Mechanizing Metatheory with LF and Twelf

Robert Harper
with Daniel R. Licata

Carnegie Mellon University

University of Oregon Summer School on Logic and Theorem Proving in Programming Languages
July, 2008
Part I

Overview
What We’ll Learn

Representation of languages and logics in LF.
What We’ll Learn

Representation of languages and logics in LF.

- Higher-Order Abstract Syntax (HOAS)
What We’ll Learn

Representation of languages and logics in LF.

- Higher-Order Abstract Syntax (HOAS)
- Judgements-as-Types Principle
What We’ll Learn

Representation of languages and logics in LF.
- Higher-Order Abstract Syntax (HOAS)
- Judgements-as-Types Principle

Mechanization of metatheory using Twelf.
What We’ll Learn

Representation of languages and logics in LF.
- Higher-Order Abstract Syntax (HOAS)
- Judgements-as-Types Principle

Mechanization of metatheory using Twelf.
- Relational Metathory.
What We’ll Learn

Representation of languages and logics in LF.
  • Higher-Order Abstract Syntax (HOAS)
  • Judgements-as-Types Principle

Mechanization of metatheory using Twelf.
  • Relational Metathory.
  • Checking Coverage and Totality.
How We’ll Learn It

Format:

• Lectures on theory [Harper].
• Laboratories using Twelf [Licata].

Readings:

• Practical Foundations for Programming Languages.
• Mechanizing Metatheory in a Logical Framework.
• Twelf Wiki: [http://twelf.org](http://twelf.org)
How We’ll Learn It

Format:

- Lectures on theory [Harper].
How We’ll Learn It

Format:

- **Lectures** on theory [Harper].
- **Laboratories** using Twelf [Licata].
How We’ll Learn It

Format:

- Lectures on theory [Harper].
- Laboratories using Twelf [Licata].

Readings:
How We’ll Learn It

Format:

- Lectures on theory [Harper].
- Laboratories using Twelf [Licata].

Readings:

- *Practical Foundations for Programming Languages.*
How We’ll Learn It

Format:

- **Lectures** on theory [Harper].
- **Laboratories** using Twelf [Licata].

Readings:

- *Practical Foundations for Programming Languages*.
- *Mechanizing Metatheory in a Logical Framework*. 

Twelf Wiki: [http://twelf.org](http://twelf.org)
How We’ll Learn It

Format:

• Lectures on theory [Harper].
• Laboratories using Twelf [Licata].

Readings:

• Practical Foundations for Programming Languages.
• Mechanizing Metatheory in a Logical Framework.
• Twelf Wiki: http://twelf.org
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.

- Hierarchical structure (algebraic terms).
What is LF?

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

- Hierarchical structure (algebraic terms).
- Binding and scope of identifiers.
What is LF?

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

- Hierarchical structure (algebraic terms).
- Binding and scope of identifiers.
- Context-sensitive formation rules.
What is LF?

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

- Hierarchical structure (algebraic terms).
- Binding and scope of identifiers.
- Context-sensitive formation rules.

A language is inductively presented by a collection of generators, whose types are specified by a signature.
The formation judgement $e \ exp$ states that $e$ is an arithmetic expression.

This judgement is inductively defined by these two rules:

$$\frac{n \ nat}{\overline{n} \ exp} \quad \frac{e_1 \ exp \ e_2 \ exp}{e_1 + e_2 \ exp}$$
Simple Arithmetic Expressions

The formation judgement $e \text{ exp}$ states that $e$ is an arithmetic expression.

This judgement is inductively defined by these two rules:

\[
\frac{n \text{ nat}}{n \text{ exp}} \quad \frac{e_1 \text{ exp} \quad e_2 \text{ exp}}{e_1 + e_2 \text{ exp}}
\]

The judgement $e \text{ exp}$ is the strongest (most restrictive) judgement closed under (obeying) these rules.
Abstract Syntax in LF

Each rule becomes a generator in the LF signature.
Abstract Syntax in LF

Each rule becomes a generator in the LF signature.

Define abstract syntax of expressions.

```
exp : type.
num : nat -> exp.
plus : exp -> exp -> exp.
```
Abstract Syntax in LF

Each rule becomes a generator in the LF signature.

Define natural numbers.

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

Define abstract syntax of expressions.

\[ \begin{align*}
  \text{exp} & : \text{type}. \\
  \text{num} & : \text{nat} \to \text{exp}. \\
  \text{plus} & : \text{exp} \to \text{exp} \to \text{exp}.
\end{align*} \]
Simple Arithmetic Expressions

Every arithmetic expression is uniquely represented by a closed LF term of LF type exp.

\[ \llbracket 2 + 3 \rrbracket = \text{plus} (\text{num} (s (s z))) (\text{num} (s (s (s z)))) \].

Moreover, every closed LF term of LF type exp represents a unique arithmetic expression.
Simple Arithmetic Expressions

Every arithmetic expression is uniquely represented by a closed LF term of LF type exp.

\[\langle 2 + 3 \rangle = \text{plus (num (s (s z))) (num (s (s (s z))))}\].

Moreover, every closed LF term of LF type exp represents a unique arithmetic expression.

These conditions express the adequacy of the representation of arithmetic expressions:

\[e \text{ exp} \iff \langle e \rangle : \text{expr} \].
The evaluation judgement $e \downarrow a$ states that the expression $e$ evaluates to the answer $a$.

It is defined by these rules:

\[ \begin{array}{c}
\overline{n} \downarrow \overline{n} \\
\overline{e_1} \downarrow \overline{n_1} \quad \overline{e_2} \downarrow \overline{n_2} \\
\overline{e_1 + e_2} \downarrow \overline{n}
\end{array} \]

\[ n = n_1 + n_2 \]

It is the strongest judgement closed under these rules.
The judgement is represented by a family of types:

\[
\text{eval} : \text{exp} \rightarrow \text{ans} \rightarrow \text{type}
\]
Define the type of answers:

\[
\begin{align*}
\text{ans} & : \text{type}. \\
\text{anat} & : \text{nat} \to \text{ans}.
\end{align*}
\]

The judgement is represented by a family of types:

\[
\begin{align*}
\text{eval} & : \text{exp} \to \text{ans} \to \text{type}.
\end{align*}
\]
Define the type of answers:

\[
\text{ans : type.}
\]
\[
\text{anat : nat \to ans.}
\]

The judgement is represented by a family of types:

\[
\text{eval : exp \to ans \to type.}
\]

The LF type \(\llbracket e \Downarrow a \rrbracket\) represents derivations of \(e \Downarrow a\).

\[
\nabla : e \Downarrow a \iff \llbracket \nabla \rrbracket : \text{eval} \llbracket e \rrbracket \llbracket a \rrbracket
\]
Rules as Generators

Each evaluation rule is represented by a generator:

eval/num
  : eval (num N) (anum N).

eval/plus
  : eval (plus E1 E2) (anum N)
    <- eval E1 (anum N1)
    <- eval E2 (anum N2)
    <- add N1 N2 N.
Each evaluation rule is represented by a generator:

```
// eval/num rule
: eval (num N) (anum N).

// eval/plus rule
: eval (plus E1 E2) (anum N)
  <- eval E1 (anum N1)
  <- eval E2 (anum N2)
  <- add N1 N2 N.
```

But what is meant by `add`?
Rules as Generators

Must define addition on natural numbers as well:

\[
\text{add} : \text{nat} \to \text{nat} \to \text{nat} \to \text{type}.
\]

\[
\text{add/z} : \text{add } z \ N \ N.
\]

\[
\text{add/s} : \text{add } (s \ M) \ N \ (s \ P) \leftarrow \text{add } M \ N \ P.
\]
Must define addition on natural numbers as well:

\[
\text{add} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{type}.
\]

\[
\text{add/z} : \text{add } z \text{ N N}.
\]

\[
\text{add/s} : \text{add } (s \text{ M) N (s P) } \leftarrow \text{add M N P}.
\]

**Adequacy:** \( \exists D : \text{add } \lceil m \rceil \lceil n \rceil \lceil p \rceil \text{ iff } m + n = p. \)
Fully Explicit Form

Eliminating abbreviations, and writing out parameters:

```
  eval/num
    : {N:nat} eval (num N) (anum N).

  eval/plus
    : {N:nat} {N1:nat} {N2:nat} {E1:exp} {E2:exp}
      add N1 N2 N ->
      eval E2 (anum N2) ->
      eval E1 (anum N1) ->
      eval (plus E1 E2) (anum N)
```

Twelf takes care of all of this; you never have to write declarations in fully explicit form.
Mechanized Metatheory

We can use Twelf to verify properties of representations.

1. **Modes**: functional dependencies in a type family (relation).
We can use Twelf to verify properties of representations.

1. **Modes**: functional dependencies in a type family (relation).
2. **Coverage**: all cases have been covered.
We can use Twelf to verify properties of representations.

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

We can use Twelf to verify properties of representations.

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

Twelf can prove $\forall\exists$-type properties of representations.

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

We can use Twelf to verify properties of representations.

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

Twelf can 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!
Let’s verify that \texttt{add} defines a \textit{total relation}.

\begin{verbatim}
  add : nat -> nat -> nat -> type.
  add/z : add z N N.
  add/s : add (s M) N (s P) <- add M N P.
\end{verbatim}
Let’s verify that add defines a total relation.

\[
\begin{align*}
\text{add} & : \text{nat} \to \text{nat} \to \text{nat} \to \text{type}.
\text{add/}z & : \text{add } z \ N \ N.
\text{add/}s & : \text{add } (s \ M) \ N \ (s \ P) \leftarrow \text{add } M \ N \ P.
\end{align*}
\]

That is, \( M \) and \( N \) determine \( P \) in \( \text{add } M \ N \ P \):

\[
\begin{align*}
\text{add} & : \text{nat} \to \text{nat} \to \text{nat} \to \text{type}.
%\text{mode add +M +N -P}.
\text{add/}z & : \text{add } z \ N \ N.
\text{add/}s & : \text{add } (s \ M) \ N \ (s \ P) \leftarrow \text{add } M \ N \ P.
\end{align*}
\]
Coverage and Termination

To show that $\text{add} \ M \ N \ P$ determines $P$, given $M$ and $N$, we check

1. **Coverage.** There is a clause for every $M$.
2. **Termination.** There are no circular dependencies.

Twelf declarations:
```
%worlds () (add).
%total M (add M).
```
Specifies that \texttt{add} is to be proved total on closed terms of type \texttt{nat} by structural induction on the first argument.
Coverage and Termination

To show that \( \text{add} \, M \, N \, P \) determines \( P \), given \( M \) and \( N \), we check

1. **Coverage.** There is a clause for every \( M \).
2. **Termination.** There are no circular dependencies.

Twelf declarations:

\[
\begin{align*}
&\text{%worlds () (add \_ \_ \_).} \\
&\text{%total M (add M \_ \_).}
\end{align*}
\]
Coverage and Termination

To show that \( \text{add}\ M\ N\ P \) determines \( P \), given \( M \) and \( N \), we check

1. **Coverage.** There is a clause for every \( M \).

2. **Termination.** There are no circular dependencies.

Twelf declarations:

\[
\begin{align*}
\%&\text{worlds () (add } \_ \_ \_\). \\
\%&\text{total M (add M } \_ \_\). \\
\end{align*}
\]

Specifies that \text{add} is to be proved total on closed terms of type \text{nat} by structural induction on the first argument.
Twelf has verified that
\[ \forall M, N : \text{nat} \exists P : \text{nat} \text{ add } M \cdot N \cdot P. \]

This statement may be usefully re-phrased as
\[ \forall M, N : \text{nat} \exists P : \text{nat} \exists D : \text{add } M \cdot N \cdot P \cdot \top. \]
Twelf has verified that

$$\forall M, N : \text{nat} \ \exists P : \text{nat} \ \text{add} \ M \ N \ P.$$  

This statement may be usefully re-phrased as

$$\forall M, N : \text{nat} \ \exists P : \text{nat} \ \exists D : \text{add} \ M \ N \ P \ \top.$$  

Important: we may reason directly about derivations!
We may just as easily prove that evaluation terminates!

\[
\%\texttt{worlds} \ () \ (\text{eval} \ _ \ _).
\%\texttt{total} \ E \ (\text{eval} \ E \ _).
\]

That is, Twelf has proved

\[
\forall E : \text{exp} \ \exists A : \text{ans} \ \exists D : \text{eval} \ E \ A \top
\]

This states termination of evaluation, by the adequacy of the representation.
Adding Bindings

Now enrich expressions with a binding construct:

\[
\text{let } x \text{ be } e_1 \text{ in } e_2
\]

with the meaning that \( x \) stands for \( e_1 \) within \( e_2 \).
Adding Bindings

Now enrich expressions with a binding construct:

\[
\text{let } x \text{ be } e_1 \text{ in } e_2
\]

with the meaning that \( x \) stands for \( e_1 \) within \( e_2 \).

The variable \( x \) is bound within \( e_2 \). It serves as a pronoun referring to the binding site.
Adding Bindings

Now enrich expressions with a binding construct:

\[
\text{let } x \text{ be } e_1 \text{ in } e_2
\]

with the meaning that \( x \) stands for \( e_1 \) within \( e_2 \).

The variable \( x \) is bound within \( e_2 \). It serves as a pronoun referring to the binding site.

- May be renamed, preserving pronoun structure:

\[
\text{let } x \text{ be } 3 \text{ in } x + x \quad \text{is} \quad \text{let } y \text{ be } 3 \text{ in } y + y.
\]
Adding Bindings

Now enrich expressions with a binding construct:

\begin{center}
\texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2
\end{center}

with the meaning that \( x \) stands for \( e_1 \) within \( e_2 \).

The variable \( x \) is \textbf{bound} within \( e_2 \). It serves as a \textbf{pronoun} referring to the binding site.

- May be \textbf{renamed}, preserving pronoun structure:
  \begin{center}
  let \( x \) be \( 3 \) in \( x + x \) is let \( y \) be \( 3 \) in \( y + y \).
  \end{center}

- May be \textbf{substituted} by an expression, preserving pronoun structure:
  \begin{center}
  \( [3/x](x + x) \) is \( 3 + 3 \).
  \end{center}
Adding Bindings

Enrich expressions with a let-binding:

\[
\frac{e_1 \quad x \quad \vdash e_2}{\text{let } x \text{ be } e_1 \text{ in } e_2}
\]
Adding Bindings

Enrich expressions with a let-binding:

\[
\begin{align*}
\frac{e_1 \text{ exp} \quad x \text{ exp} \vdash e_2 \text{ exp}}{
\text{let } x \text{ be } e_1 \text{ in } e_2 \text{ exp}}
\end{align*}
\]

The hypothetical judgement expresses binding structure:
Enrich expressions with a let-binding:

\[
\begin{align*}
\text{let } x \text{ be } e_1 \text{ in } e_2
\end{align*}
\]

The hypothetical judgement expresses binding structure:
Adding Bindings

Enrich expressions with a let-binding:

\[
\frac{e_1 \text{ exp} \quad x \text{ exp} \vdash e_2 \text{ exp}}{\text{let } x \text{ be } e_1 \text{ in } e_2 \text{ exp}}
\]

The hypothetical judgement expresses binding structure:

- The variable \( x \) may occur within \( e_2 \).
Adding Bindings

Enrich expressions with a let-binding:

\[
\frac{e_1 \text{ exp} \quad x \text{ exp} \vdash e_2 \text{ exp}}{\text{let } x \text{ be } e_1 \text{ in } e_2 \text{ exp}}
\]

The hypothetical judgement expresses binding structure:

- The variable \( x \) may occur within \( e_2 \).
- The name of the variable does not matter, only its referent.
Adding Bindings

Enrich expressions with a let-binding:

\[
\begin{array}{c}
e_1 \text{ exp} \\
\times \text{ exp} \vdash e_2 \text{ exp} \\
\hline
\text{let } x \text{ be } e_1 \text{ in } e_2 \text{ exp}
\end{array}
\]

The hypothetical judgement expresses binding structure:

- The variable \( x \) may occur within \( e_2 \).
- The name of the variable does not matter, only its referent.
- Substitution is valid: \([e/x]e_2 \text{ exp}\) whenever \( e \text{ exp}\).
The `let` construct is given by the declaration

\[
\text{let} : \text{exp} \rightarrow (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}.
\]

Uses higher-order functions to express binding and scope!
Higher-Order Abstract Syntax

The **let** construct is given by the declaration

\[
\text{let : exp} \rightarrow (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}.
\]

Uses higher-order functions to express binding and scope!

Representation:

\[
\begin{align*}
\ulcorner \text{let } x \text{ be } e_1 \text{ in } e_2 \urcorner &= \text{let } \ulcorner e_1 \urcorner ([x : \text{exp}] \ulcorner e_2 \urcorner).
\end{align*}
\]

where the \(\lambda\)-abstraction, \([x : \text{exp}]\), expresses the binding and scope of \(x\) in \(e_2\).
Higher-Order Abstract Syntax

The `let` construct is given by the declaration

\[
\text{let} : \text{exp} \rightarrow (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}.
\]

Uses higher-order functions to express binding and scope!

Representation:

\[
\boxed{\text{let } x \text{ be } e_1 \text{ in } e_2} \equiv \text{let } \boxed{e_1} (\boxed{[x:\text{exp}] \boxed{e_2}}).
\]

where the \( \lambda \)-abstraction, \([x:\text{exp}]\), expresses the binding and scope of \( x \) in \( e_2 \).
Consider the rule for evaluation of a `let`:

\[
\frac{e_1 \Downarrow \overline{n_1} \quad [\overline{n_1}/x]e_2 \Downarrow a}{\text{let x be } e_1 \text{ in } e_2 \Downarrow a}
\]
Consider the rule for evaluation of a let:

\[
\begin{array}{c}
e_1 \Downarrow n_1 \\
[\overline{n_1}/x]e_2 \Downarrow a \\
\text{let } x \text{ be } e_1 \text{ in } e_2 \Downarrow a
\end{array}
\]

Formulated in LF:

\[
\text{eval/let}
\]

: eval (let E1 ([x] E2 x)) A

<- eval E1 (anum N1)

<- eval (E2 (num N1)) A.
Consider the rule for evaluation of a let:

\[
\begin{align*}
    e_1 & \Downarrow n_1 \\
    \frac{\left[ n_1/x \right] e_2 \Downarrow a}{\text{let } x \text{ be } e_1 \text{ in } e_2 \Downarrow a}
\end{align*}
\]

Formulated in LF:

\[
\begin{align*}
    \text{eval/let} & \\
    : & \text{eval (let E1 ([x] E2 x)) A} \\
    & \leftarrow \text{eval E1 (anum N1)} \\
    & \leftarrow \text{eval (E2 (num N1)) A}.
\end{align*}
\]

Substitution is provided \textit{for free} by LF!
Enforcing Stronger Invariants

We can track that variables are bound to values.

\[
\begin{align*}
\text{val} & : \text{type}. \\
\text{num} & : \text{nat} \to \text{val}. \\
\text{exp} & : \text{type}. \\
\text{ret} & : \text{val} \to \text{exp}. \\
\text{plus} & : \text{exp} \to \text{exp} \to \text{exp}. \\
\text{let} & : \text{exp} \to (\text{val} \to \text{exp}) \to \text{exp}.
\end{align*}
\]

As a rule it is good practice to use types to enforce invariants on a representation.
Higher-Order Rules

We may use hypothetical judgments to represent bindings:

\[
\begin{align*}
    e_1 & \downarrow a_1 & \text{ret } x & \downarrow a_1 \vdash e_2 \downarrow a_2 \\
    \text{let } x \text{ be } e_1 \text{ in } e_2 & \downarrow a_2
\end{align*}
\]
Higher-Order Rules

We may use hypothetical judgements to represent bindings:

\[
\begin{align*}
  e_1 \downarrow a_1 & \quad \text{ret} \ x \downarrow a_1 \vdash e_2 \downarrow a_2 \\
  \text{let} \ x \ \text{be} \ e_1 \ \text{in} \ e_2 & \downarrow a_2
\end{align*}
\]

The evaluation hypothesis governs the variable \( x \) in \( e_2 \).

\[
\begin{align*}
  \text{ret} \ x \ \downarrow \ 3 \ \vdash (\text{ret}, x) + (\text{ret} \ 4) \ \downarrow \ 7
\end{align*}
\]
Higher-order rules are represented using higher-order types:

\[
\text{eval/let} \\
: \ \text{eval (let E1 ([x] E2 x)) A} \\
<- \ \text{eval E1 A1} \\
<- (\{x:val\} \ \text{eval (ret x) A1} \rightarrow \ \text{eval (E2 x) A2}).
\]
Higher-order rules are represented using higher-order types:

\[
\text{eval/let} \\
\quad : \text{eval} \ (\text{let} \ E1 \ ([x] \ E2 \ x)) \ A \\
\quad \leftarrow \text{eval} \ E1 \ A1 \\
\quad \leftarrow (\{x: \text{val}\} \ \text{eval} \ (\text{ret} \ x) \ A1 \rightarrow \text{eval} \ (E2 \ x) \ A2).
\]

The general hypothetical judgement expresses that body is evaluated relative to
Higher-order rules are represented using higher-order types:

\[
\text{eval/let} \\
: \text{eval (let } E1 ([x] E2 x)) A \\
\leftarrow \text{eval } E1 A1 \\
\leftarrow (\{x: \text{val}\} \text{eval (ret } x) A1 \rightarrow \text{eval (E2 } x) A2). \\
\]

The general hypothetical judgement expresses that body is evaluated relative to
- A fresh variable, \(x\);
Higher-order rules are represented using higher-order types:

\[
\text{eval/let} \\
\quad : \text{eval} \left( \text{let } E1 \left( \left[x\right] E2 \ x \right) \right) A \\
\quad \quad <- \text{eval} \ E1 \ A1 \\
\quad \quad <- \left\{ \left[x:\text{val} \right] \text{eval} \left( \text{ret } x \right) A1 \rightarrow \text{eval} \left( E2 \ x \right) A2 \right\}.
\]

The general hypothetical judgement expresses that body is evaluated relative to

- A fresh variable, \( x \);
- A new axiom, stating that \( x \) evaluates to value of \( E1 \).
Higher-Order Rules

The key to understanding higher-order rules is to understand the LF type theory.
Higher-Order Rules

The key to understanding higher-order rules is to understand the LF type theory.

• The type $A \rightarrow B$ consists of $B$’s, possibly using a fresh axioms for $A$. 
Higher-Order Rules

The key to understanding higher-order rules is to understand the LF type theory.

- The type $A \rightarrow B$ consists of $B$’s, possibly using a fresh axioms for $A$.
- The type $\Pi_{x:A}B$ consists of $B$’s with free variables $x$ of type $A$ in them.
The key to understanding higher-order rules is to understand the LF type theory.

- The type $A \rightarrow B$ consists of $B$’s, possibly using a *fresh* axioms for $A$.
- The type $\Pi_{x:A} B$ consists of $B$’s with *free variables* $x$ of type $A$ in them.

LF types represent *derivabilities*, not *admissibilities*!
Higher-Order Rules

The key to understanding higher-order rules is to understand the LF type theory.

- The type $A \rightarrow B$ consists of $B$’s, possibly using a fresh axioms for $A$.
- The type $\Pi_{x:A}B$ consists of $B$’s with free variables $x$ of type $A$ in them.

LF types represent derivabilities, not admissibilities!

- $J_1 \vdash J_2$ represented by $\left[ J_1 \right] \rightarrow \left[ J_2 \right]$.
Higher-Order Rules

The key to understanding higher-order rules is to understand the LF type theory.

- The type $A \rightarrow B$ consists of $B$’s, possibly using a fresh axioms for $A$.
- The type $\Pi_{x:A}B$ consists of $B$’s with free variables $x$ of type $A$ in them.

LF types represent derivabilities, not admissibilities!

- $J_1 \vdash J_2$ represented by $\llbracket J_1 \rrbracket \rightarrow \llbracket J_2 \rrbracket$.
- $|_{x:A} J$ represented by $\Pi_{x:A} \llbracket J \rrbracket$. 
Adequacy and Worlds

Adequacy of substitutive evaluation is relative to a closed world with no free derivation variables.

\[ \nabla : e \downarrow a \quad \text{iff} \quad \llbracket \nabla \rrbracket : \text{eval} \llbracket e \rrbracket \llbracket a \rrbracket. \]
Adequacy and Worlds

Adequacy of substitutive evaluation is relative to a closed world with no free derivation variables.

∇ : e ↓ a  iff ⌜∇⌝ : eval⌜e⌝ ⌜a⌝.

This is expressed by the %worlds declaration:

%worlds () (eval _ _).
%total E (eval E _).
Adequacy and Worlds

Adequacy of substitutive evaluation is relative to a closed world with no free derivation variables.

\[ \nabla : e \Downarrow a \quad \text{iff} \quad \llbracket \nabla \rrbracket : \text{eval} \llbracket e \rrbracket a. \]

This is expressed by the `%worlds` declaration:

\`
%worlds () (eval _ _).
%total E (eval E _).
\`
Adequacy and Worlds

Higher-order evaluation introduces parameters and hypotheses during evaluation.
Adequacy and Worlds

Higher-order evaluation introduces parameters and hypotheses during evaluation.

Consider **worlds** (contexts) consisting of **blocks** of the form

\[ x : \text{val}, \_ : \text{eval}(\text{ret } x) a. \]

Adequacy is now stated relative to hypotheses represented by worlds:

\[ \nabla : \text{ret } x_1 \downarrow a_1, \ldots \downarrow a \]

iff

\[ x_1 : \text{val}, \_ : \text{eval}(\text{ret } x_1) a_1, \ldots \vdash \nabla \downarrow : \text{eval} \nabla \downarrow e \downarrow a \downarrow \]
Adequacy and Worlds

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

```twelf
%block eval_block
    : some {A:ans} block {x:val} {_:eval (ret x) A}.
%worlds (eval_block) (eval _ _).
%total E (eval E _).
```

These declarations check termination of the higher-order formulation of evaluation.
Suppose we wished to extend the expression language with strings.
Typed Expressions

Suppose we wished to extend the expression language with strings. We wish to impose context-sensitive constraints on formation of expressions.

- Cannot add a string and a number.
- Cannot concatenate two numbers.
Typed Expressions

Suppose we wished to extend the expression language with strings. We wish to impose context-sensitive constraints on formation of expressions.

- Cannot add a string and a number.
- Cannot concatenate two numbers.

This is achieved using a dependent type, which is a family of types indexed by some other type.
Typed Expressions

Introduce an LF type of expression types.

\[
\begin{align*}
\text{tp} &: \text{type}. \\
\text{number} &: \text{tp}. \\
\text{string} &: \text{tp}.
\end{align*}
\]
Typed Expressions

Introduce an LF type of expression types.

\[
\begin{align*}
\text{tp} & : \text{type}. \\
\text{number} & : \text{tp}. \\
\text{string} & : \text{tp}.
\end{align*}
\]

Introduce an LF family of expressions of a type.

\[
\begin{align*}
\text{exp} & : \text{tp} \rightarrow \text{type}. \\
\text{num} & : \text{nat} \rightarrow \text{number exp}. \\
\text{lit} & : \text{str} \rightarrow \text{string exp}. \\
\text{plus} & : \text{number exp} \rightarrow \text{number exp} \rightarrow \text{number exp}. \\
\text{append} & : \text{string exp} \rightarrow \text{string exp} \rightarrow \text{string exp}.
\end{align*}
\]
Typed Expressions

Introduce an LF type of expression types.

\[
\begin{align*}
\text{tp} & : \text{type}. \\
\text{number} & : \text{tp}. \\
\text{string} & : \text{tp}.
\end{align*}
\]

Introduce an LF family of expressions of a type.

\[
\begin{align*}
\text{exp} & : \text{tp} \to \text{type}. \\
\text{num} & : \text{nat} \to \text{number exp}. \\
\text{lit} & : \text{str} \to \text{string exp}. \\
\text{plus} & : \text{number exp} \to \text{number exp} \to \text{number exp}. \\
\text{append} & : \text{string exp} \to \text{string exp} \to \text{string exp}.
\end{align*}
\]

(Must also define type \text{str} of strings.)
Typed Expressions

Evaluation is defined exactly as before, but with types.

\[
\begin{align*}
\text{ans} & : \text{tp} \rightarrow \text{type}. \\
\text{anum} & : \text{nat} \rightarrow \text{ans}. \\
\text{astr} & : \text{str} \rightarrow \text{ans}. \\
\text{eval} & : T \ \text{exp} \rightarrow T \ \text{ans} \rightarrow \text{type}.
\end{align*}
\]
Typed Expressions

Evaluation is defined exactly as before, but with types.

\[
\begin{aligned}
\text{ans} & : \text{tp} \rightarrow \text{type}. \\
\text{anum} & : \text{nat} \rightarrow \text{ans}. \\
\text{astr} & : \text{str} \rightarrow \text{ans}.
\end{aligned}
\]

\[
\text{eval} : \text{T exp} \rightarrow \text{T ans} \rightarrow \text{type}.
\]

The LF type of \text{eval} ensures preservation: the type of answer is of the same type as the expression being evaluated!
Typed Expressions

Selected evaluation rules in LF:

- **eval/num**: `eval (num N) (anum N)`.  
- **eval/lit**: `eval (lit S) (astr S)`.  
- **eval/plus**
  - `eval (plus E1 E2) (anum N)`
  - `<- eval E1 (anum N1)`
  - `<- eval E2 (anum N2)`
  - `<- add N1 N2 N.`

No significant difference compared to untyped case.
Selected **evaluation rules** in LF:

- `eval/num : eval (num N) (anum N).`
- `eval/lit : eval (lit S) (astr S).`
- `eval/plus`
  - `: eval (plus E1 E2) (anum N)`
  - `<- eval E1 (anum N1)`
  - `<- eval E2 (anum N2)`
  - `<- add N1 N2 N.`

No significant difference compared to untyped case.
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.

- Signatures to define languages and logics.
Recap and Prospectus

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

- Signatures to define languages and logics.
- Mode specifications and checking.
Recap and Prospectus

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

- Signatures to define languages and logics.
- Mode specifications and checking.
- Coverage checking.
Recap and Prospectus

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

- Signatures to define languages and logics.
- Mode specifications and checking.
- Coverage checking.
- Termination checking.
Recap and Prospectus

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

- Signatures to define languages and logics.
- Mode specifications and checking.
- Coverage checking.
- Termination checking.

Next we will cover the LF Type Theory in more detail.
Recap and Prospectus

You’ve now seen all of the **basic** features of LF and Twelf.
- Signatures to define languages and logics.
- Mode specifications and checking.
- Coverage checking.
- Termination checking.

Next we will cover the **LF Type Theory** in more detail.

Then we will develop a larger piece of **metatheory**, the type safety of MinML.
Part II

Representation
A formal system is represented fully, faithfully, and compositionally by LF canonical forms of specified type in specified worlds.
A formal system is represented fully, faithfully, and compositionally by LF canonical forms of specified type in specified worlds.

- **Full**: every syntactic object \( o \) of class \( C \) has a unique representation \( \llbracket o \rrbracket \) of type \( \llbracket C \rrbracket \).
A formal system is represented **fully, faithfully, and compositionally** by **LF canonical forms** of specified **type** in specified **worlds**.

- **Full**: every syntactic object \( o \) of class \( C \) has a unique representation \( \llbracket o \rrbracket \) of type \( \llbracket C \rrbracket \).
- **Faithful**: every canonical form of type \( \llbracket C \rrbracket \) represents a unique syntactic object of class \( C \).
A formal system is represented fully, faithfull, and compositionally by LF canonical forms of specified type in specified worlds.

- **Full**: every syntactic object \( o \) of class \( C \) has a unique representation \( \llbracket o \rrbracket \) of type \( \llbracket C \rrbracket \).
- **Faithful**: every canonical form of type \( \llbracket C \rrbracket \) represents a unique syntactic object of class \( C \).
- **Compositional**: representation commutes with substitution,
\[
\llbracket [o_2/x]o_1 \rrbracket = [\llbracket o_2 \rrbracket/x]\llbracket o_1 \rrbracket.
\]
Theory of Representation

A formal system is represented fully, faithfully, and compositionally by LF canonical forms of specified type in specified worlds.

- **Full**: every syntactic object $o$ of class $C$ has a unique representation $\llbracket o \rrbracket$ of type $\llbracket C \rrbracket$.
- **Faithful**: every canonical form of type $\llbracket C \rrbracket$ represents a unique syntactic object of class $C$.
- **Compositional**: representation commutes with substitution, $\llbracket [o_2/x]o_1 \rrbracket = [\llbracket o_2 \rrbracket/x]\llbracket o_1 \rrbracket$.

Let us now make these ideas precise.
The LF Type Theory

LF is a dependently typed $\lambda$-calculus with two levels:
The LF Type Theory

LF is a dependently typed $\lambda$-calculus with two levels:
- Families, $A$, classified by Kinds, $K$.

- Objects, $M$, classified by Types, $A$. 

The syntax is classified into levels: 
- Kind $K ::= \text{type} | \Pi x : A K$
- Family $A ::= a | A M | \Pi x : A B$

Canonical Object $M ::= R | \lambda x : A M$

Atomic Object $R ::= x | c | R M$
The LF Type Theory

LF is a dependently typed $\lambda$-calculus with two levels:

- Families, $A$, classified by Kinds, $K$.
- Objects, $M$, classified by Types, $A$. 
The LF Type Theory

LF is a dependently typed $\lambda$-calculus with two levels:

- Families, $A$, classified by Kinds, $K$.
- Objects, $M$, classified by Types, $A$.

The syntax is classified into levels:

- **Kind**
  \[
  K ::= \text{type} \mid \Pi x:A K
  \]

- **Family**
  \[
  A ::= a \mid AM \mid \Pi x:A B
  \]

- **Canonical Object**
  \[
  M ::= R \mid \lambda x:A M
  \]

- **Atomic Object**
  \[
  R ::= x \mid c \mid RM
  \]
Intuitively, canonical objects are long $\beta\eta$-normal forms.
The LF Type Theory

Intuitively, canonical objects are long $\beta\eta$-normal forms.

- No $\beta$-redices: $\lambda_{x:A} M N$.
The LF Type Theory

Intuitively, canonical objects are long $\beta\eta$-normal forms.

- No $\beta$-redices: $\lambda_{x:A} M N$.
- Fully $\eta$-expanded: $\lambda_{x:A} y \, x$, not $y$, if $y : \Pi_{x:A} B$.
Intuitively, canonical objects are long $\beta\eta$-normal forms.

- No $\beta$-redices: $\lambda_{x:A} M N$.
- Fully $\eta$-expanded: $\lambda_{x:A} y \, x$, not $y$, if $y : \Pi_{x:A} B$.

Formally, these classes are **inductively defined** without reduction or expansion.
Intuitively, canonical objects are long $\beta\eta$-normal forms.

- No $\beta$-redices: $\lambda_{x:A} M \; N$.
- Fully $\eta$-expanded: $\lambda_{x:A} y \; x$, not $y$, if $y : \Pi_{x:A} B$.

Formally, these classes are **inductively defined** without reduction or expansion.

- Predicativity (clean living) makes this possible.
The LF Type Theory

Intuitively, canonical objects are long $\beta\eta$-normal forms.

- No $\beta$-redices: $\lambda_{x:A} M N$.
- Fully $\eta$-expanded: $\lambda_{x:A} y \ x$, not $y$, if $y : \Pi_{x:A} B$.

Formally, these classes are inductively defined without reduction or expansion.

- Predicativity (clean living) makes this possible.
- Never have to worry about non-canonical objects interfering with representation.
Intuitively, canonical objects are long $\beta\eta$-normal forms.

- No $\beta$-redices: $\lambda_{x:A} M N$.
- Fully $\eta$-expanded: $\lambda_{x:A} y \ x$, not $y$, if $y : \Pi_{x:A} B$.

Formally, these classes are inductively defined without reduction or expansion.

- Predicativity (clean living) makes this possible.
- Never have to worry about non-canonical objects interfering with representation.

But substitution must be defined to preserve canonical and atomic forms!
The LF Type Theory

An LF context, $\Gamma$, is a sequence of variable declarations:

$$x_1 : A_1, \ldots, x_n : A_n$$

wherein each $A_i$ may involve the preceding variables.
The LF Type Theory

An LF context, \( \Gamma \), is a sequence of variable declarations:

\[
x_1 : A_1, \ldots, x_n : A_n
\]

wherein each \( A_i \) may involve the preceding variables.

An LF signature, \( \Sigma \), is a sequence of constant declarations:

\[
\left\{ \begin{array}{l} a_1 : K_1 \\ c_1 : A_1 \\ \vdots \\ a_m : K_m \\ c_m : A_m \end{array} \right. 
\]

where each \( A_i \) or \( K_i \) may involve the preceding constants.
The LF Type Theory

**Formation judgements of LF:**

\[ \Gamma \vdash_{\Sigma} K \text{ kind} \]

\[ \Gamma \vdash_{\Sigma} A \Rightarrow K \]

\[ \Gamma \vdash_{\Sigma} M \Leftarrow A \quad \Gamma \vdash_{\Sigma} R \Rightarrow A \]

\[ \vdash_{\Sigma} \Gamma \ ok \]

\[ \vdash \Sigma \ ok \]

Canonical objects are **analyzed**, atomic objects are **synthesized**.
Substitution judgements of LF:

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

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

\[ [M/x]N = N' \]

\[ [M/x]R = M' \]
Substitution judgements of LF:

\[\frac{}{[M/x]K = K'}\]
\[\frac{}{[M/x]A = A'}\]
\[\frac{}{[M/x]N = N'}\]
\[\frac{}{[M/x]R = M'}\]

The critical case threatens termination:

\[\frac{}{[\lambda_{y:A} M/x](x N) = [N/y]M}\]
The LF Type Theory

Substitution judgements of LF:

\[ [M/x]K = K' \]
\[ [M/x]A = A' \]
\[ [M/x]N = N' \]
\[ [M/x]R = M' \]

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:

\[ \Gamma \vdash \Sigma_{1,c : A, \Sigma_2} c \Rightarrow A \]
\[ \Gamma_1, x : A, \Gamma_2 \vdash \Sigma x \Rightarrow A \]
Variables and constants:

\[ \Gamma \vdash_{\Sigma, 1, c : A, \Sigma_2} c \Rightarrow A \quad \Gamma_1, x : A, \Gamma_2 \vdash_{\Sigma} x \Rightarrow A \]

Function application:

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

\[ \Gamma \vdash_{\Sigma} R M \Rightarrow A \]
Canonical Objects

Atomic objects of base type are canonical:

\[
\Gamma \vdash \Sigma \quad R \Rightarrow A \quad A \not= \Pi_{x:A_1} A_2
\]

\[
\Gamma \vdash \Sigma \quad R \leftarrow A
\]
Canonical Objects

Atomic objects of base type are canonical:

\[
\frac{\Gamma \vdash \Sigma \ R \Rightarrow A \quad A \neq \Pi_{x:A_1} A_2}{\Gamma \vdash \Sigma \ R \leftrightarrow A}
\]

Abstractions are canonical at higher type:

\[
\frac{\Gamma, x : A_1 \vdash \Sigma M \leftrightarrow A_2}{\Gamma \vdash \Sigma \lambda_{x:A_1} M_2 \leftrightarrow \Pi_{x:A_1} A_2}
\]
Type Families

Constants:

\[ \Gamma \vdash \Sigma_{1,a:K}, \Sigma_{2} a \Rightarrow K \]
Type Families

Constants:

\[ \Gamma \vdash_{\Sigma_1, a: K, \Sigma_2} a \Rightarrow K \]

Family instantiation:

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

\[ \Gamma \vdash_{\Sigma} A M \Rightarrow K \]
Type Families

Products of families:

\[
\frac{\Gamma \vdash \Sigma A_1 \Rightarrow \text{type} \quad \Gamma, x : A_1 \vdash \Sigma A_2 \Rightarrow \text{type}}{\Gamma \vdash \Pi x : A_1 A_2 \Rightarrow \text{type}}
\]
The kind of types:

Γ ⊢ Σ type kind
The kind of types:

\[ \Gamma \vdash_{\Sigma} \text{type kind} \]

Product of a kind family:

\[ \Gamma \vdash_{\Sigma} A_1 \Rightarrow \text{type} \quad \Gamma, x : A_1 \vdash_{\Sigma} K_2 \text{ kind} \]

\[ \Gamma \vdash_{\Sigma} \Pi_{x : A_1} K_2 \text{ kind} \]
Central principle: capture entailments.
Central principle: **capture entailments**.

- **Syntactic**: variables and substitution (general judgement).
Representation Methodology

Central principle: capture entailments.

- **Syntactic**: variables and substitution (general judgement).
- **Deductive**: derivability consequence relation (hypothetical judgement).
Central principle: capture entailments.

- **Syntactic**: variables and substitution (general judgement).
- **Deductive**: derivability consequence relation (hypothetical judgement).

A lesson of LF is that there is no real distinction between the syntactic and the deductive.
Central principle: capture entailments.

- **Syntactic**: variables and substitution (general judgement).
- **Deductive**: derivability consequence relation (hypothetical judgement).

A lesson of LF is that there is no real distinction between the syntactic and the deductive.

**Advice**: represent as wide a class of entailments as possible, to maximize utility and generality.
Representation Methodology

Syntactic classes for arithmetic expressions:

- \texttt{v val \textit{values}}
Representation Methodology

Syntactic classes for arithmetic expressions:

- \( \nu \) val     values
- \( \varepsilon \) exp     expressions
Syntactic classes for arithmetic expressions:

- \( v \text{ val} \): values
- \( e \text{ exp} \): expressions
- \( a \text{ ans} \): answers

Entailments for arithmetic expressions:

- \( x \text{ val} \vdash v \text{ val} \): values with value variables
- \( x \text{ val} \vdash e \text{ exp} \): expressions with value variables
- \( \vdash a \text{ ans} \): closed answers
- \( \vdash e \Downarrow a \): closed evaluations
Syntactic classes for arithmetic expressions:

- $v$ val values
- $e$ exp expressions
- $a$ ans answers
- $e \Downarrow a$ derivations of evaluations
Representation Methodology

Syntactic classes for arithmetic expressions:

- $v \text{val}$ values
- $e \text{exp}$ expressions
- $a \text{ans}$ answers
- $e \Downarrow a$ derivations of evaluations

Entailments for arithmetic expressions:

- $x \text{val} \vdash v \text{val}$ values with value variables
Representation Methodology

Syntactic classes for arithmetic expressions:

- \( v \text{ val} \) values
- \( e \text{ exp} \) expressions
- \( a \text{ ans} \) answers
- \( e \Downarrow a \) derivations of evaluations

Entailments for arithmetic expressions:

- \( x \text{ val} \vdash v \text{ val} \) values with value variables
- \( x \text{ val} \vdash e \text{ exp} \) expressions with value variables
Representation Methodology

**Syntactic classes** for arithmetic expressions:

- \( \nu \text{ val} \) values
- \( \epsilon \text{ exp} \) expressions
- \( \alpha \text{ ans} \) answers
- \( \epsilon \Downarrow \alpha \) derivations of evaluations

**Entailments** for arithmetic expressions:

- \( \times \text{ val} \vdash \nu \text{ val} \) values with value variables
- \( \times \text{ val} \vdash \epsilon \text{ exp} \) expressions with value variables
- \( \vdash \alpha \text{ ans} \) closed answers
Representation Methodology

**Syntactic classes** for arithmetic expressions:

- $v \ val$  values
- $e \ exp$  expressions
- $a \ ans$  answers
- $e \downarrow a$  derivations of evaluations

**Entailments** for arithmetic expressions:

- $x \ val \vdash v \ val$  values with value variables
- $x \ val \vdash e \ exp$  expressions with value variables
- $\vdash a \ ans$  closed answers
- $\vdash e \downarrow a$  closed evaluations
Representation Methodology

Embed object-language entailments as LF entailments:

\[ x_1 \text{ val}, \ldots, x_k \text{ val} \vdash v \text{ val} \]

\[ \iff \]

\[ x_1 : \text{val}, \ldots, x_k : \text{val} \vdash_{\Sigma} \neg v \Rightarrow \text{val} \]
**Representation Methodology**

Embed object-language entailments as LF entailments:

\[
x_1 \text{ val}, \ldots, x_k \text{ val} \vdash v \text{ val}
\]
\[
\iff
\]
\[
x_1 : \text{val}, \ldots, x_k : \text{val} \vdash_{\Sigma} \lnot v \Rightarrow \text{val}
\]

\[
x_1 \text{ val}, \ldots, x_k \text{ val} \vdash e \text{ exp}
\]
\[
\iff
\]
\[
x_1 : \text{val}, \ldots, x_k : \text{val} \vdash_{\Sigma} \lnot e \Rightarrow \text{exp}
\]
Check that embeddings are compositional, i.e., commute with substitution:

if $x \Downarrow v'$ $\Downarrow v$ and $v \Downarrow v$, then $\lceil [v/x]v' \rceil = \lceil [\lceil v \rceil/x] \lceil v' \rceil \rceil$.

if $x \Downarrow e \Downarrow v$ and $v \Downarrow v$, then $\lceil [v/x]e \rceil = \lceil [\lceil v \rceil/x] \lceil e \rceil \rceil$. 
Adequacy of Representations

Check that embeddings are **compositional**, *i.e.*, commute with substitution:

if $x \text{ val} \vdash v' \text{ val}$ and $v \text{ val}$, then $\lbrack [v/x]v' \rbrack = \lbrack [v'/x]v' \rbrack$.

if $x \text{ val} \vdash e \text{ val}$ and $v \text{ val}$, then $\lbrack [v/x]e \rbrack = \lbrack [v'/x]e' \rbrack$.

Equivalently, check that **object language entailments** are fully and faithfully embedded in **framework entailment**.
Adequacy of Representations

Embedding for higher-order representation of evaluation:

\[ x_1 \text{ val, } \text{ret}(x_1) \Downarrow a_1, \cdots \vdash e \Downarrow a \]

\[ \iff \]

\[ x_1 : \text{val}, \_ : \text{eval}(\text{ret } x_1) \Downarrow a_1 \downarrow, \cdots \vdash_{\Sigma} \text{eval} \Downarrow e \downarrow \Downarrow a \downarrow \]
Adequacy of Representations

Embedding for higher-order representation of evaluation:

\[ x_1 \text{ val, } \text{ret}(x_1) \Downarrow a_1, \cdots \vdash e \Downarrow a \]

\[ \iff \]

\[ x_1 : \text{val, } _\vdash \text{eval(}\text{ret } x_1)^\Gamma a_1 ^\Gamma, \cdots \vdash \Sigma \text{eval} ^\Gamma e ^\Gamma \Gamma a ^\Gamma \]

Compositionality means that evaluation under assumptions is faithfully represented.
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.
Recall: a **world** is a class of LF contexts.  

( *Twelf worlds are given as series of blocks.*)
Recall: a world is a class of LF contexts.

(Twelf worlds are given as series of blocks.)

Adequacy is always relative to a specified world.
From Adequacy to Metatheory

Recall: a world is a class of LF contexts.

(Twelf worlds are given as series of blocks.)

Adequacy is always relative to a specified world.

The syntax of a language arises as the atomic objects of certain types in specified worlds.

(Perhaps a different world for each type).
Recall: a **world** is a class of LF contexts.

*(Twelf worlds are given as series of blocks.)*

Adequacy is always **relative** to a specified world.

The syntax of a language arises as the atomic objects of certain types **in specified worlds**.

*Perhaps a different world for each type.*

Mechanized metatheory reduces to **structural induction** over the canonical forms of a specified type in a specified world.

*Modulo $\alpha$-equivalence, i.e., renaming of bound variables.***
Part III

Mechanized Metatheory
Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Determinacy of evaluation.

We will consider type safety for a small language. But:
Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

• Determinacy of evaluation.
• Decidability of type checking.

We will consider type safety for a small language. But:
Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Determinacy of evaluation.
- Decidability of type checking.
- Structural properties such as weakening or substitution.

We will consider type safety for a small language. But:
Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Determinacy of evaluation.
- Decidability of type checking.
- Structural properties such as weakening or substitution.
- Cut elimination for a logic.

We will consider type safety for a small language. But:
Much standard meta-theory is easily mechanized using Twelf.

- Determinacy of evaluation.
- Decidability of type checking.
- Structural properties such as weakening or substitution.
- Cut elimination for a logic.
- Safety of compiler transformations.

We will consider type safety for a small language. But:
Metatheory With Twelf

Much standard meta-theory is easily mechanized using Twelf.

- Determinacy of evaluation.
- Decidability of type checking.
- Structural properties such as weakening or substitution.
- Cut elimination for a logic.
- Safety of compiler transformations.

We will consider 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.

- Determinacy of evaluation.
- Decidability of type checking.
- Structural properties such as weakening or substitution.
- Cut elimination for a logic.
- Safety of compiler transformations.

We will consider type safety for a small language. But:

- Scales to serious languages such as Standard ML.
- Useful for much more than just type safety.
A MinML Fragment of ML

Abstract syntax:

\[ \begin{align*}
\text{tp} &: \text{type}. \\
\text{nat} &: \text{tp}. \\
\text{arr} &: \text{tp} \to \text{tp} \to \text{tp}. \\
\text{exp} &: \text{tp} \to \text{type}. \\
\text{z} &: \text{nat}\ \text{exp}. \\
\text{s} &: \text{nat}\ \text{exp} \to \text{nat}\ \text{exp}. \\
\text{ifz} &: \text{nat}\ \text{exp} \to \text{T}\ \text{exp} \to \\
&\quad (\text{nat}\ \text{exp} \to \text{T}\ \text{exp}) \to \text{T}\ \text{exp}. \\
\end{align*} \]

(Conditional passes predecessor to non-zero case.)
A MinML Fragment of ML

Abstract syntax, cont’d:

fun : \{T1:tp\} \{T2:tp\}
    (((arr T1 T2) exp -> T1 exp -> T2 exp) ->
     (arr T1 T2) exp).

app : (arr T1 T2) exp -> T1 exp -> T2 exp.

(Functions are self-referential to support recursion.)
Values of a type:

\[
\text{value} : T \exp \rightarrow \text{type}.
\]
\[
\% \text{ mode value +E}.
\]
Dynamic Semantics of MinML

Values of a type:

value : T exp -> type.
% mode value +E.

value/z : value z.
Dynamic Semantics of MinML

Values of a type:

value : T exp -> type.
% mode value +E.

value/z : value z.
value/s : value (s E) <- value E.
Values of a type:

\[
\begin{align*}
\text{value} &: \ T \ \text{exp} \rightarrow \ \text{type}. \\
\% \ \text{mode value} &+E. \\
\text{value}/z &: \ \text{value} \ z. \\
\text{value}/s &: \ \text{value} \ (s \ E) \leftarrow \ \text{value} \ E. \\
\text{value}/fun &: \ \text{value} \ (\text{fun} \ _ \ _ \ _). \\
\end{align*}
\]
Dynamic Semantics of MinML

Structural operational semantics:

\[
\text{step} : \ T \ \text{exp} \rightarrow T \ \text{exp} \rightarrow \text{type}.
\%
\text{mode} \ \text{step} +E1 -E2.
\]
Dynamic Semantics of MinML

Structural operational semantics:

\[
\text{step} : T \text{ exp} \rightarrow T \text{ exp} \rightarrow \text{ type}.
\]

\%
mode \text{step} +E1 -E2.

\[
\text{step}/s : \text{step} (s \ E) (s \ E') \leftarrow \text{step} \ E \ E'.
\]
Dynamic Semantics of MinML

Structural operational semantics:

\[ \text{step} : \text{T exp} \rightarrow \text{T exp} \rightarrow \text{type}. \]
\[ \text{\% mode step } +\text{E1} -\text{E2}. \]

\[ \text{step/s} : \text{step (s E) (s E')} \leftarrow \text{step E E'}. \]

\[ \text{step/ifz/arg} \]
\[ : \text{step (ifz E E1 ([x] E2 x)) (ifz E' E1 ([x] E2 x))} \]
\[ \leftarrow \text{step E E'}. \]
Dynamic Semantics of MinML

Structural operational semantics:

\[
\text{step} : T \text{ exp } \rightarrow T \text{ exp } \rightarrow \text{ type.}
\]

\%
\text{ mode step } +E1 -E2.

\[
\begin{align*}
\text{step/s} & : \text{step} (s \ E) (s \ E') \leftarrow \text{step} E E'. \\
\text{step/ifz/arg} & : \text{step} (\text{ifz} E E1 ([x] E2 x)) (\text{ifz} E' E1 ([x] E2 x)) \\
& \leftarrow \text{step} E E'. \\
\text{step/ifz/z} & : \text{step} (\text{ifz} z E1 ([x] E2 x)) E1.
\end{align*}
\]
Dynamic Semantics of MinML

Structural operational semantics:

\[
\text{step} : \ T \ \text{exp} \rightarrow \ T \ \text{exp} \rightarrow \ \text{type}.
\]

\%

mode \ \text{step} \ +E1 \ -E2.

\text{step}/s : \ \text{step} \ (s \ E) \ (s \ E') \leftarrow \ \text{step} \ E \ E'.

\text{step}/ifz/arg

: \ \text{step} \ (\text{ifz} \ E \ E1 \ ([x] \ E2 \ x)) \ (\text{ifz} \ E' \ E1 \ ([x] \ E2 \ x)) \leftarrow \ \text{step} \ E \ E'.

\text{step}/ifz/z

: \ \text{step} \ (\text{ifz} \ z \ E1 \ ([x] \ E2 \ x)) \ E1.

\text{step}/ifz/s

: \ \text{step} \ (\text{ifz} \ (s \ E) \ E1 \ ([x] \ E2 \ x)) \ (E2 \ E) \leftarrow \ \text{value} \ E.
Dynamic Semantics of MinML

Structural operational semantics, cont’d:

\[
\text{step/app/fun} \\
: \text{step (app } E_1 \ E_2) \ (\text{app } E_1' \ E_2) \ \\
\leftarrow \text{step } E_1 \ E_1'.
\]
Dynamic Semantics of MinML

Structural operational semantics, cont’d:

**step/app/fun**

: step (app E1 E2) (app E1’ E2) <- step E1 E1’.

**step/app/arg**

: step (app E1 E2) (app E1 E2’) <- value E1 <- step E2 E2’.
Dynamic Semantics of MinML

Structural operational semantics, cont’d:

step/app/fun
: step (app E1 E2) (app E1’ E2)
<- step E1 E1’.

step/app/arg
: step (app E1 E2) (app E1 E2’)
<- value E1 <- step E2 E2’.

step/app/beta-v
: step
  (app (fun T1 T2 ([f] [x] E f x)) E2)
  (E (fun T1 T2 ([f] [x] E f x)) E2)
<- value E2.
We used Twelf to prove that evaluation terminates:

```
  eval : T exp → T val → type.
  %mode eval +E -V.
  ...
  %worlds () (eval _ _).
  %total D (eval D _).
```
We used Twelf to prove that evaluation terminates:

\[
\text{eval} : \text{T exp} \rightarrow \text{T val} \rightarrow \text{type}.
\]
\[
\text{\%mode eval } +\text{E } -\text{V}.
\]
\[
\ldots
\]
\[
\text{\%worlds () (eval } _\_\). \\
\text{\%total D (eval D } _\).
\]

We will use the same method to verify metatheorems!
Progress Theorem: if $e : \tau$, then either $e$ value, or there exists $e'$ such that $e \mapsto e'$. 
Progress Theorem: if $e : \tau$, then either $e$ value, or there exists $e'$ such that $e \mapsto e'$.

A constructive proof of progress defines a transformation that sends a derivation of $e : \tau$ into either a derivation of $e$ value or a derivation of $e \mapsto e'$ for some $e'$. 

Proving Metatheorems With Twelf
Proving Metatheorems With Twelf

Progress Theorem: if $e : \tau$, then either $e$ value, or there exists $e'$ such that $e \mapsto e'$.

A constructive proof of progress defines a transformation that sends a derivation of $e : \tau$ into either a derivation of $e$ value or a derivation of $e \mapsto e'$ for some $e'$.

We define this transformation as a relation, then show that it is total to prove the theorem.
Progress Theorem: if \( e : \tau \), then either \( e \text{ value} \), or there exists \( e' \) such that \( e \mapsto e' \).

A constructive proof of progress defines a transformation that sends a derivation of \( e : \tau \) into either a derivation of \( e \text{ value} \) or a derivation of \( e \mapsto e' \) for some \( e' \).

We define this transformation as a relation, then show that it is total to prove the theorem.

The content of the proof is a dependently typed program that performs the transformation and is defined for all inputs.
The intrinsic representation guarantees type preservation:

\[
\text{step} : \text{T exp} \rightarrow \text{T exp} \rightarrow \text{type}.
\]
Metatheory of MinML

The intrinsic representation guarantees type preservation:

\[
\text{step} : T\ exp \to T\ exp \to \text{type}.
\]

Progress: if \(E : T\ exp\), then either value \(E\) or steps \(E\ E'\).
The intrinsic representation guarantees type preservation:

\[ \text{step} : T \exp \rightarrow T \exp \rightarrow \text{type}. \]

**Progress:** if \( E : T \exp \), then either \( \text{value } E \) or \( \text{steps } E \ E' \).

**Progress, re-formulated:** for every object \( E : T \exp \), either

- there exists an object \( Dv \) of type \( \text{val } E \), or
- there exists an object \( Ds \) of type \( \text{steps } E \ E' \).
The intrinsic representation guarantees type preservation:

\[
\text{step} : T \text{ exp} \rightarrow T \text{ exp} \rightarrow \text{type}.
\]

Progress: if \( E : T \text{ exp} \), then either \( \text{value} \ E \) or \( \text{steps} \ E \ E' \).

Progress, re-formulated: for every object \( E : T \text{ exp} \), either
- there exists an object \( D_v \) of type \( \text{val} \ E \), or
- there exists an object \( D_s \) of type \( \text{steps} \ E \ E' \).

Progress, re-re-formulated: for every object \( E : T \text{ exp} \), there exists an object \( D \) of type \( \text{val-or-step} \ E \).
Define \texttt{val-or-step} judgement:

\begin{verbatim}
val-or-step : T exp -> type.
\end{verbatim}
Define `val-or-step` judgement:

```
val-or-step : T exp -> type.
vos/val : val-or-step E <- value E.
```

State progress theorem relationally:
```
prog : {E : T exp} val-or-step E -> type.
% mode prog +E -Dvos.
```

Thus `prog E D` relates `E : T exp` to `D : val-or-step E`.
Define \texttt{val-or-step} judgement:

\begin{align*}
\text{val-or-step} & : T \exp \rightarrow \text{type}. \\
vos/\text{val} & : \text{val-or-step} E \leftarrow \text{value} E. \\
vos/\text{step} & : \text{val-or-step} E \leftarrow \text{step} E. 
\end{align*}
Define \textit{val-or-step} judgement:

\begin{align*}
\text{val-or-step} & : T \text{ exp} \to \text{ type}. \\
\text{vos/val} & : \text{val-or-step} E \leftarrow \text{value} E. \\
\text{vos/step} & : \text{val-or-step} E \leftarrow \text{step} E.
\end{align*}

State \textit{progress theorem} relationally:

\begin{align*}
\text{prog} & : \{E : T \text{ exp}\} \text{ val-or-step} E \to \text{ type}. \\
\% \text{ mode prog} & +E \ -\text{Dvos}.
\end{align*}
Define \texttt{val-or-step} judgement:

\[
\begin{align*}
\text{val-or-step} & : T \exp \to \text{type}. \\
vos/\text{val} & : \text{val-or-step} E \leftarrow \text{value} E. \\
vos/\text{step} & : \text{val-or-step} E \leftarrow \text{step} E .
\end{align*}
\]

State \textit{progress theorem} relationally:

\[
\text{prog} : \{ E : T \exp \} \text{val-or-step} E \to \text{type}. \\
\% \text{mode prog } +E -\text{Dvos}.
\]

Thus \texttt{prog E D} relates \( E : T \exp \) to \( D : \text{val-or-step} E \).
Axiomatize the progress relation:
- : prog z (vos/val value/z).
Axiomatize the progress relation:

- : prog z (vos/val value/z).
- : prog (s E) Dvos’
  <- prog E Dvos
  <- prog/s Dvos Dvos’.
Progress Theorem

Axiomatize the progress relation:

- : prog z (vos/val value/z).
- : prog (s E) Dvos'
  <- prog E Dvos
  <- prog/s Dvos Dvos’.
- : prog (ifz E E1 ([x] E2 x)) (vos/step Dstep)
  <- prog E Dvos
  <- prog/ifz Dvos _ _ Dstep.
Axiomatize the progress relation, cont’d:

- : prog (fun _) (vos/val value/fun).
Axiomatize the progress relation, cont’d:

- : prog (fun _ _ _) (vos/val value/fun).
- : prog (app E1 E2) (vos/step Dstep)
  <- prog E1 Dvos1
  <- prog E2 Dvos2
  <- prog/app Dvos1 Dvos2 Dstep.
Axiomatize the progress relation, cont’d:

- : prog (fun _ _ _) (vos/val value/fun).
- : prog (app E1 E2) (vos/step Dstep)
  <- prog E1 Dvos1
  <- prog E2 Dvos2
  <- prog/app Dvos1 Dvos2 Dstep.

Prove the theorem:

%worlds () (prog _ _).
%total Dof (prog Dof _).
Progress Theorem

To quote Paul Taylor, “Theorems, like management, get all the credit, but the lemmas do all the work.”
To quote Paul Taylor, “Theorems, like management, get all the credit, but the lemmas do all the work.”

We have reduced progress to three lemmas.
Progress Theorem

To quote Paul Taylor, “Theorems, like management, get all the credit, but the lemmas do all the work.”

We have reduced progress to three lemmas.

If either value $E_0$ or stepsto $E_0$ $E_0'$, then
To quote Paul Taylor, “Theorems, like management, get all the credit, but the lemmas do all the work.”

We have reduced progress to three lemmas.

If either value $E_0$ or stepsto $E_0$ $E_0'$, then

1. either value $(s \ E_0)$ or stepsto $(s \ E_0) (s \ E_0')$;
Progress Theorem

To quote Paul Taylor, “Theorems, like management, get all the credit, but the lemmas do all the work.”

We have reduced progress to three lemmas.

If either value E0 or stepsto E0 E0’, then

1. either value (s E0) or stepsto (s E0) (s E0’);
2. stepsto (ifz E0 E1 ([x] E2 x)) E’;
Progress Theorem

To quote Paul Taylor, “Theorems, like management, get all the credit, but the lemmas do all the work.”

We have reduced progress to three lemmas.

If either value $E_0$ or stepsto $E_0$ $E_0'$, then

1. either value $(s\ E_0)$ or stepsto $(s\ E_0)\ (s\ E_0')$;
2. stepsto (ifz $E_0$ $E_1$ ($[x]$ $E_2$ $x$)) $E'$;
3. if value $E_1$ or stepsto $E_1$ $E_1'$, then
   stepsto (app $E_0$ $E_1$) $E'$. 
Progress Lemma: Successor

prog/s
    : val-or-step E -> val-or-step (s E) -> type.
% mode prog/s +Dvos1 -Dvos2.
Progress Lemma: Successor

prog/s
  : val-or-step E -> val-or-step (s E) -> type.
% mode prog/s +Dvos1 -Dvos2.
- : prog/s
   (vos/step Dstep)
   (vos/step (step/s Dstep)).
Progress Lemma: Successor

\[ \text{prog/s} : \text{val-or-step } E \rightarrow \text{val-or-step } (s \ E) \rightarrow \text{type.} \%
\]
\[ \text{mode prog/s } +\text{Dvos1 } -\text{Dvos2.} \]

\[ \text{prog/s} \]
\[ (\text{vos/step } \text{Dstep}) \]
\[ (\text{vos/step } (\text{step/s } \text{Dstep})). \]

\[ \text{prog/s} \]
\[ (\text{vos/val } \text{Dval}) \]
\[ (\text{vos/val } (\text{value/s } \text{Dval})). \]
Progress Lemma: Successor

prog/s
  : val-or-step E → val-or-step (s E) → type.
% mode prog/s +Dvos1 -Dvos2.
- : prog/s
  (vos/step Dstep)
  (vos/step (step/s Dstep)).
- : prog/s
  (vos/val Dval)
  (vos/val (value/s Dval)).
% worlds () (prog/s _ _).
% total (prog/s _ _).
Progress Lemma: Conditional

\[
\text{prog/ifz} : \text{val-or-step} (E : \text{nat exp}) \\
\quad \rightarrow \{E_1\} \{E_2\} (\text{step} (\text{ifz} \ E \ E_1 ([x] \ E_2 \ x)) \ E') \\
\quad \rightarrow \text{type}.
\]
%mode prog/ifz +E +E1 +E2 -Dstep.
Progress Lemma: Conditional

\[
\text{prog/ifz} : \text{val-or-step} \ (E : \text{nat exp}) \\
\quad \rightarrow \{E_1\} \ \{E_2\} \ (\text{step} \ (\text{ifz} \ E \ E_1 \ ([x] \ E_2 \ x)) \ E') \\
\quad \rightarrow \text{type}.
\]

\%mode prog/ifz +E +E1 +E2 -Dstep.

\(- : \text{prog/ifz} \ (\text{vos/step Dstep}) \ _ \ _ \ (\text{step/ifz/arg Dstep}).\)
Progress Lemma: Conditional

\[
\text{prog/ifz : val-or-step (E : nat exp)}
\quad \rightarrow \{E1\} \{E2\} (\text{step (ifz E E1 ([x] E2 x)) E'})
\quad \rightarrow \text{type}.
\]

%mode prog/ifz +E +E1 +E2 -Dstep.

- : prog/ifz (vos/val value/z) _ _ step/ifz/z.
Progress Lemma: Conditional

prog/ifz : val-or-step (E : nat exp)
  -> \{E1\} \{E2\} (step (ifz E E1 ([x] E2 x)) E')
  -> type.
%mode prog/ifz +E +E1 +E2 -Dstep.
- : prog/ifz (vos/val value/z) _ _ step/ifz/z.
- : prog/ifz
  (vos/val (value/s Dval))
  _ _
  (step/ifz/s Dval).
Progress Lemma: Conditional

prog/ifz : val-or-step (E : nat exp)
    -> \{E1\} \{E2\} (step (ifz E E1 ([x] E2 x)) E')
    -> type.
%mode prog/ifz +E +E1 +E2 -Dstep.
- : prog/ifz (vos/val value/z) _ _ step/ifz/z.
- : prog/ifz
    (vos/val (value/s Dval))
    _ _
    (step/ifz/s Dval).
%worlds () (prog/ifz _ _ _ _).
%total (prog/ifz _ _ _ _).
Progress Lemma: Application

prog/app
    : val-or-step (E1 : (arr T2 T) exp)
    -> val-or-step (E2 : T2 exp)
    -> step (app E1 E2) E'
    -> type.
%mode prog/app +Dvos1 +Dvos2 -Dstep.
Progress Lemma: Application

prog/app
  : val-or-step (E1 : (arr T2 T) exp)
  -> val-or-step (E2 : T2 exp)
  -> step (app E1 E2) E'
  -> type.
%mode prog/app +Dvos1 +Dvos2 -Dstep.
- : prog/app
  (vos/step Dstep1)
- (step/app/fun Dstep1).
Progress Lemma: Application

\[- : \text{prog/app} \]
\[
(\text{vos/val Dval1}) \\
(\text{vos/step Dstep2}) \\
(\text{step/app/arg Dstep2 Dval1}).
\]
Progress Lemma: Application

- : prog/app
  (vos/val Dval1)
  (vos/step Dstep2)
  (step/app/arg Dstep2 Dval1).

- : prog/app
  (vos/val Dval1)
  (vos/val Dval2)
  (step/app/beta-v Dval2).
Progress Lemma: Application

- : prog/app
  (vos/val Dval1)
  (vos/step Dstep2)
  (step/app/arg Dstep2 Dval1).

- : prog/app
  (vos/val Dval1)
  (vos/val Dval2)
  (step/app/beta-v Dval2).

%worlds () (prog/app _ _ _).
%total (prog/app _ _ _).
Twelf is in **daily** use as a tool for language design and implementation.

- **Natural** pattern-driven, dependently typed programming with direct support for structural features of languages and logics.
Where From Here?

Twelf is in daily use as a tool for language design and implementation.

- **Natural** pattern-driven, dependently typed programming with direct support for structural features of languages and logics.
- Readable and maintainable **proofs**, not proof scripts.

Try it, you'll like it!
Twelf is in daily use as a tool for language design and implementation.

- **Natural** pattern-driven, dependently typed programming with direct support for structural features of languages and logics.
- Readable and maintainable proofs, not proof scripts.
- Imposes **healthy** reality and sanity check on language designs.

Try it, you'll like it!
Twelf is in daily use as a tool for language design and implementation.

- **Natural** pattern-driven, dependently typed programming with direct support for structural features of languages and logics.
- Readable and maintainable **proofs**, not proof scripts.
- Imposes **healthy** reality and sanity check on language designs.
- Exposes, and **helps correct**, subtle design errors early in the process. (Greatly diminishes POPL deadline anxiety!)

Try it, you’ll like it!