

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:

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.*

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**.

Simple Arithmetic Expressions

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

This judgement is inductively defined by these two **rules**:

$$\frac{n \mathbf{nat}}{\bar{n} \mathbf{exp}}$$

$$\frac{e_1 \mathbf{exp} \quad e_2 \mathbf{exp}}{e_1 + e_2 \mathbf{exp}}$$

Simple Arithmetic Expressions

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

This judgement is inductively defined by these two **rules**:

$$\frac{n \mathbf{nat}}{\bar{n} \mathbf{exp}} \qquad \frac{e_1 \mathbf{exp} \quad e_2 \mathbf{exp}}{e_1 + e_2 \mathbf{exp}}$$

The judgement $e \mathbf{exp}$ 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.

```
nat : type.  
z : nat.  
s : nat -> nat.
```

Define abstract syntax of expressions.

```
exp : type.  
num : nat -> exp.  
plus : exp -> exp -> exp.
```

Simple Arithmetic Expressions

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

```
⌈2 + 3⌋ = plus (num (s (s z))) (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`.

`⌈2 + 3⌉ = 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:

$e \text{ **exp** } \text{ iff } \lceil e \rceil : \text{exp}.$

Evaluation Judgement

The **evaluation judgement** $e \Downarrow a$ states that the expression e evaluates to the answer a .

It is defined by these *rules*:

$$\overline{\overline{n} \Downarrow \overline{n}} \qquad \frac{e_1 \Downarrow \overline{n_1} \quad e_2 \Downarrow \overline{n_2} \quad n = n_1 + n_2}{e_1 + e_2 \Downarrow \overline{n}}$$

It is the **strongest** judgement closed under these rules.

Judgements as Types

The judgement is represented by a **family of types**:

`eval : exp -> ans -> type.`

Judgements as Types

Define the type of **answers**:

```
ans : type.
```

```
anat : nat -> ans.
```

The judgement is represented by a **family of types**:

```
eval : exp -> ans -> type.
```

Judgements as Types

Define the type of **answers**:

```
ans : type.  
anat : nat -> ans.
```

The judgement is represented by a **family of types**:

```
eval : exp -> ans -> type.
```

The LF type $\text{eval} \ulcorner e \urcorner \ulcorner a \urcorner$ represents **derivations** of $e \Downarrow a$.

$$\nabla : e \Downarrow a \quad \text{iff} \quad \ulcorner \nabla \urcorner : \text{eval} \ulcorner e \urcorner \ulcorner a \urcorner$$

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.
```

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.
```

But what is meant by **add**?

Rules as Generators

Must define addition on natural numbers as well:

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

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

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

Rules as Generators

Must define addition on natural numbers as well:

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

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

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

Adequacy: $\exists D : \text{add } \ulcorner m \urcorner \ulcorner n \urcorner \ulcorner p \urcorner \text{ iff } m + n = p.$

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).

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.

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.

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.

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

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.

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

This is sufficient for a **large body** of metareasoning!

Mode Checking

Let's verify that `add` defines a **total relation**.

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

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

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

Mode Checking

Let's verify that `add` defines a **total relation**.

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

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

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

That is, M and N determine P in `add M N P`:

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

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

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

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

Coverage and Termination

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

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

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 _ _).
```

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 `add` is to be proved total on **closed** terms of type `nat` by structural induction on the first argument.

Mechanized Metatheory

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

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

Mechanized Metatheory

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!

Evaluation Terminates

We may just as easily prove that **evaluation** terminates!

```
%worlds () (eval _ _).  
%total E (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:

let x **be** e_1 **in** e_2

with the meaning that x stands for e_1 within e_2 .

Adding Bindings

Now enrich expressions with a **binding** construct:

let x **be** e_1 **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:

let x **be** e_1 **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:

let x **be** $\bar{3}$ **in** $x + x$ is **let** y **be** $\bar{3}$ **in** $y + y$.

Adding Bindings

Now enrich expressions with a **binding** construct:

let x **be** e_1 **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:

let x **be** $\bar{3}$ **in** $x + x$ is **let** y **be** $\bar{3}$ **in** $y + y$.

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

$[\bar{3}/x](x + x)$ is $\bar{3} + \bar{3}$.

Adding Bindings

Enrich expressions with a let-binding:

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

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:

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:

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:

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

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

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!

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:

$$\lceil \text{let } x \text{ be } e_1 \text{ in } e_2 \rceil = \text{let } \lceil e_1 \rceil ([x : \text{exp}] \lceil e_2 \rceil).$$

where the λ -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:

$$\lceil \text{let } x \text{ be } e_1 \text{ in } e_2 \rceil = \text{let } \lceil e_1 \rceil \ ([x : \text{exp}] \lceil e_2 \rceil).$$

where the **λ -abstraction**, $[x : \text{exp}]$, expresses the binding and scope of x in e_2 .

Evaluation, Revisited

Consider the rule for evaluation of a `let`:

$$\frac{e_1 \Downarrow \bar{n}_1 \quad [\bar{n}_1/x]e_2 \Downarrow a}{\mathbf{let\ } x \mathbf{ be\ } e_1 \mathbf{ in\ } e_2 \Downarrow a}$$

Evaluation, Revisited

Consider the rule for evaluation of a `let`:

$$\frac{e_1 \Downarrow \bar{n}_1 \quad [\bar{n}_1/x]e_2 \Downarrow a}{\mathbf{let\ } x \mathbf{ be\ } e_1 \mathbf{ in\ } e_2 \Downarrow a}$$

Formulated in LF:

```
eval/let
```

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

```
<- eval E1 (anum N1)
```

```
<- eval (E2 (num N1)) A.
```

Evaluation, Revisited

Consider the rule for evaluation of a `let`:

$$\frac{e_1 \Downarrow \bar{n}_1 \quad [\bar{n}_1/x]e_2 \Downarrow a}{\mathbf{let\ } x \mathbf{ be\ } e_1 \mathbf{ in\ } e_2 \Downarrow a}$$

Formulated in LF:

```
eval/let
```

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

```
  <- eval E1 (anum N1)
```

```
  <- eval (E2 (num N1)) A.
```

Substitution is provided **for free** by LF!

Enforcing Stronger Invariants

We can track that variables are bound to values.

```
val : type.  
num : nat -> val.  
exp : type.  
ret : val -> exp.  
plus : exp -> exp -> exp.  
let : exp -> (val -> exp) -> exp.
```

As a rule it is good practice to use types to enforce invariants on a representation.

Higher-Order Rules

We may use **hypothetical judgements** to represent bindings:

$$\frac{e_1 \Downarrow a_1 \quad \mathbf{ret} \ x \ \Downarrow a_1 \vdash e_2 \ \Downarrow a_2}{\mathbf{let} \ x \ \mathbf{be} \ e_1 \ \mathbf{in} \ e_2 \ \Downarrow a_2}$$

Higher-Order Rules

We may use **hypothetical judgements** to represent bindings:

$$\frac{e_1 \Downarrow a_1 \quad \mathbf{ret\ } x \Downarrow a_1 \vdash e_2 \Downarrow a_2}{\mathbf{let\ } x \mathbf{ be\ } e_1 \mathbf{ in\ } e_2 \Downarrow a_2}$$

The **evaluation hypothesis** governs the variable x in e_2 .

$$\mathbf{ret\ } x \Downarrow \bar{3} \vdash (\mathbf{ret}, x) + (\mathbf{ret\ } \bar{4}) \Downarrow \bar{7}$$

Higher-Order Rules in LF

Higher-order rules are represented using **higher-order types**:

```
eval/let
```

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

```
  <- eval E1 A1
```

```
  <- ({x:val} eval (ret x) A1 -> eval (E2 x) A2).
```

Higher-Order Rules in LF

Higher-order rules are represented using **higher-order types**:

```
eval/let
  : eval (let E1 ([x] E2 x)) A
  <- eval E1 A1
  <- ({x:val} eval (ret x) A1 -> eval (E2 x) A2).
```

The **general hypothetical judgement** expresses that body is evaluated relative to

Higher-Order Rules in LF

Higher-order rules are represented using **higher-order types**:

```
eval/let
  : eval (let E1 ([x] E2 x)) A
  <- eval E1 A1
  <- ({x:val} eval (ret x) A1 -> eval (E2 x) A2).
```

The **general hypothetical judgement** expresses that body is evaluated relative to

- A **fresh variable**, x ;

Higher-Order Rules in LF

Higher-order rules are represented using **higher-order types**:

```
eval/let
  : eval (let E1 ([x] E2 x)) A
  <- eval E1 A1
  <- ({x:val} eval (ret x) A1 -> eval (E2 x) A2).
```

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 axiom** 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.

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**!

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 $\ulcorner J_1 \urcorner \rightarrow \ulcorner J_2 \urcorner$.

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 $\ulcorner J_1 \urcorner \rightarrow \ulcorner J_2 \urcorner$.
- $\lambda_{x:A} J$ represented by $\Pi_{x:\ulcorner A \urcorner} \ulcorner J \urcorner$.

Adequacy and Worlds

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

$$\nabla : e \Downarrow a \quad \text{iff} \quad \ulcorner \nabla \urcorner : \text{eval} \ulcorner e \urcorner \ulcorner a \urcorner.$$

Adequacy and Worlds

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

$$\nabla : e \Downarrow a \quad \text{iff} \quad \ulcorner \nabla \urcorner : \text{eval} \ulcorner e \urcorner \ulcorner a \urcorner.$$

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.

$$\nabla : e \Downarrow a \quad \text{iff} \quad \ulcorner \nabla \urcorner : \text{eval} \ulcorner e \urcorner \ulcorner a \urcorner.$$

This is expressed by the **%worlds** declaration:

```
%worlds () (eval _ _).  
%total E (eval E _).
```

Adequacy for the higher-order formulation must consider **derivations under hypotheses**.

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:

$$\begin{aligned} \nabla : \text{ret } x_1 \Downarrow a_1, \dots \Downarrow a \\ \text{iff} \\ x_1 : \text{val}, _ : \text{eval}(\text{ret } x_1) a_1, \dots \vdash \ulcorner \nabla \urcorner : \text{eval} \ulcorner e \urcorner \ulcorner a \urcorner \end{aligned}$$

Adequacy and Worlds

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

```
%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.

Typed Expressions

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**.

```
tp : type.  
number : tp.  
string : tp.
```

Typed Expressions

Introduce an LF type of **expression types**.

```
tp : type.  
number : tp.  
string : tp.
```

Introduce an LF family of **expressions of a type**.

```
exp : tp -> type.  
num : nat -> number exp.  
lit : str -> string exp.  
plus : number exp -> number exp -> number exp.  
append : string exp -> string exp -> string exp.
```

Typed Expressions

Introduce an LF type of **expression types**.

```
tp : type.  
number : tp.  
string : tp.
```

Introduce an LF family of **expressions of a type**.

```
exp : tp -> type.  
num : nat -> number exp.  
lit : str -> string exp.  
plus : number exp -> number exp -> number exp.  
append : string exp -> string exp -> string exp.
```

(Must also define type str of strings.)

Typed Expressions

Evaluation is defined **exactly** as before, but with types.

```
ans : tp -> type.
```

```
anum : nat -> ans.
```

```
astr : str -> ans.
```

```
eval : T exp -> T ans -> type.
```

Typed Expressions

Evaluation is defined **exactly** as before, but with types.

```
ans : tp -> type.
```

```
anum : nat -> ans.
```

```
astr : str -> ans.
```

```
eval : T exp -> T ans -> type.
```

The LF type of `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.
```

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.

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

Theory of Representation

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

Theory of Representation

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 $\lceil o \rceil$ of type $\lceil C \rceil$.

Theory of Representation

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 $\lceil o \rceil$ of type $\lceil C \rceil$.
- **Faithful**: every canonical form of type $\lceil C \rceil$ represents a unique syntactic object of class C .

Theory of Representation

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

Theory of Representation

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

Let us now make these ideas precise.

The LF Type Theory

LF is a **dependently typed** λ -calculus with two levels:

The LF Type Theory

LF is a **dependently typed** λ -calculus with two levels:

- **Families**, A , classified by **Kinds**, K .

The LF Type Theory

LF is a **dependently typed** λ -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** λ -calculus with two levels:

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

The syntax is classified into levels:

<i>Kind</i>	$K ::= \mathbf{type} \mid \Pi_{x:A}K$
<i>Family</i>	$A ::= a \mid A M \mid \Pi_{x:A}B$
<i>Canonical Object</i>	$M ::= R \mid \lambda_{x:A}M$
<i>Atomic Object</i>	$R ::= x \mid c \mid R M$

The LF Type Theory

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

The LF Type Theory

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

- No β -redices: $\lambda_{x:A} M N$.

The LF Type Theory

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

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

The LF Type Theory

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

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

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

The LF Type Theory

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

- No β -redices: $\lambda_{x:A} M N$.
- Fully η -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 β -redices: $\lambda_{x:A} M N$.
- Fully η -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.

The LF Type Theory

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

- No β -redices: $\lambda_{x:A} M N$.
- Fully η -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**, Γ , is a sequence of variable declarations:

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

wherein each A_i may involve the preceding variables.

The LF Type Theory

An LF **context**, Γ , is a sequence of variable declarations:

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

wherein each A_i may involve the preceding variables.

An LF **signature**, Σ , is a sequence of constant declarations:

$$\left\{ \begin{array}{l} a_1 : K_1 \\ c_1 : A_1 \end{array} \right\}, \dots, \left\{ \begin{array}{l} a_m : K_m \\ c_m : A_m \end{array} \right\}$$

where each A_i or K_i may involve the preceding constants.

The LF Type Theory

Formation judgements of LF:

$$\Gamma \vdash_{\Sigma} K \text{ kind}$$

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

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

$$\vdash_{\Sigma} \Gamma \text{ ok} \quad \vdash \Sigma \text{ ok}$$

Canonical objects are **analyzed**, atomic objects are **synthesized**.

The LF Type Theory

Substitution judgements of LF:

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

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

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

The LF Type Theory

Substitution judgements of LF:

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

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

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

The **critical case** threatens termination:

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

The LF Type Theory

Substitution judgements of LF:

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

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

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

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:

$$\overline{\Gamma \vdash_{\Sigma_1, c:A, \Sigma_2} c \Rightarrow A}$$

$$\overline{\Gamma_1, x : A, \Gamma_2 \vdash_{\Sigma} x \Rightarrow A}$$

Atomic Objects

Variables and constants:

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

Function application:

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

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:

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

Type Families

Constants:

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

Family instantiation:

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

Type Families

Products of families:

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

Kinds

The kind of types:

$$\overline{\Gamma \vdash_{\Sigma} \text{type kind}}$$

The kind of types:

$$\overline{\Gamma \vdash_{\Sigma} \text{type kind}}$$

Product of a kind family:

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

Representation Methodology

Central principle: capture entailments.

Representation Methodology

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).

Representation Methodology

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.

Representation Methodology

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:

- v **val** values

Representation Methodology

Syntactic classes for arithmetic expressions:

- v **val** values
- e **exp** expressions

Representation Methodology

Syntactic classes for arithmetic expressions:

- v **val** values
- e **exp** expressions
- a **ans** answers

Representation Methodology

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 **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

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

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

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:

$$\begin{array}{c} x_1 \mathbf{val}, \dots, x_k \mathbf{val} \vdash v \mathbf{val} \\ \longleftrightarrow \\ x_1 : \mathbf{val}, \dots, x_k : \mathbf{val} \vdash_{\Sigma} \lceil v \rceil \Rightarrow \mathbf{val} \end{array}$$

Representation Methodology

Embed object-language entailments as LF entailments:

$$x_1 \mathbf{val}, \dots, x_k \mathbf{val} \vdash v \mathbf{val}$$
$$\longleftrightarrow$$
$$x_1 : \mathbf{val}, \dots, x_k : \mathbf{val} \vdash_{\Sigma} \lceil v \rceil \Rightarrow \mathbf{val}$$
$$x_1 \mathbf{val}, \dots, x_k \mathbf{val} \vdash e \mathbf{exp}$$
$$\longleftrightarrow$$
$$x_1 : \mathbf{val}, \dots, x_k : \mathbf{val} \vdash_{\Sigma} \lceil e \rceil \Rightarrow \mathbf{exp}$$

Adequacy of Representations

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

if $x \mathbf{val} \vdash v' \mathbf{val}$ and $v \mathbf{val}$, then $\ulcorner [v/x]v' \urcorner = \ulcorner v \urcorner / x \urcorner \ulcorner v' \urcorner$.

if $x \mathbf{val} \vdash e \mathbf{val}$ and $v \mathbf{val}$, then $\ulcorner [v/x]e \urcorner = \ulcorner v \urcorner / x \urcorner \ulcorner e \urcorner$.

Adequacy of Representations

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

if $x \mathbf{val} \vdash v' \mathbf{val}$ and $v \mathbf{val}$, then $\ulcorner [v/x]v' \urcorner = \ulcorner v \urcorner / x \urcorner \ulcorner v' \urcorner$.

if $x \mathbf{val} \vdash e \mathbf{val}$ and $v \mathbf{val}$, then $\ulcorner [v/x]e \urcorner = \ulcorner v \urcorner / x \urcorner \ulcorner e \urcorner$.

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

Adequacy of Representations

Embedding for **higher-order representation** of evaluation:

$$\begin{array}{c} x_1 \text{ val}, \text{ret}(x_1) \Downarrow a_1, \dots \vdash e \Downarrow a \\ \longleftrightarrow \\ x_1 : \text{val}, _ : \text{eval}(\text{ret } x_1) \ulcorner a_1 \urcorner, \dots \vdash_{\Sigma} \text{eval} \ulcorner e \urcorner \ulcorner a \urcorner \end{array}$$

Adequacy of Representations

Embedding for **higher-order representation** of evaluation:

$$\begin{array}{c} x_1 \text{ val}, \text{ret}(x_1) \Downarrow a_1, \dots \vdash e \Downarrow a \\ \longleftrightarrow \\ x_1 : \text{val}, _ : \text{eval}(\text{ret } x_1) \ulcorner a_1 \urcorner, \dots \vdash_{\Sigma} \text{eval} \ulcorner e \urcorner \ulcorner a \urcorner \end{array}$$

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.

From Adequacy to Metatheory

Recall: a **world** is a class of LF contexts.

(Twelf worlds are given as series of blocks.)

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.

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).

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).

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

(Modulo α -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:**

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:**

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:

`tp : type.`

`nat : tp.`

`arr : tp -> tp -> tp.`

`exp : tp -> type.`

`z : nat exp.`

`s : nat exp -> nat exp.`

`ifz : nat exp -> T exp ->`
`(nat exp -> T exp) -> T exp.`

(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.)

Dynamic Semantics of MinML

Values of a type:

```
value : T exp -> type.  
% 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 \rightarrow type.

% mode value +E.

value/z : value z.

value/s : value (s E) \leftarrow value E.

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.
```

```
value/fun : value (fun _ _ _).
```

Dynamic Semantics of MinML

Structural operational semantics:

`step : T exp -> T exp -> type.`

`% mode step +E1 -E2.`

Dynamic Semantics of MinML

Structural operational semantics:

`step : T exp -> T exp -> type.`

`% mode step +E1 -E2.`

`step/s : step (s E) (s E') <- step E E'.`

Dynamic Semantics of MinML

Structural operational semantics:

`step : T exp -> T exp -> type.`

`% mode step +E1 -E2.`

`step/s : step (s E) (s E') <- step E E'.`

`step/ifz/arg`

`: step (ifz E E1 ([x] E2 x)) (ifz E' E1 ([x] E2 x))`
`<- step E E'.`

Dynamic Semantics of MinML

Structural operational semantics:

step : $T \text{ exp} \rightarrow T \text{ exp} \rightarrow \text{type}$.

% mode step +E1 -E2.

step/s : step (s E) (s E') <- step E E'.

step/ifz/arg

: step (ifz E E1 ([x] E2 x)) (ifz E' E1 ([x] E2 x))
<- step E E'.

step/ifz/z

: step (ifz z E1 ([x] E2 x)) E1.

Dynamic Semantics of MinML

Structural operational semantics:

step : $T \text{ exp} \rightarrow T \text{ exp} \rightarrow \text{type}$.

% mode step +E1 -E2.

step/s : step (s E) (s E') <- step E E'.

step/ifz/arg

: step (ifz E E1 ([x] E2 x)) (ifz E' E1 ([x] E2 x))
<- step E E'.

step/ifz/z

: step (ifz z E1 ([x] E2 x)) E1.

step/ifz/s

: step (ifz (s E) E1 ([x] E2 x)) (E2 E)
<- value E.

Dynamic Semantics of MinML

Structural operational semantics, cont'd:

step/app/fun

: step (app E1 E2) (app E1' E2)

<- step E1 E1'.

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.
```

Proving Metatheorems With Twelf

We used Twelf to prove that evaluation terminates:

```
eval : T exp -> T val -> type.  
%mode eval +E -V.  
...  
%worlds () (eval _ _).  
%total D (eval D _).
```

Proving Metatheorems With Twelf

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 will use the **same** method to verify metatheorems!

Proving Metatheorems With Twelf

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

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' .

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.

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.

The content of the proof is a **dependently typed program** that performs the transformation and is defined for all inputs.

Metatheory of MinML

The intrinsic representation guarantees **type preservation**:

$\text{step} : \mathbb{T} \text{ exp} \rightarrow \mathbb{T} \text{ exp} \rightarrow \text{type}.$

Metatheory of MinML

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 value E or steps $E E'$.

Metatheory of MinML

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 value E or steps $E E'$.

Progress, re-formulated: for every object $E : T \text{ exp}$, either

- there exists an object Dv of type $\text{val } E$, or
- there exists an object Ds of type $\text{steps } E E'$.

Metatheory of MinML

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 value E or steps $E E'$.

Progress, re-formulated: for every object $E : T \text{ exp}$, either

- there exists an object Dv of type $\text{val } E$, or
- there exists an object Ds of type $\text{steps } E E'$.

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

Progress Theorem

Define **val-or-step** judgement:

`val-or-step : T exp -> type.`

Progress Theorem

Define **val-or-step** judgement:

`val-or-step : T exp -> type.`

`vos/val : val-or-step E <- value E.`

Progress Theorem

Define **val-or-step** judgement:

`val-or-step : T exp -> type.`

`vos/val : val-or-step E <- value E.`

`vos/step : val-or-step E <- step E ..`

Progress Theorem

Define **val-or-step** judgement:

```
val-or-step : T exp -> type.
```

```
vos/val : val-or-step E <- value E.
```

```
vos/step : val-or-step E <- step E ..
```

State **progress theorem** relationally:

```
prog : {E : T exp} val-or-step E -> type.
```

```
% mode prog +E -Dvos.
```

Progress Theorem

Define **val-or-step** judgement:

```
val-or-step : T exp -> type.
```

```
vos/val : val-or-step E <- value E.
```

```
vos/step : val-or-step E <- step E ..
```

State **progress theorem** relationally:

```
prog : {E : T exp} val-or-step E -> type.
```

```
% mode prog +E -Dvos.
```

Thus $\text{prog } E \ D$ relates $E : T \text{ exp}$ to $D : \text{val-or-step } E$.

Progress Theorem

Axiomatize the progress relation:

- : prog z (vos/val value/z).

Progress Theorem

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.

Progress Theorem

Axiomatize the progress relation, cont'd:

- : prog (fun _ _ _) (vos/val value/fun).

Progress Theorem

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.

Progress Theorem

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.”

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.

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

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 $\text{stepsto } E0 \ E0'$, then

- 1 either value $(s \ E0)$ or $\text{stepsto } (s \ E0) \ (s \ E0')$;

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 $E0$ or $\text{stepsto } E0 \ E0'$, then

- 1 either value $(s \ E0)$ or $\text{stepsto } (s \ E0) \ (s \ E0')$;
- 2 $\text{stepsto } (\text{ifz } E0 \ E1 \ ([x] \ E2 \ x)) \ E'$;
- 3 if value $E1$ or $\text{stepsto } E1 \ E1'$, then $\text{stepsto } (\text{app } E0 \ E1) \ E'$.

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

```
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)).
```

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

```
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.
```

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/step Dstep) - - (step/ifz/arg Dstep).
```

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/step Dstep) _ _ (step/ifz/arg 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/step Dstep) _ _ (step/ifz/arg 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/step Dstep) _ _ (step/ifz/arg 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

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

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.

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.

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.
- Imposes **healthy** reality and sanity check on language designs.

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.
- 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!