Mechanizing Metatheory with LF and Twelf

Taus Brock-Nannestad
Chris Martens
Carsten Schürmann

CADE Twelf Tutorial

June 10, 2013
(Modified from 2009 POPL tutorial by the CMU POP Group)

What We'll Learn

Representation of languages and logics in LF

What We'll Learn

Representation of languages and logics in LF

- Syntax

What We'll Learn

Representation of languages and logics in LF

- Syntax $\quad e ::= x \mid \lambda x.e \mid e_1\ e_2$

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

$$\text{Typing} : \Gamma \vdash e : \tau$$

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

$$\text{Operational semantics}: \; e \mapsto e'$$

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

Translations between languages; compiler passes

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

Proof theory for a logic

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

Mechanization of metatheory using Twelf

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

Mechanization of metatheory using Twelf

Type safety

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

Mechanization of metatheory using Twelf

Progress: If $e : \tau$, then e *value* or else $e \mapsto e'$ (for some e').

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

Mechanization of metatheory using Twelf

Decidability of type checking

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

Mechanization of metatheory using Twelf

Cut elimination for a logic

What We'll Learn

Representation of languages and logics in LF

- Syntax
- Judgements

Mechanization of metatheory using Twelf

Correctness of compiler transformations

Part I

Basic Twelf Skills

Basic Twelf Skills

1. Representing syntax and judgements
2. Totality of judgements
3. Proving metatheorems

Basic Twelf Skills

1. Representing syntax and judgements
2. Totality of judgements
3. Proving metatheorems

What is LF?

LF is a logical framework, a general theory of abstract syntax.

What is LF?

LF is a logical framework, a general theory of abstract syntax.

- Now: basic abstract syntax and judgements

What is LF?

LF is a logical framework, a general theory of abstract syntax.

- Now: basic abstract syntax and judgements
- Later: binding and scope of identifiers, context-sensitive rules

What is LF?

LF is a logical framework, a general theory of abstract syntax.

- Now: basic abstract syntax and judgements
- Later: binding and scope of identifiers, context-sensitive rules

A language is inductively presented by a collection of generators, whose types are specified by a signature.

Natural Numbers

The formation judgement n **nat** states that n is a natural number.

Natural Numbers

The formation judgement n **nat** states that n is a natural number.

This judgement is inductively defined by these two rules:

$$\frac{}{\mathit{zero}\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathit{succ}(n)\ \mathbf{nat}}$$

Natural Numbers

The formation judgement n **nat** states that n is a natural number.

This judgement is inductively defined by these two rules:

$$\frac{}{\mathit{zero}\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathit{succ}(n)\ \mathbf{nat}}$$

The judgement n **nat** is the strongest (most restrictive) judgement closed under (obeying) these rules.

Abstract Syntax in LF

$$\frac{}{\mathit{zero}\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathit{succ}(n)\ \mathbf{nat}}$$

Methodology:

- Syntactic category becomes an LF type
- Each rule becomes a generator in the LF signature.

In Twelf:

```
nat  : type.
zero : nat.
succ : nat -> nat.
```

Abstract Syntax in LF

$$\frac{}{\mathit{zero}\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathit{succ}(n)\ \mathbf{nat}}$$

Methodology:

- Syntactic category becomes an LF type
- Each rule becomes a generator in the LF signature.

In Twelf:

```
nat  : type.
zero : nat.
succ : nat -> nat.
```

Abstract Syntax in LF

$$\frac{}{\mathit{zero}\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathit{succ}(n)\ \mathbf{nat}}$$

Methodology:

- Syntactic category becomes an LF type
- Each rule becomes a generator in the LF signature.

In Twelf:

```
nat  : type.
zero : nat.
succ : nat -> nat.
```

LF Representation

```
nat  : type.
zero : nat.
succ : nat -> nat.
```

Construct terms by applying generators:

```
1 : nat = succ zero
2 : nat = succ (succ zero)
```

LF Representation

```
nat  : type.
zero : nat.
succ : nat -> nat.
```

Construct terms by applying generators:

```
1 : nat = succ zero
2 : nat = succ  1
```

Correctness of Representation

$$\frac{}{\mathit{zero}\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathit{succ}(n)\ \mathbf{nat}}$$

represented by

```
nat  : type.
zero : nat.
succ : nat -> nat.
```

Correctness of Representation

$$\frac{}{\mathit{zero}\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathit{succ}(n)\ \mathbf{nat}}$$

represented by

```
nat  : type.
zero : nat.
succ : nat -> nat.
```

Why is this a correct representation?

Correctness of Representation

Representation is adequate iff isomorphic to informal syntax

In this case, we define a bijection between:

- numbers n **nat**
- LF canonical forms (synonym: "terms") $M : \texttt{nat}$

$$\begin{aligned} \ulcorner \mathit{zero} \urcorner &= \texttt{zero} \\ \ulcorner \mathit{succ}(n) \urcorner &= \texttt{succ} \ulcorner n \urcorner \end{aligned}$$

Correctness of Representation

Representation is adequate iff isomorphic to informal syntax

In this case, we define a bijection between:

- numbers n **nat**
- LF canonical forms (synonym: "terms") $M : \texttt{nat}$

$$\begin{aligned} \ulcorner \mathit{zero} \urcorner &= \texttt{zero} \\ \ulcorner \mathit{succ}(n) \urcorner &= \texttt{succ} \ulcorner n \urcorner \end{aligned}$$

Injectivity: easy

Correctness of Representation

Representation is adequate iff isomorphic to informal syntax

In this case, we define a bijection between:

- numbers n **nat**
- LF canonical forms (synonym: "terms") $M : \texttt{nat}$

$$\begin{aligned} \ulcorner \mathit{zero} \urcorner &= \texttt{zero} \\ \ulcorner \mathit{succ}(n) \urcorner &= \texttt{succ} \ulcorner n \urcorner \end{aligned}$$

Injectivity: easy
Surjectivity: take our word for it for now

```
zero : nat.
succ : nat -> nat.
```

Addition Judgement

The addition judgement $m + n$ *is* p states that the sum of m and n is p.

Addition Judgement

The addition judgement $m + n \ \ is \ \ p$ states that the sum of m and n is p.

It is defined by these *rules*:

$$\frac{}{zero + n \ \ is \ \ n} \qquad \frac{m + n \ \ is \ \ p}{succ(m) + n \ \ is \ \ succ(p)}$$

Addition Judgement

The addition judgement $m + n \ \ is \ \ p$ states that the sum of m and n is p.

It is defined by these *rules*:

$$\frac{}{zero + n \ \ is \ \ n} \qquad \frac{m + n \ \ is \ \ p}{succ(m) + n \ \ is \ \ succ(p)}$$

It is the strongest judgement closed under these rules.

Addition Judgement

$$\frac{}{zero + n \;\; is \;\; n} \qquad \frac{m + n \;\; is \;\; p}{succ(m) + n \;\; is \;\; succ(p)}$$

Example: $1 + 1 \;\; is \;\; 2$

$$\frac{\dfrac{}{zero + succ(zero) \;\; is \;\; succ(zero)}}{succ(zero) + succ(zero) \;\; is \;\; succ(succ(zero))}$$

Judgements as Types

The judgement is represented by a family of types in LF:

```
add : nat -> nat -> nat -> type.
```

Judgements as Types

The judgement is represented by a family of types in LF:

```
add : nat -> nat -> nat -> type.
```

Adequacy: The LF type $\texttt{add}\ulcorner m\urcorner\ulcorner n\urcorner\ulcorner p\urcorner$ classifies derivations of $m + n \ \mathit{is} \ p$.

$$\nabla \text{ derives } m + n \ \mathit{is} \ p$$

iff

$$\ulcorner\nabla\urcorner : \texttt{add}\ulcorner m\urcorner\ulcorner n\urcorner\ulcorner p\urcorner$$

Rules as Generators

$$\frac{}{zero + n \;\; is \;\; n} \qquad\qquad \frac{m + n \;\; is \;\; p}{succ(m) + n \;\; is \;\; succ(p)}$$

Each addition rule is represented by a generator:

```
add/z : add zero N N.

add/s : add (succ M) N (succ P)
        <- add M N P.
```

Rules as Generators

$$\frac{}{zero + n \;\; is \;\; n} \qquad\qquad \frac{m + n \;\; is \;\; p}{succ(m) + n \;\; is \;\; succ(p)}$$

Each addition rule is represented by a generator:

```
add/z : add zero N N.

add/s : add (succ M) N (succ P)
        <- add M N P.
```

Use capital letters for schema variables

Representing Syntax and Judgements

Let's type these rules into Twelf!

Exercise

Define multiplication:

$$\frac{}{zero * n \;\; is \;\; zero} \qquad \frac{m * n \;\; is \;\; p \quad n + p \;\; is \;\; p'}{succ(m) * n \;\; is \;\; p'}$$

Syntax for multiple premises:

```
rule : conclusion
       <- premise1
       <- premise2.
```

Basic Twelf Skills

1. Representing syntax and judgements
2. Totality of judgements
3. Proving metatheorems

Totality of Judgements

We can use Twelf to verify totality of judgements:

For all m **nat** and n **nat**,
there exists a p **nat** such that $m + n$ *is* p

Totality of Judgements

We can use Twelf to verify totality of judgements:

For all m **nat** and n **nat**,
there exists a p **nat** such that $m + n$ *is* p

- addition is total, with first two args as inputs; third as output

Totality of Judgements

We can use Twelf to verify totality of judgements:

For all m **nat** and n **nat**,
there exists a p **nat** such that $m + n$ *is* p

- addition is total, with first two args as inputs; third as output
- addition is total with mode + + −

Totality of Judgements

Recast as a statement about LF terms:

For all m **nat** and n **nat**,
there exists a p **nat** such that $m + n$ *is* p

becomes

For all `m:nat` and `n:nat`,
there exists some `p:nat` and ∇`:add m n p`

Totality of Judgements

Twelf checks:

1. Modes: state functional dependencies in a type family (relation).

Totality of Judgements

Twelf checks:

1. Modes: state functional dependencies in a type family (relation).
2. Coverage: all cases have been covered.

Totality of Judgements

Twelf checks:

1. Modes: state functional dependencies in a type family (relation).
2. Coverage: all cases have been covered.
3. Termination: no circular definitions.

Totality of Judgements

Twelf checks:

1. Modes: state functional dependencies in a type family (relation).
2. Coverage: all cases have been covered.
3. Termination: no circular definitions.

Syntax:

- %mode: state the mode
- %total: coverage and termination

Totality of Judgements

```
add : nat -> nat -> nat -> type.
```

Thm: $\forall$ `m:nat` and `n:nat`, $\exists$ `p:nat` and ∇`:add m n p`

Totality of Judgements

```
add : nat -> nat -> nat -> type.
```

Thm: $\forall$ `m:nat` and `n:nat`, $\exists$ `p:nat` and ∇`:add m n p`

Twelf declarations:

```
%mode add +M +N -P.

%total M (add M _ _).
```

Totality of Judgements

```
add : nat -> nat -> nat -> type.
```

Thm: $\forall$ `m:nat` and `n:nat`, $\exists$ `p:nat` and ∇`:add m n p`

Twelf declarations:

```
%mode add +M +N -P.

%total M (add M _ _).
```

The first two args are inputs and the third is an output

Totality of Judgements

```
add : nat -> nat -> nat -> type.
```

Thm: $\forall$ `m:nat` and `n:nat`, $\exists$ `p:nat` and ∇`:add m n p`

Twelf declarations:

```
%mode add +M +N -P.

%total M (add M _ _).
```

Asks Twelf to prove totality by induction on the first argument

Totality of Judgements

```
add : nat -> nat -> nat -> type.
```

Thm: $\forall$ `m:nat` and `n:nat`, $\exists$ `p:nat` and ∇`:add m n p`

Twelf declarations:

```
%mode add +M +N -P.
%worlds () (add _ _ _).
%total M (add M _ _).
```

`add` is to be proved total on closed terms of type `nat`

Totality of Judgements

Let's go over these declarations in more detail in Twelf...

Exercise

Can you use `add` to do subtraction? Is it total...

- with mode `+ - +` ?
- with mode `- + +` ?

Can you use `mult` to do (exact) division? Is it total...

- with mode `+ - +` ?
- with mode `- + +` ?

Basic Twelf Skills

1. Representing syntax and judgements
2. Totality of judgements
3. Proving metatheorems

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety
- Determinacy of evaluation.

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety
- Determinacy of evaluation.
- Decidability of type checking.

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety
- Determinacy of evaluation.
- Decidability of type checking.
- Cut elimination for a logic.

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety
- Determinacy of evaluation.
- Decidability of type checking.
- Cut elimination for a logic.
- Correctness of compiler transformations.

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety
- Determinacy of evaluation.
- Decidability of type checking.
- Cut elimination for a logic.
- Correctness of compiler transformations.

We will consider several examples today:

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety
- Determinacy of evaluation.
- Decidability of type checking.
- Cut elimination for a logic.
- Correctness of compiler transformations.

We will consider several examples today:

- Warm-up: arithmetic

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety
- Determinacy of evaluation.
- Decidability of type checking.
- Cut elimination for a logic.
- Correctness of compiler transformations.

We will consider several examples today:

- Warm-up: arithmetic
- Type safety for a small language (but scales to serious languages such as Standard ML).

Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Type safety
- Determinacy of evaluation.
- Decidability of type checking.
- Cut elimination for a logic.
- Correctness of compiler transformations.

We will consider several examples today:

- Warm-up: arithmetic
- Type safety for a small language (but scales to serious languages such as Standard ML).
- More advanced examples later

Example

$$\frac{}{zero + n \;\; is \;\; n}\; add/z \qquad\qquad \frac{m + n \;\; is \;\; p}{succ(m) + n \;\; is \;\; succ(p)}\; add/s$$

Lemma: For all m **nat**, we can derive $m + zero \;\; is \;\; m$.

Proof: induction on m.

Example

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \ \mathit{is}\ \ m$.

Case for $m = \mathit{zero}$**:**

To show: $\mathit{zero} + \mathit{zero}\ \ \mathit{is}\ \ \mathit{zero}$.

True by rule:

$$\frac{}{\mathit{zero} + \mathit{zero}\ \ \mathit{is}\ \ \mathit{zero}}\ \mathit{add}/z$$

Example

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \ \mathit{is}\ \ m$.

Case for $m = \mathit{succ}(m')$**:**

To show: $\mathit{succ}(m') + \mathit{zero}\ \ \mathit{is}\ \ \mathit{succ}(m')$

By IH we get derivation ∇ of $m' + \mathit{zero}\ \ \mathit{is}\ \ m'$.

By rule:

$$\frac{\begin{array}{c}\nabla \\ m' + \mathit{zero}\ \ \mathit{is}\ \ m'\end{array}}{\mathit{succ}(m') + \mathit{zero}\ \ \mathit{is}\ \ \mathit{succ}(m')}\ \mathit{add}/s$$

Metatheory With Twelf

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \ \mathit{is}\ \ m$.

Metatheory With Twelf

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \ \mathit{is}\ \ m$.

- Proof defines a transformation that generates a derivation of $m + \mathit{zero}\ \ \mathit{is}\ \ m$ for every number m.

Metatheory With Twelf

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \ \mathit{is}\ \ m$.

- Proof defines a transformation that generates a derivation of $m + \mathit{zero}\ \ \mathit{is}\ \ m$ for every number m.
- Can represent transformation as a binary relation that relates each number to a derivation

Metatheory With Twelf

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \ \mathit{is}\ \ m$.

- Proof defines a transformation that generates a derivation of $m + \mathit{zero}\ \ \mathit{is}\ \ m$ for every number m.
- Can represent transformation as a binary relation that relates each number to a derivation
- Show totality of relation to prove the theorem.

Sound familiar?

Metatheory With Twelf

We already know how to define total relations! E.g.

```
add : nat -> nat -> nat -> type.
%mode add +M +N -P.
%total M (add M _ _).
```

Metatheory With Twelf

We already know how to define total relations! E.g.

```
add : nat -> nat -> nat -> type.
%mode add +M +N -P.
%total M (add M _ _).
```

What is a Twelf proof?

- Define a relation as an LF type family
- Get Twelf to prove it `%total`

Representing Theorem Statements

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \mathit{is}\ m$.

i.e.

Lemma: For all `M:nat`, there exists a `D:(add M zero M)`.

Representing Theorem Statements

Lemma: For all *m* **nat**, we can derive *m* + *zero* *is* *m*.

i.e.

Lemma: For all `M:nat`, there exists a `D:(add M zero M)`.

Theorem statement becomes type family, mode, worlds:

```
rhzero : {m : nat} add m zero m -> type.
%mode rhzero +M -D.
%worlds () (rhzero _ _).
```

Representing Proofs

Lemma: For all `M:nat`, there exists a `D:(add M zero M)`.

```
rhzero : {m : nat} add m zero m -> type.
%mode rhzero +M -D.
%worlds () (rhzero _ _).
```

Proof becomes:

- Cases = generators populating the family
- Twelf checks `%total`:

  Lemma: For all `M:nat`, there exists a `D:(add M zero M)`
  and `D':(rhzero M D)`.

First Case

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \ \mathit{is}\ \ m$.

Case for $m = \mathit{zero}$**:**

To show: $\mathit{zero} + \mathit{zero}\ \ \mathit{is}\ \ \mathit{zero}$.

True by rule:

$$\frac{}{\mathit{zero} + \mathit{zero}\ \ \mathit{is}\ \ \mathit{zero}}\ \mathit{add}/z$$

Second Case

Lemma: For all m **nat**, we can derive $m + \mathit{zero}\ \ \mathit{is}\ \ m$.

Case for $m = \mathit{succ}(m')$**:**

To show: $\mathit{succ}(m') + \mathit{zero}\ \ \mathit{is}\ \ \mathit{succ}(m')$

By IH we get derivation ∇ of $m' + \mathit{zero}\ \ \mathit{is}\ \ m'$.

By rule:

$$\frac{\begin{array}{c}\nabla \\ m' + \mathit{zero}\ \ \mathit{is}\ \ m'\end{array}}{\mathit{succ}(m') + \mathit{zero}\ \ \mathit{is}\ \ \mathit{succ}(m')}\ \mathit{add}/s$$

Metatheory in Twelf

We can use this methodology to prove $\forall\exists$-type properties of representations:

$$\forall M_1 : A_1 \ldots \forall M_k : A_k \; \exists N_1 : B_1 \ldots \exists N_l : B_l \; \top$$

This is sufficient for a large body of metareasoning!

Another example

Analogous lemma for successor'ing the right-hand arg:

> **Lemma:** *For all nats* m, n, p*,*
> *if* $m + n$ *is* p *then* $m + succ(n)$ *is* $succ(p)$*.*

Proof: induction over derivation of $m + n$ *is* p.

Exercise

Commutativity: *For all nats* m, n, p*,*
if $m + n$ *is* p *then* $n + m$ *is* p*.*

Hint: do induction over the derivation of $m + n$ *is* p and use previous two lemmas!

Recap and Prospectus

You've now seen all of the basic features of LF and Twelf.

Recap and Prospectus

You've now seen all of the basic features of LF and Twelf.

- Representation: LF signatures for languages and logics.

Recap and Prospectus

You've now seen all of the basic features of LF and Twelf.

- Representation: LF signatures for languages and logics.
- Totality of judgements: mode, worlds, total

Recap and Prospectus

You've now seen all of the basic features of LF and Twelf.

- Representation: LF signatures for languages and logics.
- Totality of judgements: mode, worlds, total
- Metatheory: proofs as total relations

Recap and Prospectus

You've now seen all of the basic features of LF and Twelf.

- Representation: LF signatures for languages and logics.
- Totality of judgements: mode, worlds, total
- Metatheory: proofs as total relations

Recap and Prospectus

You've now seen all of the basic features of LF and Twelf.

- Representation: LF signatures for languages and logics.
- Totality of judgements: mode, worlds, total
- Metatheory: proofs as total relations

Next: apply these tools to study a simple programming language.

Part II

Representing Variable Binding

Syntax for MinML

$$
\begin{array}{lcll}
\tau & ::= & \mathit{num} \mid \tau_1 \Rightarrow \tau_2 & \\
 & & & \\
e & ::= & z \mid s(e) & \text{(constructors for } \mathit{num}\text{)} \\
 & & \mid \mathit{ifz}(e, e_0, x.e_1) & \text{(case-analysis for } \mathit{num}\text{)} \\
 & & \mid \mathit{fn}\, x{:}\tau.e \mid e_1\ e_2 & \text{(functions and application)} \\
 & & \mid x & \text{(variables)}
\end{array}
$$

Write τ **type** and e **exp** for syntax generated by these grammars.

Representing Syntax

Remember the methodology:

- Syntactic category becomes an LF type
- Each rule becomes a generator in the LF signature.

Types are easy:

$$\tau \quad ::= \quad num \mid \tau_1 \Rightarrow \tau_2$$

Let's represent this in Twelf...

Representing Syntax

Encoding function:

$$\begin{aligned} \ulcorner \mathit{num} \urcorner &= \texttt{num} \\ \ulcorner \tau_1 \Rightarrow \tau_2 \urcorner &= \texttt{arr}\ \ulcorner \tau_1 \urcorner\ \ulcorner \tau_2 \urcorner \end{aligned}$$

Adequacy: $\ulcorner \tau \urcorner$ is a bijection between τ **type** and LF terms of type `tp`

Representing Syntax

Expressions:

$$
\begin{array}{rcll}
e & ::= & z \mid s(e) \mid e_1\ e_2 & \text{(easy)} \\
 & & \mid \mathit{ifz}(e, e_0, x.e_1) \mid \mathit{fn}\ x{:}\tau.e & \text{(binding constructs)} \\
 & & \mid x & \text{(variables)}
\end{array}
$$

Let's do the easy ones first...

Representing Syntax

Encoding function:

$$
\begin{array}{rcl}
\ulcorner z \urcorner & = & \mathtt{z} \\
\ulcorner s(e) \urcorner & = & \mathtt{s}\ \ulcorner e \urcorner \\
\ulcorner e_1\ e_2 \urcorner & = & \mathtt{app}\ \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \\
\ulcorner \mathit{fn}\, x{:}\tau.e \urcorner & = & ??? \\
\ulcorner \mathit{ifz}(e, e_0, x.e_1) \urcorner & = & ??? \\
\ulcorner x \urcorner & = & ???
\end{array}
$$

How do we represent binding constructs and variables?

Representing Bindings

fn $x{:}\tau.e$ constructs an expression out of two things:

- domain type τ

Representing Bindings

fn $x{:}\tau.e$ constructs an expression out of two things:

- domain type τ
- binder $x.e$, where the variable x is bound in e

Representing Bindings

fn $x{:}\tau.e$ constructs an expression out of two things:

- domain type τ
- binder $x.e$, where the variable x is bound in e

  x serves as a pronoun referring to the binding site:

Representing Bindings

$\mathit{fn}\, x{:}\tau.e$ constructs an expression out of two things:

- domain type τ
- binder $x.e$, where the variable x is bound in e

  x serves as a pronoun referring to the binding site:
  - May be renamed, preserving pronoun structure:

$$\mathit{fn}\, x{:}\mathit{num}.s(x) \quad \text{is} \quad \mathit{fn}\, y{:}\mathit{num}.s(y).$$

Representing Bindings

$\mathit{fn}\, x{:}\tau.e$ constructs an expression out of two things:

- domain type τ
- binder $x.e$, where the variable x is bound in e

  x serves as a pronoun referring to the binding site:
  - May be renamed, preserving pronoun structure:

    $$\mathit{fn}\, x{:}\mathit{num}.s(x) \quad \text{is} \quad \mathit{fn}\, y{:}\mathit{num}.s(y).$$

  - May be substituted by an expression, preserving pronoun structure:

    $$[s(s(z))/x](s(x)) \quad \text{is} \quad s(s(s(z)))$$

Representing Bindings

Notate the formation of fn with a general judgement:

$$\frac{\tau\ \textbf{type} \quad x\ \textbf{exp} \mid e\ \textbf{exp}}{\mathit{fn}\ x{:}\tau.e\ \textbf{exp}}$$

Representing Bindings

Notate the formation of fn with a general judgement:

$$\frac{\tau\ \textbf{type} \quad x\ \textbf{exp} \mid e\ \textbf{exp}}{\mathit{fn}\ x{:}\tau.e\ \textbf{exp}}$$

- The variable x may occur within e.

Representing Bindings

Notate the formation of fn with a general judgement:

$$\frac{\tau\ \textbf{type} \quad x\ \textbf{exp} \mid e\ \textbf{exp}}{\mathit{fn}\ x{:}\tau.e\ \textbf{exp}}$$

- The variable x may occur within e.
- General judgement can be α-converted

$$(x\ \textbf{exp} \mid e\ \textbf{exp}) \equiv_\alpha (y\ \textbf{exp} \mid [y/x]e\ \textbf{exp})$$

Representing Bindings

Notate the formation of fn with a general judgement:

$$\frac{\tau\ \textbf{type} \quad x\ \textbf{exp} \mid e\ \textbf{exp}}{\mathit{fn}\ x{:}\tau.e\ \textbf{exp}}$$

- The variable x may occur within e.
- General judgement can be α-converted

$$(x\ \textbf{exp} \mid e\ \textbf{exp}) \equiv_\alpha (y\ \textbf{exp} \mid [y/x]e\ \textbf{exp})$$

- Substitution is valid: $[e/x]e_2\ \textbf{exp}$ whenever $e\ \textbf{exp}$.

Higher-Order Abstract Syntax

$$\frac{\tau\ \textbf{type} \quad x\ \textbf{exp} \mid e\ \textbf{exp}}{\mathit{fn}\ x{:}\tau.e\ \textbf{exp}}$$

represented by LF generator:

```
fn :  tp -> (exp -> exp) -> exp.
```

Higher-Order Abstract Syntax

$$\frac{\tau\ \textbf{type} \quad x\ \textbf{exp} \mid e\ \textbf{exp}}{\mathit{fn}\ x{:}\tau.e\ \textbf{exp}}$$

represented by LF generator:

```
fn :  tp -> (exp -> exp) -> exp.
```

Uses higher-order functions to express binding and scope!

Higher-Order Abstract Syntax

$$\frac{\tau\ \textbf{type} \quad x\ \textbf{exp} \mid e\ \textbf{exp}}{\mathit{fn}\ x{:}\tau.e\ \textbf{exp}}$$

represented by LF generator:

```
fn :  tp -> (exp -> exp) -> exp.
```

Uses higher-order functions to express binding and scope!

Expression with a free variable x **exp** $\mid$ e **exp**
represented by
LF function of type `exp -> exp`

Higher-Order Abstract Syntax

Expression with a free variable x **exp** | e **exp**
represented by
LF function of type `exp -> exp`

LF functions `exp -> exp`:

- Intro: `([x:exp] M)` , where `M:exp` assuming `x:exp`

Higher-Order Abstract Syntax

Expression with a free variable x **exp** | e **exp**
represented by
LF function of type `exp -> exp`

LF functions `exp -> exp`:

- Intro: `([x:exp] M)` , where `M:exp` assuming `x:exp`
- Elim: `M1 M2` (like we've been writing all along)

Higher-Order Abstract Syntax

```
fn :  tp -> (exp -> exp) -> exp.
```

Representation:

$$\ulcorner fn\ x{:}\tau.e \urcorner = \texttt{fn}\ulcorner\tau\urcorner\,(\texttt{[x:exp]}\ulcorner e\urcorner)$$
$$\ulcorner x \urcorner = \texttt{x}$$

- LF function represents the binding and scope of x in e
- object-language variables represented by LF variables

Higher-Order Abstract Syntax

```
fn :  tp -> (exp -> exp) -> exp.
```

Representation:

$$\ulcorner fn\ x{:}\tau.e \urcorner = \texttt{fn}\ulcorner\tau\urcorner\,\texttt{([x:exp]}\ulcorner e\urcorner\texttt{)}$$
$$\ulcorner x \urcorner = \texttt{x}$$

- LF function represents the binding and scope of x in e
- object-language variables represented by LF variables

Higher-Order Abstract Syntax

```
fn :  tp -> (exp -> exp) -> exp.
```

Representation:

$$\ulcorner fn\ x{:}\tau.e \urcorner = \texttt{fn}\ulcorner\tau\urcorner\,(\texttt{[x:exp]}\ulcorner e\urcorner)$$
$$\ulcorner x \urcorner = \texttt{x}$$

- LF function represents the binding and scope of x in e
- object-language variables represented by LF variables

Exercise

$$\ulcorner \mathit{ifz}(e, e_0, x.e_1) \urcorner \quad = \quad ???$$

Adequacy

Basic idea: Define a bijection between e **exp** and `M : exp`

Questions:

- How does the bijection treat free variables?

Adequacy

Basic idea: Define a bijection between e **exp** and `M : exp`

Questions:

- How does the bijection treat free variables?
- What is an isomorphism of syntax with binding?

Adequacy

Basic idea: Define a bijection between e **exp** and `M : exp`

Questions:

- How does the bijection treat free variables?
- What is an isomorphism of syntax with binding?
- Do all LF functions of type `exp -> exp` represent syntax?

Part III

LF and Adequacy

LF and Adequacy

1. Simply-typed LF: Canonical forms; substitution
2. Adequacy of exp
3. Dependent types

LF and Adequacy

1. Simply-typed LF: Canonical forms; substitution
2. Adequacy of exp
3. Dependent types

The LF Type Theory

LF is a dependently typed λ-calculus

Simply typed fragment:

Types	`A` $::=$ `a` $\mid$ `A1 -> A2`
Canonical forms	`M` $::=$ `x M1...Mn` $\mid$ `c M1...Mn` $\mid$ `[x] M`

The LF Type Theory

LF is a dependently typed λ-calculus

Simply typed fragment:

Types	`A` $::=$ `a` $\mid$ `A1 -> A2`
Canonical forms	`M` $::=$ `x M1...Mn` $\mid$ `c M1...Mn` $\mid$ `[x] M`

Base types like `nat`, `tp`, `exp`

The LF Type Theory

LF is a dependently typed λ-calculus

Simply typed fragment:

Types	`A` $::=$ `a` $\mid$ `A1 -> A2`
Canonical forms	`M` $::=$ `x M1...Mn` $\mid$ `c M1...Mn` $\mid$ `[x] M`

Function types like `exp -> exp`

The LF Type Theory

LF is a dependently typed λ-calculus

Simply typed fragment:

Types	`A` $::=$ `a` $\mid$ `A1 -> A2`
Canonical forms	`M` $::=$ `x M1...Mn` $\mid$ `c M1...Mn` $\mid$ `[x] M`

Variable or constant applied to canonical forms (no β-redices)

The LF Type Theory

LF is a dependently typed λ-calculus

Simply typed fragment:

Types	`A` $::=$ `a` $\mid$ `A1 -> A2`
Canonical forms	`M` $::=$ `x M1...Mn` $\mid$ `c M1...Mn` $\mid$ `[x] M`

λ-abstraction

Canonical Forms

Types	$\texttt{A} ::= \texttt{a} \mid \texttt{A1 -> A2}$
Canonical forms	$\texttt{M} ::= \texttt{x M1}\ldots\texttt{Mn} \mid \texttt{c M1}\ldots\texttt{Mn} \mid \texttt{[x] M}$

Signature	$\Sigma ::= \varepsilon \mid \Sigma, \texttt{a:type} \mid \Sigma, \texttt{c:A}$
Context	$\Gamma ::= \varepsilon \mid \Gamma, \texttt{x:A}$

Canonical Forms

Types	$\texttt{A} ::= \texttt{a} \mid \texttt{A1 -> A2}$
Canonical forms	$\texttt{M} ::= \texttt{x M1}\ldots\texttt{Mn} \mid \texttt{c M1}\ldots\texttt{Mn} \mid \texttt{[x] M}$

Signature	$\Sigma ::= \varepsilon \mid \Sigma, \texttt{a:type} \mid \Sigma, \texttt{c:A}$
Context	$\Gamma ::= \varepsilon \mid \Gamma, \texttt{x:A}$

Base types (`nat`) and constants (`zero`, `succ`) declared in signature

Canonical Forms

Types	$\texttt{A} ::= \texttt{a} \mid \texttt{A1 -> A2}$
Canonical forms	$\texttt{M} ::= \texttt{x M1} \ldots \texttt{Mn} \mid \texttt{c M1} \ldots \texttt{Mn} \mid \texttt{[x] M}$

Signature	$\Sigma ::= \varepsilon \mid \Sigma, \texttt{a:type} \mid \Sigma, \texttt{c:A}$
Context	$\Gamma ::= \varepsilon \mid \Gamma, \texttt{x:A}$

Variables bound in context

Canonical Forms

Types	$\texttt{A} ::= \texttt{a} \mid \texttt{A1 -> A2}$
Canonical forms	$\texttt{M} ::= \texttt{x M1}\ldots\texttt{Mn} \mid \texttt{c M1}\ldots\texttt{Mn} \mid \texttt{[x] M}$
Signature	$\Sigma ::= \varepsilon \mid \Sigma, \texttt{a:type} \mid \Sigma, \texttt{c:A}$
Context	$\Gamma ::= \varepsilon \mid \Gamma, \texttt{x:A}$

Inductive definition of canonical forms:

$$\Gamma \vdash_\Sigma \texttt{M} : \texttt{A}$$

`M` is a canonical form of type `A` in signature Σ and context Γ

Canonical Forms

Constants:

$$\frac{\texttt{c:(A1 ->...-> An -> a)} \in \Sigma \quad \Gamma \vdash_\Sigma \texttt{Mi} : \texttt{Ai}}{\Gamma \vdash_\Sigma \texttt{c M1...Mn} : \texttt{a}}$$

Canonical Forms

Constants:

$$\frac{\texttt{c:(A1 ->...-> An -> a)} \in \Sigma \quad \Gamma \vdash_\Sigma \texttt{Mi} : \texttt{Ai}}{\Gamma \vdash_\Sigma \texttt{c M1...Mn} : \texttt{a}}$$

Example:

$$\frac{\texttt{arr:(tp -> tp -> tp)} \in \Sigma \quad \dfrac{\vdots}{\varepsilon \vdash_\Sigma \texttt{num} : \texttt{tp}} \quad \dfrac{\vdots}{\varepsilon \vdash_\Sigma \texttt{num} : \texttt{tp}}}{\varepsilon \vdash_\Sigma \texttt{(arr num num)} : \texttt{tp}}$$

Canonical Forms

Variables:

$$\frac{\texttt{x:(A1 ->...-> An -> a)} \in \Gamma \quad \Gamma \vdash_{\Sigma} \texttt{Mi} : \texttt{Ai}}{\Gamma \vdash_{\Sigma} \texttt{x M1...Mn} : \texttt{a}}$$

Canonical Forms

Variables:

$$\frac{\texttt{x:(A1 ->...-> An -> a)} \in \Gamma \quad \Gamma \vdash_\Sigma \texttt{Mi} : \texttt{Ai}}{\Gamma \vdash_\Sigma \texttt{x M1...Mn} : \texttt{a}}$$

Example:

$$\frac{}{\texttt{x:exp} \vdash_\Sigma \texttt{x} : \texttt{exp}}$$

Canonical Forms

Functions:

$$\frac{\Gamma, \texttt{x:A} \vdash_\Sigma \texttt{M} : \texttt{B}}{\Gamma \vdash_\Sigma \texttt{([x] M)} : \texttt{A -> B}}$$

Canonical Forms

Functions:

$$\frac{\Gamma, \texttt{x:A} \vdash_\Sigma \texttt{M} : \texttt{B}}{\Gamma \vdash_\Sigma \texttt{([x] M)} : \texttt{A -> B}}$$

Example:

$$\frac{\Gamma, \texttt{x:exp} \vdash_\Sigma \texttt{x} : \texttt{exp}}{\Gamma \vdash_\Sigma \texttt{([x] x)} : \texttt{exp -> exp}}$$

Canonical Forms

Functions:

$$\frac{\Gamma, \mathtt{x}:\mathtt{A} \vdash_\Sigma \mathtt{M} : \mathtt{B}}{\Gamma \vdash_\Sigma \mathtt{([x]\ M)} : \mathtt{A\ ->\ B}}$$

Example:

$$\frac{\Gamma, \mathtt{x}:\mathtt{exp} \vdash_\Sigma \mathtt{x} : \mathtt{exp}}{\Gamma \vdash_\Sigma \mathtt{([x]\ x)} : \mathtt{exp\ ->\ exp}}$$

This is the only rule for `A -> B`, so canonical forms are η-expanded

Induction on Canonical Forms

Canonical forms $\quad$ `M` $::=$ `x M1 ... Mn` $\mid$ `c M1 ... Mn` $\mid$ `[x] M`

Adequacy and metatheory are based on structural induction on canonical forms

Example: What are the canonical forms of type `exp`?

Canonical Forms of exp

What are the canonical forms of type exp
in context x_1:exp, . . . , x_n:exp?

Canonical Forms of exp

What are the canonical forms of type exp
in context x_1:exp, . . . , x_n:exp?

- x_i

Canonical Forms of `exp`

What are the canonical forms of type `exp`
in context x_1:`exp`, . . . , x_n:`exp`?

- x_i
- `app M1 M2` where `M1` and `M2` are canonical at `exp`

Canonical Forms of `exp`

What are the canonical forms of type `exp`
in context x_1:`exp`, ..., x_n:`exp`?

- x_i
- `app M1 M2` where `M1` and `M2` are canonical at `exp`
- `fn M1 M2` where `M1:tp` and `M2:(exp -> exp)` are canon.

Canonical Forms of `exp`

What are the canonical forms of type `exp`
in context x_1:`exp`, . . . , x_n:`exp`?

- x_i
- `app M1 M2` where `M1` and `M2` are canonical at `exp`
- `fn M1 M2` where `M1:tp` and `M2:(exp -> exp)` are canon.
  `fn M1 ([x] M)` where `M` is canonical assuming `x:exp`

No "exotic terms"!

Canonical Forms of `exp`

What are the canonical forms of type `exp`
in context x_1:`exp`, . . . , x_n:`exp`?

- x_i
- `app M1 M2` where `M1` and `M2` are canonical at `exp`
- `fn M1 M2` where `M1:tp` and `M2:(exp -> exp)` are canon.
  `fn M1 ([x] M)` where `M` is canonical assuming `x:exp`
- `z`

No "exotic terms"!

Canonical Forms of `exp`

What are the canonical forms of type `exp`
in context x_1:`exp`, . . . , x_n:`exp`?

- x_i
- `app M1 M2` where `M1` and `M2` are canonical at `exp`
- `fn M1 M2` where `M1:tp` and `M2:(exp -> exp)` are canon.
  `fn M1 ([x] M)` where `M` is canonical assuming `x:exp`
- `z`
- `s M` where `M` is canonical at `exp`

No "exotic terms"!

Canonical Forms of `exp`

What are the canonical forms of type `exp`
in context x_1:`exp`, ..., x_n:`exp`?

- x_i
- `app M1 M2` where `M1` and `M2` are canonical at `exp`
- `fn M1 M2` where `M1:tp` and `M2:(exp -> exp)` are canon.
  `fn M1 ([x] M)` where `M` is canonical assuming `x:exp`
- `z`
- `s M` where `M` is canonical at `exp`
- `ifz M M0 M1` where `M,M0` canonical at `exp`
  and `M1` canonical at `(exp -> exp)`

No "exotic terms"!

LF Substitution

$$\texttt{[M' / x] M}$$

where $\Gamma, \texttt{x:A} \vdash_\Sigma \texttt{M} : \texttt{B}$ and $\Gamma \vdash_\Sigma \texttt{M'} : \texttt{A}$

LF Substitution

$$\texttt{[M' / x] M}$$

where $\Gamma, \texttt{x:A} \vdash_\Sigma \texttt{M} : \texttt{B}$ and $\Gamma \vdash_\Sigma \texttt{M'} : \texttt{A}$

But canonical forms are not closed under substitution:

$$\begin{aligned} \texttt{A} &= \texttt{exp -> exp} \\ \texttt{M'} &= \texttt{[x] x} \\ \texttt{M} &= \texttt{x N} \end{aligned}$$

Substitution results in β-redex `([x] x) N` !

LF Substitution

$$\texttt{[M' / x] M}$$

where $\Gamma, \texttt{x:A} \vdash_\Sigma \texttt{M} : \texttt{B}$ and $\Gamma \vdash_\Sigma \texttt{M'} : \texttt{A}$

But canonical forms are not closed under substitution

Two solutions:

LF Substitution

$$\texttt{[M' / x] M}$$

where $\Gamma, \texttt{x:A} \vdash_\Sigma \texttt{M} : \texttt{B}$ and $\Gamma \vdash_\Sigma \texttt{M'} : \texttt{A}$

But canonical forms are not closed under substitution

Two solutions:

- Old: Allow non-canonical forms, use $\beta\eta$-equality

LF Substitution

$$[\ \mathtt{M'}\ /\ \mathtt{x}\ ]\ \mathtt{M}$$

where $\Gamma, \mathtt{x}:\mathtt{A} \vdash_\Sigma \mathtt{M} : \mathtt{B}$ and $\Gamma \vdash_\Sigma \mathtt{M'} : \mathtt{A}$

But canonical forms are not closed under substitution

Two solutions:

- Old: Allow non-canonical forms, use $\beta\eta$-equality
- New: Use hereditary substitution to directly compute the canonical result of substitution

LF Substitution

$$[\ \texttt{M'}\ /\ \texttt{x}\ ]\ \texttt{M}$$

where $\Gamma, \texttt{x:A} \vdash_\Sigma \texttt{M} : \texttt{B}$ and $\Gamma \vdash_\Sigma \texttt{M'} : \texttt{A}$

But canonical forms are not closed under substitution

Two solutions:

- Old: Allow non-canonical forms, use $\beta\eta$-equality
- New: Use hereditary substitution to directly compute the canonical result of substitution

Twelf: allows non-canonical forms in source text;
uses hereditary substitution under the hood

LF and Adequacy

1. Simply-typed LF: Canonical forms; substitution
2. Adequacy of `exp`
3. Dependent types

Adequacy

Basic idea: Define a bijection between e **exp** and `M : exp`

Questions:

- How does the bijection treat free variables?
- What is an isomorphism of syntax with binding?
- Do all LF functions of type `exp -> exp` represent syntax?

Adequacy

Basic idea: Define a bijection between e **exp** and `M : exp`

Questions:

- How does the bijection treat free variables?
- What is an isomorphism of syntax with binding?
- Do all LF functions of type `exp -> exp` represent syntax?

Adequacy for expressions with binding

Isomorphism specified by

- A context-indexed family of bijections

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

Adequacy for expressions with binding

Isomorphism specified by

- A context-indexed family of bijections

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$
$$\longleftrightarrow$$
$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

- That is compositional: respects substitution

$$\ulcorner [e/x]e' \urcorner = [\ulcorner e \urcorner / x] \ulcorner e' \urcorner$$

Encoding of Syntax

Define a family of bijections

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

Encoding of Syntax

Define a family of bijections

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

Variables:

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid x_i\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} x_i : \mathsf{exp}$$

Encoding of Syntax

Application:

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e_1\ e_2\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \mathsf{app}\ulcorner e_1 \urcorner\ulcorner e_2 \urcorner$$

where

Encoding of Syntax

Application:

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e_1\ e_2\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \mathsf{app}\ulcorner e_1 \urcorner\ulcorner e_2 \urcorner$$

where

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e_1\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e_1 \urcorner{:}\mathsf{exp}$$

and

Encoding of Syntax

Application:

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e_1\ e_2\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \mathsf{app}\ulcorner e_1 \urcorner\ulcorner e_2 \urcorner$$

where

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e_1\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e_1 \urcorner{:}\mathsf{exp}$$

and

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e_2\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e_2 \urcorner{:}\mathsf{exp}$$

Encoding of Syntax

Functions:

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid \mathit{fn}\ x{:}\tau.e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \mathsf{fn}\ \ulcorner\tau\urcorner\,([x{:}\mathsf{exp}]\ulcorner e\urcorner)$$

where

Encoding of Syntax

Functions:

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid \mathit{fn}\ x{:}\tau.e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \mathsf{fn}\ \ulcorner\tau\urcorner\ ([x{:}\mathsf{exp}]\ulcorner e\urcorner)$$

where

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid \tau\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner\tau\urcorner{:}\mathsf{tp}$$

and

Encoding of Syntax

Functions:

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid \mathit{fn}\ x{:}\tau.e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \mathsf{fn}\ \ulcorner\tau\urcorner\ ([x{:}\mathsf{exp}]\ulcorner e\urcorner)$$

where

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid \tau\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner\tau\urcorner{:}\mathsf{tp}$$

and

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp}, x\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}, x{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e\urcorner{:}\mathsf{exp}$$

Encoding of Syntax

Functions:

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid \mathit{fn}\ x{:}\tau.e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \mathsf{fn}\ \ulcorner\tau\urcorner\ (\lceil x{:}\mathsf{exp}\rceil \ulcorner e\urcorner)$$

where

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid \tau\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner\tau\urcorner{:}\mathsf{tp}$$

and

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp}, x\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}, x{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e\urcorner{:}\mathsf{exp}$$

Compositionality

Demand that encoding commutes with substitution:

if x **exp** $\mid e'$ **exp** and e **exp**, then $\ulcorner [e/x]e' \urcorner = [\ulcorner e \urcorner / x] \ulcorner e' \urcorner$.

where

Compositionality

Demand that encoding commutes with substitution:

$$\text{if } x\ \mathbf{exp} \mid e'\ \mathbf{exp} \text{ and } e\ \mathbf{exp},\ \text{then } \ulcorner [e/x]e' \urcorner = [\ulcorner e \urcorner / x] \ulcorner e' \urcorner.$$

where

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}$$

and

Compositionality

Demand that encoding commutes with substitution:

$$\text{if } x\ \mathbf{exp} \mid e'\ \mathbf{exp} \text{ and } e\ \mathbf{exp}, \text{ then } \ulcorner [e/x]e' \urcorner = [\ulcorner e \urcorner / x] \ulcorner e' \urcorner.$$

where

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}$$

and

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp}, x\ \mathbf{exp} \mid e'\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}, x{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e' \urcorner : \mathsf{exp}$$

Compositionality

$$\ulcorner [e/x]e' \urcorner = [\ulcorner e \urcorner / x] \ulcorner e' \urcorner.$$

Proves that object language's notion of substitution is faithfully represented by LF substitution

Proving Adequacy

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

- $\ulcorner e \urcorner$ is a function Proof: induction on e

Proving Adequacy

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

- $\ulcorner e \urcorner$ is a function Proof: induction on e
- Compositionality Proof: induction on e

Proving Adequacy

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

- $\ulcorner e \urcorner$ is a function Proof: induction on e
- Compositionality Proof: induction on e
- $\ulcorner e \urcorner$ is injective Proof: induction on e

Proving Adequacy

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

- $\ulcorner e \urcorner$ is a function Proof: induction on e
- Compositionality Proof: induction on e
- $\ulcorner e \urcorner$ is injective Proof: induction on e
- $\ulcorner e \urcorner$ is surjective Proof: induction on canonical forms

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} M : \mathsf{exp}.$$

Canonical Forms of `exp`

What are the canonical forms of type `exp`
in context x_1:`exp`, ..., x_n:`exp`?

- x_i
- `app M1 M2` where `M1` and `M2` are canonical at `exp`
- `fn M1 M2` where `M1:tp` and `M2:(exp -> exp)` are canon.

  `fn M1 ([x] M)` where `M` is canonical assuming `x:exp`
- `z`
- `s M` where `M` is canonical at `exp`
- `ifz M M0 M1` where `M,M0` canonical at `exp`
  and `M1` canonical at `(exp -> exp)`

Encoding of Syntax

Suppose we allow

`f:(exp -> exp)`, `x1:exp`,...,`xn:exp`

Surjectivity fails: new canonical forms

- `f zero`

Encoding of Syntax

Suppose we allow

```
f:(exp -> exp), x1:exp,...,xn:exp
```

Surjectivity fails: new canonical forms

- `f zero`
- `ifz M M0 ([x] f x)`

Encoding of Syntax

Suppose we allow

```
f:(exp -> exp), x1:exp,...,xn:exp
```

Surjectivity fails: new canonical forms

- `f zero`
- `ifz M M0 ([x] f x)`

Lesson: adequacy depends crucially on the world (set of LF contexts)

Adequacy

Basic idea: Define a bijection between e **exp** and `M : exp`

Questions:

- How does the bijection treat free variables?
- What is an isomorphism of syntax with binding?
- Do all LF functions of type `exp -> exp` represent syntax?

No "exotic terms"

Suppose context has the form `x1:exp`, . . . , `xn:exp`

- By canonical forms, every term `exp -> exp` is `([x:exp] M)`

No "exotic terms"

Suppose context has the form `x1:exp`, ..., `xn:exp`

- By canonical forms, every term `exp -> exp` is `([x:exp] M)`
- `M` has type

$$\texttt{x1:exp}, \ldots, \texttt{xn:exp}, \texttt{x:exp} \vdash_{\Sigma_{\text{exp}}} \texttt{M:exp}$$

No "exotic terms"

Suppose context has the form `x1:exp`, ..., `xn:exp`

- By canonical forms, every term `exp -> exp` is `([x:exp] M)`
- `M` has type

$$\texttt{x1:exp}, \ldots, \texttt{xn:exp}, \texttt{x:exp} \vdash_{\Sigma_{\text{exp}}} \texttt{M:exp}$$

- Every such `M` represents an expression with free vars `x1`...`xn`,`x`

No "exotic terms"

Suppose context has the form `x1:exp`, ..., `xn:exp`

- By canonical forms, every term `exp -> exp` is `([x:exp] M)`
- `M` has type

$$\texttt{x1:exp}, \ldots, \texttt{xn:exp}, \texttt{x:exp} \vdash_{\Sigma_{\text{exp}}} \texttt{M:exp}$$

- Every such `M` represents an expression with free vars `x1`...`xn`,`x`
- So `([x] M)` represents an expression with a bound variable

No "exotic terms"

Suppose context has the form `x1:exp, ..., xn:exp`

- By canonical forms, every term `exp -> exp` is `([x:exp] M)`
- `M` has type

$$\texttt{x1:exp}, \ldots, \texttt{xn:exp}, \texttt{x:exp} \vdash_{\Sigma_{\text{exp}}} \texttt{M:exp}$$

- Every such `M` represents an expression with free vars `x1...xn,x`
- So `([x] M)` represents an expression with a bound variable

Intuitive explanation: `exp` is a base type, so the only thing you can do with `x:exp` is use it (e.g., no case analysis)

LF and Adequacy

1. Simply-typed LF: Canonical forms; substitution
2. Adequacy of `exp`
3. Dependent types

The LF Type Theory

Dependently-typed LF:

Canonical forms	`M` ::= `x M1...Mn` | `c M1...Mn` | `[x] M`
Types	`A` ::= `a` | `A1 -> A2`
Kinds	`K` ::= `type` | `{x:A} K`

The LF Type Theory

Dependently-typed LF:

Canonical forms	`M` ::= `x M1...Mn` | `c M1...Mn` | `[x] M`
Types	`A` ::= `a M1...Mn` | `A1 -> A2`
Kinds	`K` ::= `type` | `{x:A} K`

Family constants applied to args: `add M N P`

The LF Type Theory

Dependently-typed LF:

Canonical forms	`M ::= x M1...Mn | c M1...Mn | [x] M`
Types	`A ::= a M1...Mn | {x:A} B`
Kinds	`K ::= type | {x:A} K`

Dependent function types:

`add/z : {n:nat} add zero n n`

The LF Type Theory

Dependently-typed LF:

Canonical forms	$\texttt{M} ::= \texttt{x M1}\ldots\texttt{Mn} \mid \texttt{c M1}\ldots\texttt{Mn} \mid \texttt{[x] M}$
Types	$\texttt{A} ::= \texttt{a M1}\ldots\texttt{Mn} \mid \texttt{\{x:A\} B}$
Kinds	$\texttt{K} ::= \texttt{type} \mid \texttt{\{x:A\} K}$

Kinds of family constants:

```
rhzero :  {n:nat} {_:add n zero n} type
```

The LF Type Theory

Dependently-typed LF:

Canonical forms	`M` ::= `x M1...Mn` | `c M1...Mn` | `[x] M`
Types	`A` ::= `a M1...Mn` | `{x:A} B`
Kinds	`K` ::= `type` | `{x:A} K`

Kinds of family constants:

```
rhzero :  {n:nat} add n zero n -> type
```

(`A -> B` means `{_:A} B`)

Dependent Application

$$\frac{\Gamma \vdash_\Sigma \texttt{M1} : \{\texttt{x:A}\}\ \texttt{B} \quad \Gamma \vdash_\Sigma \texttt{M2} : \texttt{A}}{\Gamma \vdash_\Sigma (\texttt{M1}\ \texttt{M2}) : [\texttt{M2/x}]\texttt{B}}$$

Dependent Application

$$\frac{\Gamma \vdash_\Sigma \texttt{M1} : \{\texttt{x:A}\}\ \texttt{B} \quad \Gamma \vdash_\Sigma \texttt{M2} : \texttt{A}}{\Gamma \vdash_\Sigma \texttt{(M1 M2)} : \texttt{[M2/x]B}}$$

Example:

```
add/z : {n:nat} add zero n n
0+2=2 : add zero 2 2 = add/z 2
```

Dependent Application

$$\frac{\Gamma \vdash_\Sigma \texttt{M1} : \{\texttt{x:A}\}\ \texttt{B} \quad \Gamma \vdash_\Sigma \texttt{M2} : \texttt{A}}{\Gamma \vdash_\Sigma \texttt{(M1 M2)} : \texttt{[M2/x]B}}$$

Example:

```
add/s : {m:nat} {n:nat} {p:nat}
           add m n p
        -> add (succ m) n (succ p)
partial : add 0 1 2 -> add 1 1 3
  = [d] add/s 0 1 2 d
```

Summary

- $\Gamma \vdash_\Sigma \mathtt{M} : \mathtt{A}$

  Inductive definition of canonical forms $\mathtt{M}$ of type $\mathtt{A}$,
  in context Γ and signature Σ

Summary

- $\Gamma \vdash_\Sigma \mathtt{M} : \mathtt{A}$

  Inductive definition of canonical forms $\mathtt{M}$ of type $\mathtt{A}$,
  in context Γ and signature Σ

- Adequacy: compositional bijection between

  object-language syntax
  and
  canonical forms of a specified type in specified contexts

$$x_1\ \mathbf{exp}, \ldots, x_n\ \mathbf{exp} \mid e\ \mathbf{exp}$$

$$\longleftrightarrow$$

$$x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash_{\Sigma_{\mathsf{exp}}} \ulcorner e \urcorner : \mathsf{exp}.$$

Consequences of Adequacy

An adequate representation obviates the object language itself!

Consequences of Adequacy

An adequate representation obviates the object language itself!

That is, the object language exists solely as embedded in LF; all other representations are nugatory.

Consequences of Adequacy

An adequate representation obviates the object language itself!

That is, the object language exists solely as embedded in LF; all other representations are nugatory.

Representation in LF becomes normative for representations of object languages.

Consequences of Adequacy

An adequate representation obviates the object language itself!

That is, the object language exists solely as embedded in LF; all other representations are nugatory.

Representation in LF becomes normative for representations of object languages.

Experience has shown that it improves our understanding of an object language to formalize it in LF.

Consequences of Adequacy

An adequate representation obviates the object language itself!

That is, the object language exists solely as embedded in LF; all other representations are nugatory.

Representation in LF becomes normative for representations of object languages.

Experience has shown that it improves our understanding of an object language to formalize it in LF.

Metatheory can be mechanized as proofs by induction on canonical forms.

Induction on Canonical Forms

```
add : nat -> nat -> nat -> type
%total M (add M _ _ _).
```

Proof: induction on canonical forms of type `nat`
(= structural induction on syntax)

Induction on Canonical Forms

```
rhsucc : add M N P -> add M (succ N) (succ P) -> type.
%total D (rhsucc D _).
```

Proof: induction on canonical forms of type `add M N P`
(= rule induction on derivations)

Part IV

Representing Hypothetical Judgements

Judgements

Static Semantics:

- $\Gamma \vdash e : \tau$

Dynamic Semantics:

- e **val**
- $e \mapsto e'$

Static semantics

Static semantics for natural numbers

$$\frac{}{\Gamma \vdash z : num}\ of/z \qquad \frac{\Gamma \vdash e : num}{\Gamma \vdash s(e) : num}\ of/s$$

$$\frac{\Gamma \vdash e : num \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : num \vdash e_1 : \tau}{\Gamma \vdash ifz(e, e_0, x.e_1) : \tau}\ of/ifz$$

Static semantics for functions and application

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ of/app \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash fn\ x{:}\tau_1.e : \tau_1 \to \tau_2}\ of/fn$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ dx$$

Higher-Order Rules

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2}$$

Higher-Order Rules

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e : \tau_2}{\mathit{fn}\ x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2}$$

Really a general, hypothetical judgement:

Higher-Order Rules

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathit{fn}\, x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e : \tau_2}{\mathit{fn}\, x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2}$$

Really a general, hypothetical judgement:

- General in a fresh parameter x

Higher-Order Rules

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathit{fn}\, x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e : \tau_2}{\mathit{fn}\, x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2}$$

Really a general, hypothetical judgement:

- General in a fresh parameter x
- Hypothetial in a new axiom stating that x has type τ_1

Higher-Order Metavariables

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathit{fn}\, x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e : \tau_2}{\mathit{fn}\, x{:}\tau_1 . e : \tau_1 \Rightarrow \tau_2}$$

- e is a metavar standing for *terms that may mention* x

Higher-Order Metavariables

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\, x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\, x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2}$$

- e is a metavar standing for *terms that may mention* x
- Can make this explicit by writing e_x

Higher-Order Metavariables

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\, x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\, x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2}$$

- e is a metavar standing for *terms that may mention* x
- Can make this explicit by writing e_x
- Look at $e\_$ in isolation as a higher-order metavariable

Hypothetical Judgements in LF

$$\Gamma \vdash e : \tau$$

Idea: just as with syntax, use LF variables to represent hypotheses

Hypothetical Judgements in LF

$$\Gamma \vdash e : \tau$$

Idea: just as with syntax, use LF variables to represent hypotheses

- Typing judgement is not a three-place relation on Γ, e, and τ

Hypothetical Judgements in LF

$$\Gamma \vdash e : \tau$$

Idea: just as with syntax, use LF variables to represent hypotheses

- Typing judgement is not a three-place relation on Γ, e, and τ
- Binary relation on e and τ, where Γ is represented by the LF context

Hypothetical Judgements in LF

$$\Gamma \vdash e : \tau$$

Idea: just as with syntax, use LF variables to represent hypotheses

- Typing judgement is not a three-place relation on Γ, e, and τ
- Binary relation on e and τ, where Γ is represented by the LF context

```
of :  exp -> tp -> type.
```

Higher-Order Rules in LF

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2}$$

Higher-order rules are represented using higher-order types

```
of/fn : of (fn T1 ([x] E x)) (arr T1 T2)
        <- ({x:exp} of x T1 -> of (E x) T2).
```

Higher-Order Rules in LF

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2}$$

represented by

```
of/fn : of (fn T1 ([x] E x)) (arr T1 T2)
        <- ({x:exp} of x T1 -> of (E x) T2).
```

- `T1:tp` and `T2:tp` correspond to τ_1 and τ_2

Higher-Order Rules in LF

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2}$$

represented by

```
of/fn : of (fn T1 ([x] E x)) (arr T1 T2)
        <- ({x:exp} of x T1 -> of (E x) T2).
```

- `T1:tp` and `T2:tp` correspond to τ_1 and τ_2
- `E:(exp -> exp)` corresponds to higher-order metavar $e\_$

Higher-Order Rules in LF

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2}$$

represented by

```
of/fn : of (fn T1 ([x] E x)) (arr T1 T2)
        <- ({x:exp} of x T1 -> of (E x) T2).
```

Higher-Order Rules in LF

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2}$$

represented by

```
of/fn : of (fn T1 ([x] E x)) (arr T1 T2)
        <- ({x:exp} of x T1 -> of (E x) T2).
```

General, hypothetical judgement: body is typed relative to

Higher-Order Rules in LF

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1.e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\ x{:}\tau_1.e_x : \tau_1 \Rightarrow \tau_2}$$

represented by

```
of/fn : of (fn T1 ([x] E x)) (arr T1 T2)
        <- ({x:exp} of x T1 -> of (E x) T2).
```

General, hypothetical judgement: body is typed relative to

- A fresh variable `x`

Higher-Order Rules in LF

$$\frac{\Gamma, x : \tau_1 \vdash e_x : \tau_2}{\Gamma \vdash \mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2} \quad \text{i.e.} \quad \frac{x\ \mathbf{exp} \mid x : \tau_1 \vdash e_x : \tau_2}{\mathit{fn}\ x{:}\tau_1 . e_x : \tau_1 \Rightarrow \tau_2}$$

represented by

```
of/fn : of (fn T1 ([x] E x)) (arr T1 T2)
        <- ({x:exp} of x T1 -> of (E x) T2).
```

General, hypothetical judgement: body is typed relative to

- A fresh variable `x`
- A new axiom stating that `x` has type `E1`

Static semantics

Static semantics for natural numbers

$$\frac{}{\Gamma \vdash z : num}\ of/z \qquad \frac{\Gamma \vdash e : num}{\Gamma \vdash s(e) : num}\ of/s$$

$$\frac{\Gamma \vdash e : num \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : num \vdash e_1 : \tau}{\Gamma \vdash ifz(e, e_0, x.e_1) : \tau}\ of/ifz$$

Static semantics for functions and application

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ of/app \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash fn\ x{:}\tau_1.e : \tau_1 \to \tau_2}\ of/fn$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ dx$$

Adequacy and Worlds

Typing introduces parameters and hypotheses

Consider a world (set of contexts) consisting of blocks of the form

$$\texttt{x:exp}, \texttt{dx:of x T} \qquad \text{(for some } \texttt{T:tp}\text{)}$$

Adequacy relative to hypotheses in this world:

$$\nabla \text{ derives } x_1 : \tau_1, \ldots \vdash e : \tau$$
$$\text{iff}$$
$$\texttt{x1:exp}, \texttt{dx1:of x1}\ulcorner\tau_1\urcorner, \ldots \vdash \ulcorner\nabla\urcorner : \texttt{of}\ulcorner e\urcorner\ulcorner\tau\urcorner$$

Adequacy and Worlds

Worlds are declared in Twelf using %block and %worlds:

```
%block of_block
   :                block {x:exp} {dx:of x T}.
%worlds (of_block) (of _ _).
```

Twelf checks: all assumptions made in rules are have specified form

Adequacy and Worlds

Worlds are declared in Twelf using %block and %worlds:

```
%block of_block
   : some {T:tp} block {x:exp} {dx:of x T}.
%worlds (of_block) (of _ _).
```

Twelf checks: all assumptions made in rules are have specified form

Dynamic semantics

Dynamic semantics for functions and application

$$\frac{}{\mathit{fn}\ x{:}\tau.e\ \mathbf{val}}\ \textit{value/fn}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \textit{step/app/fn} \qquad \frac{e_1\ \mathbf{val} \qquad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \textit{step/app/arg}$$

$$\frac{e_2\ \mathbf{val}}{(\mathit{fn}\ x{:}\tau.e)\ e_2 \mapsto [e_2/x]e}\ \textit{step/app/beta}$$

Dynamic semantics

Dynamic semantics for natural numbers

$$\frac{}{z\ \mathbf{val}}\ \textit{value/z} \qquad \frac{e\ \mathbf{val}}{s(e)\ \mathbf{val}}\ \textit{value/s}$$

$$\frac{e \mapsto e'}{s(e) \mapsto s(e')}\ \textit{step/s} \qquad \frac{e \mapsto e'}{\mathit{ifz}(e, e_0, x.e_1) \mapsto \mathit{ifz}(e', e_0, x.e_1)}\ \textit{step/ifz/arg}$$

$$\frac{}{\mathit{ifz}(z, e_0, x.e_1) \mapsto e_0}\ \textit{step/ifz/z}$$

$$\frac{e\ \mathbf{val}}{\mathit{ifz}(s(e), e_0, x.e_1) \mapsto [e/x]e_1}\ \textit{step/ifz/s}$$

Part V

Type Safety for MinML

Type Safety

Progress: If $\cdot \vdash e : \tau$, then e **val** or else $e \mapsto e'$ (for some e').

Preservation: If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$

Progress theorem

Progress: If $\cdot \vdash e : \tau$, then e **val** or else $e \mapsto e'$ (for some e').

Proof: induction on derivation of $\cdot \vdash e : \tau$.

Progress theorem (zero)

If $\cdot \vdash e : \tau$, then e **val** or else $e \mapsto e'$ (for some e').

Case: $\dfrac{}{z : num}\ \textit{of/z}$
$(e = z, \tau = num)$

- By rule *value/z*, z **val**

Progress theorem (fn)

If $\cdot \vdash e : \tau$, then e **val** or else $e \mapsto e'$ (for some e').

Case: $\dfrac{x : \tau_1 \vdash e_0 : \tau_2}{\mathit{fn}\, x{:}\tau.e_0 : \tau_1 \to \tau_2}\ \mathit{of/fn}$

$(e = \mathit{fn}\, x{:}\tau.e_0, \tau = \tau_1 \to \tau_2)$

- By rule *value/fn*, $\mathit{fn}\, x{:}\tau.e_0$ **val**

Progress theorem (app)

If $\cdot \vdash e : \tau$, then e **val** or else $e \mapsto e'$ (for some e').

Case: $\dfrac{e_1 : \tau' \to \tau \qquad e_2 : \tau'}{e_1\ e_2 : \tau}$ *of/app*

$(e = e_1\ e_2)$

- By the induction hypothesis, e_1 **val** or else $e_1 \mapsto e_1'$, and additionally e_2 **val** or else $e_2 \mapsto e_2'$
- Case analysis to prove that $e_1\ e_2 \mapsto e'$:
  - If $e_1 \mapsto e_1'$, then $e_1\ e_2 \mapsto e_1'\ e_2$ by rule *step/app/fn*
  - If e_1 **val** and $e_2 \mapsto e_2'$, then $e_1\ e_2 \mapsto e_1\ e_2'$ by rule *step/app/arg*
  - If e_1 **val** and e_2 **val**, then by canonical forms, $e_1 = \mathit{fn}\, x{:}\tau'.e_0$ and so $(\mathit{fn}\, x{:}\tau'.e_0)\ e_2 \mapsto [e_2/x]e_0$ by rule *step/app/beta*

Progress theorem (app)

If $\cdot \vdash e : \tau$, then e **val** or else $e \mapsto e'$ (for some e').

Case: $\dfrac{e_1 : \tau' \to \tau \qquad e_2 : \tau'}{e_1\ e_2 : \tau}$ *of/app*

$(e = e_1\ e_2)$

- By the induction hypothesis, e_1 **val** or else $e_1 \mapsto e_1'$, and additionally e_2 **val** or else $e_2 \mapsto e_2'$
- Case analysis to prove that $e_1\ e_2 \mapsto e'$:
  - If $e_1 \mapsto e_1'$, then $e_1\ e_2 \mapsto e_1'\ e_2$ by rule *step/app/fn*
  - If e_1 **val** and $e_2 \mapsto e_2'$, then $e_1\ e_2 \mapsto e_1\ e_2'$ by rule *step/app/arg*
  - If e_1 **val** and e_2 **val**, then by further case analysis on the derivation of $e_1 : \tau \to \tau'$, $e_1 = \mathit{fn}\, x{:}\tau'.e_0$ and so $(\mathit{fn}\, x{:}\tau'.e_0)\ e_2 \mapsto [e_2/x]e_0$ by rule *step/app/beta*

Progress theorem (successor)

If $\cdot \vdash e : \tau$, then e **val** or else $e \mapsto e'$ (for some e').

Case: $\dfrac{e_1 : num}{s(e_1) : num}$ *of/s*

$(e = s(e_1), \tau = num)$

- By the induction hypothesis, e_1 **val** or else $e_1 \mapsto e_1'$
- Case analysis to prove $s(e_1)$ **val** or else $s(e_1) \mapsto e'$:
  - If e_1 **val**, then $s(e_1)$ **val** by rule *value/s*
  - If $e_1 \mapsto e_1'$ for some e_1', then $s(e_1) \mapsto s(e_1')$ by rule *step/s*

Progress theorem (ifz)

If $\cdot \vdash e : \tau$, then e **val** or else $e \mapsto e'$ (for some e').

Case: $$\frac{e_{arg} : num \qquad e_0 : \tau \qquad x : num \vdash e_1 : \tau}{ifz(e_{arg}, e_0, x.e_1) : \tau}\ \textit{of/ifz}$$

$(e = ifz(e_{arg}, e_0, x.e_1))$

- By the induction hypothesis, e_{arg} **val** or else $e_{arg} \mapsto e'_{arg}$
- Case analysis to prove that $ifz(e_{arg}, e_0, x.e_1) \mapsto e'$:
  - If $e_{arg} \mapsto e'_{arg}$, then $ifz(e_{arg}, e_0, x.e_1) \mapsto ifz(e'_{arg}, e_0, x.e_1)$ by rule *step/ifz/arg*.
  - If e_{arg} **val**, then by canonical forms, either
    - $e_{arg} = z$, and $ifz(e_{arg}, e_0, x.e_1) \mapsto e_0$ by rule *step/ifz/z*.
    - $e_{arg} = s(e_{pred})$ where e_{pred} **val**, and $ifz(e_{arg}, e_0, x.e_1) \mapsto e_0$ by rule *step/ifz/s*.

Part VI

Bonus Slides: Canonical LF

Canonical LF

Formation judgements of LF:

$$\Gamma \vdash_\Sigma K \,\mathsf{kind}$$

$$\Gamma \vdash_\Sigma A \Rightarrow K$$

$$\Gamma \vdash_\Sigma M \Leftarrow A \qquad \Gamma \vdash_\Sigma R \Rightarrow A$$

$$\vdash_\Sigma \Gamma \; ok \qquad \vdash \Sigma \; ok$$

Canonical objects are analyzed, atomic objects are synthesized.

Canonical LF

Substitution judgements of LF:

$$[M/x]K = K'$$

$$[M/x]A = A'$$

$$[M/x]N = N' \qquad [M/x]R = M'$$

Canonical LF

Substitution judgements of LF:

$$[M/x]K = K'$$

$$[M/x]A = A'$$

$$[M/x]N = N' \qquad [M/x]R = M'$$

The critical case threatens termination:

$$[\lambda_{y:A} M/x](x\, N) = [N/y]M$$

Canonical LF

Substitution judgements of LF:

$$[M/x]K = K'$$

$$[M/x]A = A'$$

$$[M/x]N = N' \qquad [M/x]R = M'$$

The critical case threatens termination:

$$[\lambda_{y:A} M/x](x\, N) = [N/y]M$$

But the erased type (dependency-free simple type) of the substituting object gets smaller!

Atomic Objects

Variables and constants:

$$\frac{}{\Gamma \vdash_{\Sigma_1, c:A, \Sigma_2} c \Rightarrow A} \qquad \frac{}{\Gamma_1, x : A, \Gamma_2 \vdash_{\Sigma} x \Rightarrow A}$$

Atomic Objects

Variables and constants:

$$\frac{}{\Gamma \vdash_{\Sigma_1, c:A, \Sigma_2} c \Rightarrow A} \qquad \frac{}{\Gamma_1, x : A, \Gamma_2 \vdash_{\Sigma} x \Rightarrow A}$$

Function application:

$$\frac{\Gamma \vdash_{\Sigma} R \Rightarrow \Pi_{x:A_1} A_2 \quad \Gamma \vdash_{\Sigma} M \Leftarrow A_1 \quad [M/x]A_2 = A}{\Gamma \vdash_{\Sigma} R\, M \Rightarrow A}$$

Canonical Objects

Atomic objects of base type are canonical:

$$\frac{\Gamma \vdash_\Sigma R \Rightarrow A \quad A \neq \Pi_{x:A_1} A_2}{\Gamma \vdash_\Sigma R \Leftarrow A}$$

Canonical Objects

Atomic objects of base type are canonical:

$$\frac{\Gamma \vdash_\Sigma R \Rightarrow A \quad A \neq \Pi_{x:A_1} A_2}{\Gamma \vdash_\Sigma R \Leftarrow A}$$

Abstractions are canonical at higher type:

$$\frac{\Gamma, x : A_1 \vdash_\Sigma M \Leftarrow A_2}{\Gamma \vdash_\Sigma \lambda_{x:A_1} M_2 \Leftarrow \Pi_{x:A_1} A_2}$$

Type Families

Constants:

$$\frac{}{\Gamma \vdash_{\Sigma_1, a:K, \Sigma_2} a \Rightarrow K}$$

Type Families

Constants:

$$\frac{}{\Gamma \vdash_{\Sigma_1, a:K, \Sigma_2} a \Rightarrow K}$$

Family instantiation:

$$\frac{\Gamma \vdash_\Sigma A \Rightarrow \Pi_{x:A_1} K_2 \quad \Gamma \vdash_\Sigma M \Leftarrow A_1 \quad [M/x]K_2 = K}{\Gamma \vdash_\Sigma A\, M \Rightarrow K}$$

Type Families

Products of families:

$$\frac{\Gamma \vdash_\Sigma A_1 \Rightarrow \text{type} \quad \Gamma, x : A_1 \vdash_\Sigma A_2 \Rightarrow \text{type}}{\Gamma \vdash_\Sigma \Pi_{x:A_1} A_2 \Rightarrow \text{type}}$$

Kinds

The kind of types:

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{type}\,\mathsf{kind}}$$

Kinds

The kind of types:

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{type}\,\mathsf{kind}}$$

Product of a kind family:

$$\frac{\Gamma \vdash_\Sigma A_1 \Rightarrow \mathsf{type} \quad \Gamma, x : A_1 \vdash_\Sigma K_2\,\mathsf{kind}}{\Gamma \vdash_\Sigma \Pi_{x:A_1} K_2\,\mathsf{kind}}$$