M) | \text{callcc}_A(M)
\]
\[
E ::= [] | E M | V E | E\{A\}
\]

3.2.2 Call-By-Name (CBN) Strategy

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

\[
V ::= \lambda x: A. M | \Lambda u: K. M
\]
\[
R ::= (\lambda x: A. M_1) M_2 | (\Lambda u: K. M)\{A\} | \text{abort}_A(M) | \text{callcc}_A(M)
\]
\[
E ::= [] | E M | E\{A\}
\]
3.2.1 Call-By-Value (CBV) Strategy

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

\[ V ::= x \mid \lambda x: A. M \mid \Lambda u: K. M \]

\[ R ::= (\lambda x: A. M) V \mid (\Lambda u: K. M)\{A\} \mid \text{abort}_A(M) \mid \text{callcc}_A(M) \]

\[ E ::= [] \mid E M \mid V E \mid E\{A\} \]

3.2.2 Call-By-Name (CBN) Strategy

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

\[ V ::= \lambda x: A. M \mid \Lambda u: K. M \]

\[ R ::= (\lambda x: A. M_1) M_2 \mid (\Lambda u: K. M)\{A\} \mid \text{abort}_A(M) \mid \text{callcc}_A(M) \]

\[ E ::= [] \mid E M \mid E\{A\} \]

Case Study 2: CBV/CBN

Definition 2.1 (Syntax)

**Kinds**

\[ K ::= \Omega | K_1 \Rightarrow K_2 \]

** Constructors**

\[ A ::= \alpha | u | A_1 \rightarrow A_2 | \forall u : K . A | \lambda u : K . A | A_1 A_2 \]

**Terms**

\[ M ::= x | \lambda x : A . M | M_1 M_2 | \Lambda u : K . M | M \{ A \} | \text{callcc}_A(M) | \text{abort}_A(M) \]

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

<table>
<thead>
<tr>
<th>Evctx : type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>lam : tp ↦ (tm ↦ tm) ↦ tm.</td>
</tr>
<tr>
<td>app : tm ↦ tm ↦ tm.</td>
</tr>
<tr>
<td>Lam : kd ↦ (tp ↦ tm) ↦ tm.</td>
</tr>
<tr>
<td>App : tm ↦ tp ↦ tm.</td>
</tr>
<tr>
<td>callcc : tp ↦ tm ↦ tm.</td>
</tr>
<tr>
<td>abort : tp ↦ tm ↦ tm.</td>
</tr>
</tbody>
</table>

< : 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

\[
\text{lambda} \sqsubseteq \text{tm.} \quad \text{Lambda} \sqsubseteq \text{tm.} \quad \text{control} \sqsubseteq \text{tm.}
\]

\[
\text{lam} :: \top \rightarrow (\top \rightarrow \top) \rightarrow \text{lambda}. \quad \text{abort} :: \top \rightarrow \top \rightarrow \text{control}. \\
\text{Lam} :: \top \rightarrow (\top \rightarrow \top) \rightarrow \text{Lambda}. \quad \text{callcc} :: \top \rightarrow \top \rightarrow \text{control}.
\]
3.2.2 Call-By-Name (CBN) Strategy

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

\[
\begin{align*}
V & ::= \lambda x:A.M \mid \Lambda u:K.M \\
R & ::= (\lambda x:A.M_1)M_2 \mid (\Lambda u:K.M)\{A\} \mid \\
& \quad \text{abort}_A(M) \mid \text{callcc}_A(M) \\
E & ::= \mathbf{[]} \mid EM \mid E\{A\}
\end{align*}
\]

\[\text{n/red} \subseteq \text{tm}.
\]

\[\text{lambda} \leq \text{n/red}.
\]

\[\text{Lambda} \leq \text{n/red}.
\]

\[\text{app} :: \text{lambda} \to \top \to \text{n/red}.
\]

\[\text{App} :: \text{Lambda} \to \top \to \text{n/red}.
\]

\[\text{control} \leq \text{n/red}.
\]

\[\text{n/evctx} \subseteq \text{evctx}.
\]

\[\text{<>} :: \text{n/evctx}.
\]

\[\text{capp1} :: \text{n/evctx} \to \top \to \text{n/evctx}.
\]

\[\text{cApp} :: \text{n/evctx} \to \top \to \text{n/evctx}.
\]
3.2.1 Call-By-Value (CBV) Strategy

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

\[ V ::= x \mid \lambda x:A.M \mid \Lambda u:K.M \]
\[ R ::= (\lambda x:A.M) V \mid (\Lambda u:K.M)\{A\} \mid \text{abort}_A(M) \mid \text{callcc}_A(M) \]
\[ E ::= [] \mid EM \mid V E \mid E\{A\} \]

\( v/val, v/red, v/lambda \sqsubseteq \text{tm}. \)

\( v/evctx \sqsubseteq \text{evctx}. \)

\( \text{lam} :: \top \rightarrow (v/val \rightarrow \top) \rightarrow v/lambda. \)

\( v/lambda \sqsubseteq v/val. \)

\( \text{Lambda} \sqsubseteq v/val. \)

\( \text{app} :: v/lambda \rightarrow v/val \rightarrow v/red. \)

\( \text{App} :: \text{Lambda} \rightarrow \top \rightarrow v/red. \)

\( \text{control} \sqsubseteq v/red. \)

\( \text{cApp} :: v/evctx \rightarrow \top \rightarrow v/evctx. \)

\( \text{capp1} :: v/evctx \rightarrow \top \rightarrow v/evctx. \)

\( \text{capp2} :: v/val \rightarrow v/evctx \rightarrow v/evctx. \)
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 \( \vdash_{sf} \).
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 \[ \Gamma ::= \epsilon \mid \Gamma, \alpha : K \]

Fig. 1. Syntax

kd : type. tp : type.

t : kd.
sing : tp \rightarrow kd.
pi : kd \rightarrow (tp \rightarrow kd) \rightarrow kd.
sigma : kd \rightarrow (tp \rightarrow kd) \rightarrow kd.
Case Study 3: singletons

The application of constructors would usually be written in the form independent of a particular place of user so when or assignment is written, the captures avoiding substitution of terms so they are omitted. For our purposes, which deal with constructor equality, we need not be concerned with terms. For our purposes, which deal with constructor equality, we need not be concerned with constructor variables, which makes it useful to have free constructor variables. This makes it useful to have free constructor variables, which makes it useful to have

The syntax of the singleton calculus is given in Figure 1. It consists of a class of singleton kinds, but we will use their results almost entirely.

In addition, the syntax provides a class of kinds, which classify constructors. The class of constructors contains variables of type constructors, usually referred to as "constructors" for brevity, and a class of kinds. The kind structure is the novelty of the singleton calculus. The base kinds include Sigma, Pi, and a collection of primitive type operators such as int. The correctness proof draws from the work of Karl Crary.

In Section 2, I present the singleton elimination strategy and state its correctness including remarks appear in Section 5. Section 3 is dedicated to the proof of the correctness theorem and consists of a class of singleton kinds, but we will use their results almost entirely.

Fig. 1. Syntax

\[
\begin{align*}
\text{kinds} & \quad K ::= T \mid S(c) \mid \Pi \alpha:K_1.K_2 \mid \Sigma \alpha:K_1.K_2 \\
\text{constructors} & \quad c ::= \alpha \mid b \mid \lambda \alpha:K.c \mid c_1 \cdot c_2 \mid \langle c_1, c_2 \rangle \mid \pi_1 c \mid \pi_2 c \\
\text{assignments} & \quad \Gamma ::= \epsilon \mid \Gamma, \alpha:K
\end{align*}
\]

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

\[ \Gamma \vdash K_1 = K_2 \]

\[ \begin{align*}
\Gamma \vdash \text{ok} \\
\Gamma \vdash T = T
\end{align*} \quad (14) \]

\[ \Gamma \vdash c_1 = c_2 : T \\
\Gamma \vdash S(c_1) = S(c_2)
\]

\[ (15) \]

\[ \begin{align*}
\Gamma \vdash K'_2 = K'_1 \\
\Gamma, \alpha : K'_1 \vdash K''_1 = K''_2
\end{align*} \]

\[ \Gamma \vdash \Pi \alpha : K'_1.K''_1 = \Pi \alpha : K'_2.K''_2 \quad (16) \]

\[ \begin{align*}
\Gamma \vdash K'_1 = K'_2 \\
\Gamma, \alpha : K'_1 \vdash K''_1 = K''_2
\end{align*} \]

\[ \Gamma \vdash \Sigma \alpha : K'_1.K''_1 = \Sigma \alpha : K'_2.K''_2 \quad (17) \]

\[ \text{keq : kd} \rightarrow \text{kd} \rightarrow \text{type.} \]
\[ \text{eq : tp} \rightarrow \text{tp} \rightarrow \text{kd} \rightarrow \text{type.} \]
\[ \text{kof : tp} \rightarrow \text{kd} \rightarrow \text{type.} \]

r14 : keq t t.

r15 : keq (sing C1) (sing C2) \leftarrow eq C1 C2 t.

r16 : keq (pi K1' [a] K1'' a) (pi K2' [a] K2'' a) \leftarrow keq K2' K1' \leftarrow (\{a\} kof a K1' \rightarrow keq (K1'' a) (K2'' a)).
Case Study 3: singletons

Kind Equivalence

\[ \Gamma \vdash K_1 = K_2 \]

\[
\begin{align*}
\Gamma \vdash \text{ok} & \quad \Gamma \vdash T = T \\
\Gamma \vdash c_1 = c_2 : T & \quad \Gamma \vdash S(c_1) = S(c_2)
\end{align*}
\]

(14)

\[
\begin{align*}
\Gamma \vdash K'_2 = K'_1 & \quad \Gamma, \alpha:K'_1 \vdash K''_1 = K''_2 \\
\Gamma \vdash \Pi \alpha:K'_1.K''_1 = \Pi \alpha:K'_2.K''_2
\end{align*}
\]

(15)

\[
\begin{align*}
\Gamma \vdash K'_1 = K'_2 & \quad \Gamma, \alpha:K'_1 \vdash K''_1 = K''_2 \\
\Gamma \vdash \Sigma \alpha:K'_1.K''_1 = \Sigma \alpha:K'_2.K''_2
\end{align*}
\]

(16)

(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
LFR: an expressive and practical logical framework

(I think Knuth would be intrigued!)