Twelf Tutorial
Twelf Encoding of Minilang

John Altidor
Proving language properties are important.
  ▶ Rule out certain errors (e.g. assuming wrong number of bytes for an object).
  ▶ Well-defined behavior throughout execution (e.g. no segmentation fault or accessing wrong parts of memory).
  ▶ Publishing.
Motivation

- Proving language properties are important.
  - Rule out certain errors (e.g. assuming wrong number of bytes for an object).
  - Well-defined behavior throughout execution (e.g. no segmentation fault or accessing wrong parts of memory).
  - Publishing.

- But proofs are long, error prone, and difficult to validate.
  - +20 pages is common for a type safety proof.
Typical Proof Structure

- Example taken from type soundness proof of TameFJ calculus.

Lemma 33 (Inversion Lemma (method invocation)).

If:

a. $\Delta; \Gamma \vdash e, \langle P\rangle m(\bar{e}) : T \vdash \Delta'$

b. $\emptyset \vdash \Delta$ OK

c. $\Delta \vdash \Delta'$ OK

d. $\forall x \in dom(\Gamma) : \Delta \vdash \Gamma(x)$ OK

then:

there exists $\Delta_n$

where:

$\Delta', \Delta_n = \Delta'', \overline{\Delta}$

$\Delta \vdash \Delta', \Delta_n$ OK

$\Delta; \Gamma \vdash e : \exists \Delta''.N \mid \emptyset$

$mType(m, N) = \langle Y \triangleleft B \triangleright U \rightarrow U$

$\Delta; \Gamma \vdash e : \exists \Delta'.R \mid \emptyset$

$match(sift(R, U, Y), P, Y, T)$

$\Delta \vdash P$ OK

$\Delta, \Delta'', \overline{\Delta} \vdash \overline{[T/Y]B}$

Proof by structural induction on the derivation of $\Delta; \Gamma \vdash e. \langle P\rangle m(\bar{e}) : T \vdash \Delta'$

with a case analysis on the last step:

- Lots of steps, lemmas, and opportunities for errors in proofs of language properties.
What is a Proof Assistant

Multiple ways of proving theorems with a computer:

- **Automatic theorem provers** find complete proofs on their own.
  - Not all proofs can be derived automatically.

- **Proof checkers** simply verify the proofs they are given.
  - These proofs must be specified in an extremely detailed, low-level form.

- **Proof assistants** are a hybrid of both.
  - "Hard steps" of proofs (the ones requiring deep insight) are provided by human.
  - "Easy steps" of proofs can be filled in automatically.

(above bullet points taken from UPenn’s Software Foundations course slides)
Multiple ways of proving theorems with a computer:

- **Automatic theorem provers** find complete proofs on their own.
  - Not all proofs can be derived automatically.

- **Proof checkers** simply verify the proofs they are given.
  - These proofs must be specified in an extremely detailed, low-level form.
What is a Proof Assistant

Multiple ways of proving theorems with a computer:

- **Automatic theorem provers** find complete proofs on their own.
  - Not all proofs can be derived automatically.

- **Proof checkers** simply verify the proofs they are given.
  - These proofs must be specified in an extremely detailed, low-level form.

- **Proof assistants** are a hybrid of both.
  - “Hard steps” of proofs (the ones requiring deep insight) are provided by human.
  - “Easy steps” of proofs can be filled in automatically.

(above bullet points taken from UPenn’s Software Foundations course slides)
Automated support for **derIVING PROOFS** and **CHECKING PROOFS** of language properties.

Implementation of the LF calculus (calculus for reasoning about deductive systems).

Alternatives: Coq, Isabelle, Agda, etc.

Presenting Example Twelf Encoding of Minilang.
Twelf is a **constructive** (not classical) proof assistant.

- Proposition is true iff there exists a proof of it.
- Law of excluded middle not assumed: $P \lor \neg P$.
  - Proving $P \lor \neg P$ **requires** either:
    - Proof of $P$ OR Proof of $\neg P$. 

ln $x = u$ such that $x = e^u$.

Definition in Isabelle/HOL:

- Proofs are programs.
Twelf is a **constructive** (not classical) proof assistant.

- Proposition is true iff there exists a proof of it.
- Law of excluded middle not assumed: $P \lor \neg P$.
  - Proving $P \lor \neg P$ requires either:
    - Proof of $P\ OR$ Proof of $\neg P$.
  - No case-split on undecidable propositions. Not allowed:
    - If $\text{halts}(\text{TuringMachine})$ then proof of $A$ else proof of $B$. 

In Twelf: Writing a proof = Writing a program.

**Proofs are programs.**
Twelf is a constructive (not classical) proof assistant.

Proposition is true iff there exists a proof of it.

Law of excluded middle not assumed: \( P \lor \neg P \).
  
  Proving \( P \lor \neg P \) requires either:
  
  - Proof of \( P \) OR Proof of \( \neg P \).
  
  No case-split on undecidable propositions. Not allowed:
  
  - If \( \text{halts}(TuringMachine) \) then proof of \( A \) else proof of \( B \).

No choice operator (\( \epsilon x. P(x) \) proposed by David Hilbert).

- \( \ln(x) = u \) such that \( x = e^u \).

- Definition in Isabelle/HOL:
  
  \[
  \text{definition ln :: real} \Rightarrow \text{real where}
  \ln x = \text{THE } u. \text{exp } u = x.
  \]
Twelf is a **constructive** (not classical) proof assistant.

Proposition is true **iff** there exists a proof of it.

Law of excluded middle not assumed: \( P \lor \neg P \).

- Proving \( P \lor \neg P \) **requires** either:
  - Proof of \( P \) OR Proof of \( \neg P \).
  - No case-split on undecidable propositions. Not allowed:
    - If \( \text{halts}(\text{TuringMachine}) \) then proof of \( A \) else proof of \( B \).

- No choice operator (\( \epsilon x. P(x) \) proposed by David Hilbert).
  - \( \ln(x) = u \) such that \( x = e^u \).
  - Definition in Isabelle/HOL:
    - definition \( \text{ln} :: \text{real} \Rightarrow \text{real} \) where
      \[
      \ln x = \text{THE} \ u. \ \exp u = x.
      \]

In Twelf: Writing a proof = Writing a program.

- Proofs are programs.
Lecture will involve in-class exercises.
Can try Twelf without installation.
Twelf Live Server:

http://twelf.org/live/

Links to starter code of examples will be provided.
Three **levels** of objects in Twelf:

- **Kinds** are at highest level.
- **Types** are at second level.
- **Terms** are at lowest level.

Each type is of a certain kind.

(Off-cell syntax: "someType : someKind")

Each term is of a certain type.

(Off-cell syntax: "someTerm : someType")

Twelf overloads languages constructs with same syntax.

(elegant but confusing too)

Contrived examples:

- **Term** [1, 2, 3] is of type **ArrayInt**.
- **Type** ArrayInt is of kind **Array**.

The kind type is a pre-defined kind in Twelf.
Three **levels** of objects in Twelf:

- **Kinds** are at highest level.
- **Types** are at second level.
- **Terms** are at lowest level.

Each type is of a certain kind.
(Twelf syntax: “someType : someKind”)

Each term is of a certain type.
(Twelf syntax: “someTerm : someType”)

Twelf overloads languages constructs with same syntax.
(elegant but confusing too)
Three **levels** of objects in Twelf:
- **Kinds** are at highest level.
- **Types** are at second level.
- **Terms** are at lowest level.

Each type is of a certain kind.
(Twelf syntax: “someType : someKind”)

Each term is of a certain type.
(Twelf syntax: “someTerm : someType”)

Twelf overloads languages constructs with same syntax.
(elegant but confusing too)

Contrived examples:
- Term [1, 2, 3] is of type ArrayInt.
- Type ArrayInt is of kind Array.
Three **levels** of objects in Twelf:
- **Kinds** are at highest level.
- **Types** are at second level.
- **Terms** are at lowest level.

Each type is of a certain kind.
(Twelf syntax: “someType : someKind”)

Each term is of a certain type.
(Twelf syntax: “someTerm : someType”)

Twelf overloads languages constructs with same syntax.
(elegant but confusing too)

Contrived examples:
- Term [1, 2, 3] is of type ArrayInt.
- Type ArrayInt is of kind Array.

The kind **type** is a pre-defined kind in Twelf.
Twelf supports defining functions:

```plaintext
int : type. one : int.
plusOne : int -> int.
```

- `plusOne` is a **function term**.
- `plusOne` takes in a term of type `int` and returns a term of type `int`.
- The type of function term `plusOne` is `int -> int`.
- `(plusOne one)` has type `int`.

Functions taking in multiple arguments are represented using their curried form:

```plaintext
plus : int -> int -> int.
```

- `int -> int -> int` is curried form of `(int, int) -> int`.
- `int -> int -> int` = `int -> (int -> int)`.
- `(plus one)` has type `int -> int`.
- `(plus one one)` has type `int`.
Functions

- Twelf supports defining functions:
  
  \[
  \text{int} : \text{type. one : int.}
  \]
  
  \[
  \text{plusOne} : \text{int} \rightarrow \text{int.}
  \]
  
  - plusOne is a function term.
  - plusOne takes in a term of type \text{int} and returns a term of type \text{int}.
  - The type of function term plusOne is \text{int} \rightarrow \text{int}.
  - \((\text{plusOne one})\) has type \text{int}.

- Functions taking in multiple arguments are represented using their curried form:
  
  \[
  \text{plus: int} \rightarrow \text{int} \rightarrow \text{int}.
  \]
  
  - int \rightarrow int \rightarrow int is curried form of \((\text{int}, \text{int}) \rightarrow \text{int}\).
  - int \rightarrow int \rightarrow int = \text{int} \rightarrow (\text{int} \rightarrow \text{int}).
  - \((\text{plus one})\) has type \text{int} \rightarrow \text{int}.
  - \((\text{plus one one})\) has type \text{int}.
Recall that type is a kind (type of types).

Functions can also return types:

equalsOne : int -> type.

- equalsOne is a function term.
- equalsOne takes in a term of type int and returns a type of kind type.
- The type of function term equalsOne is int -> type.
- (equalsOne one) is a type of kind type.
Recall that \textit{type} is a \textbf{kind} (type of types).

Functions can also return \textbf{types}:

- \textbf{equalsOne} : \textit{int} \rightarrow \textit{type}.
  - \textbf{equalsOne} is a function \textbf{term}.
  - \textbf{equalsOne} takes in a term of type \textit{int} and returns a \textbf{type} of kind \textit{type}.
  - The type of function term \textbf{equalsOne} is \textit{int} \rightarrow \textit{type}.
  - \((\text{equalsOne one})\) is a \textbf{type} of kind \textit{type}.

- \textbf{oneIsOne} : (\text{equalsOne one}).
  - Defines a new \textbf{term} \textbf{oneIsOne} of \textbf{type} \((\text{equalsOne one})\).
Functions Returning Types

- Recall that type is a kind (type of types).
- Functions can also return types:
  - \text{equalsOne} : \text{int} \rightarrow \text{type}.
    - \text{equalsOne} is a function term.
    - \text{equalsOne} takes in a term of type int and returns a type of kind type.
    - The type of function term \text{equalsOne} is \text{int} \rightarrow \text{type}.
    - \((\text{equalsOne} \ one)\) is a type of kind type.
- \text{oneIsOne} : (\text{equalsOne} \ one).
  - Defines a new term \text{oneIsOne} of type \text{equalsOne} \ one).
- A function type is a kind if its return type is also a kind.
  - \text{int} \rightarrow \text{type} is a kind.
  - \text{int} \rightarrow (\text{int} \rightarrow \text{type}) is a kind.
  - \text{int} \rightarrow \text{int} \rightarrow \text{type} = \text{int} \rightarrow (\text{int} \rightarrow \text{type}) is a kind.
- type is not allowed on the left-hand side of arrow (\rightarrow).
The object language is Minilang (the object of study).

Syntactic categories encoded w/ object types (defined types).

- `exp : type.`
- Defines type `exp` of kind `type`.
- `exp` represents syntactic category `e`.
- Terms in the grammar of `e` have type `exp`. 
The object language is Minilang (the object of study).

Syntactic categories encoded w/ object types (defined types).

- \( \text{exp} : \text{type} \).
- Defines type \( \text{exp} \) of kind \( \text{type} \).
- \( \text{exp} \) represents syntactic category \( e \).
- Terms in the grammar of \( e \) have type \( \text{exp} \).

Grammar productions encoded w/ \textbf{functions} between syntactic categories.

- \( \text{add} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp} \).
- \( \text{Expression} \ e \ : \ := \ +(\text{e}_1;\ \text{e}_2) \)
- \( \text{add} \) takes in \textbf{two} arguments.
- \( \text{exp} \rightarrow \text{exp} \rightarrow \text{exp} \) is curried form of \( (\text{exp}, \ \text{exp}) \rightarrow \text{exp} \).
Abstract syntax from earlier slides is **first-order abstract syntax (FOAS)**.

- Each AST has form $o(t_1, t_2, \ldots, t_n)$, where $o$ is operator and $t_1, \ldots, t_n$ are ASTs. Example:
Abstract syntax from earlier slides is **first-order abstract syntax (FOAS)**.

- Each AST has form $o(t_1, t_2, \ldots, t_n)$, where $o$ is operator and $t_1, \ldots, t_n$ are ASTs. Example:
  - $+(\text{num}[3]; \text{num}[4])$
  - $\text{add} (\text{enat} 3) (\text{enat} 4)$
Terms w/ variables using Higher-Order Abstract Syntax (HOAS)

- Abstract syntax from earlier slides is **first-order abstract syntax (FOAS)**.
  - Each AST has form \( o(t_1, t_2, \ldots, t_n) \), where \( o \) is operator and \( t_1, \ldots, t_n \) are ASTs. Example:
    - \(+ (\text{num}[3]; \text{num}[4])\)
    - \(\text{add} \quad \text{(enat 3)} \quad \text{(enat 4)}\)

- ASTs in **Higher-Order Abstract Syntax (HOAS)**:
  - Each \( t_i \) in \( o(t_1, t_2, \ldots, t_n) \) has form:
    \[
    x_1, x_2, \ldots, x_k . t
    \]
  - \( t \) is a FO-AST.
  - Each \( x_j \) is a variable bound in \( t \).
  - \( k \geq 0; \) if \( k = 0 \), then no variable is declared.
First, let expression in FOAS:

\[ \text{let}(x; e_1; e_2) \]

"x.e_2" captures that x is bound in e_2.

HOAS lets us know where variables are being bound.

\[ \text{let}(3; x.+\(x; 4)) \equiv \text{let}(3; y.+\(y; 4)) \]

Two preceding terms above are \textit{alpha-equivalent}.
First, let expression in FOAS:

\[
\text{let}(x; \ e_1; \ e_2)
\]

let expression in HOAS:

\[
\text{let}(e_1; \ x.e_2)
\]

“\(x.e_2\)” captures that \(x\) is bound in \(e_2\).
HOAS encoding of let expression

- First, let expression in FOAS:

  \text{let}(x; e_1; e_2)

- let expression in HOAS:

  \text{let}(e_1; x.e_2)

- “x.e_2” captures that x is bound in e_2.

- HOAS lets us know where variables are being bound.

  \text{let}(3; x.+(x; 4)) \equiv \text{let}(3; y.+(y; 4))

- Two preceding terms above are \textit{alpha-equivalent}.
Functions are really terms with holes/unknowns: \((3 + \bullet)\).
Higher-Order Terms are Functions

- Functions are really terms with **holes**/unknowns: \((3 + \bullet)\).
- Holes are represented by **variables**.
- Holes filled in by **applying** (terms w/ holes)/functions.
- Holes **abstract** details.
Higher-Order Terms are Functions

- Functions are really terms with **holes**/unknowns: \((3 + \bullet)\).
- Holes are represented by **variables**.
- Holes filled in by applying (terms w/ holes)/functions.
- Holes **abstract** details.
- “\(\times \cdot e\)” represented by **lambda abstraction** “\(\lambda x : \tau. e\)”.
- Twelf’s syntax of “\(\lambda x : \tau. e\)” : “\([x : \tau] \ e\)”
Twelf type signature of `let`:

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

Example HOAS term in Twelf:

Concrete Syntax

```
let x = 1 + 2
in x + 3
```

Twelf HOAS

```
let (add 1 2)
(\[x:exp\] add x 3)
```

No need to define object (Minilang) variables.

LF variables remove need for object variables.

No need to define substitution (nor requisite theorems) as well.
let expression in Twelf HOAS

- Twelf type signature of \texttt{let}:

\[
\texttt{let : exp} \to \ (\texttt{exp} \to \texttt{exp}) \to \texttt{exp}.
\]

- Example HOAS term in Twelf:

\[
\begin{array}{c|c}
\text{Concrete Syntax} & \text{Twelf HOAS} \\
\hline
\texttt{let x = 1 + 2 in x + 3} & \texttt{let (add 1 2) ([x:exp] add x 3)} \\
\end{array}
\]
let expression in Twelf HOAS

- Twelf type signature of let:

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

- Example HOAS term in Twelf:

<table>
<thead>
<tr>
<th>Concrete Syntax</th>
<th>Twelf HOAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{let } x = 1 + 2 \text{ in } x + 3)</td>
<td>(\text{let } (\text{add } 1 2) (\sgn{[x:exp]} \text{add } x 3))</td>
</tr>
</tbody>
</table>

- No need to define object (Minilang) variables.
- **LF variables** remove need for object variables.
- No need to define substitution (nor requisite theorems) as well.
Predicates in Twelf

- Predicates defined with **functions returning types (not terms)**.
  - Also, called **type families**.

- Typing Predicate: $e : \tau$

- Twelf Encoding: $\text{of} : \text{exp} \to \text{typ} \to \text{type}$.
Predicates in Twelf

- Predicates defined with functions returning types (not terms).
  - Also, called type families.
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \exp \rightarrow \typ \rightarrow \text{type} \).
- Type families return dependent types.
  - Types that contain terms (or depend on terms).
Predicates in Twelf

- Predicates defined with **functions returning types (not terms)**.
  - Also, called **type families**.
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \ exp \rightarrow \ typ \rightarrow \text{type} \).
- Type families return **dependent types**.
  - Types that contain terms (or depend on terms).
- Examples:
Predicates in Twelf

- Predicates defined with functions returning types (not terms).
  - Also, called type families.
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: `of : exp -> typ -> type`.
- Type families return dependent types.
  - Types that contain terms (or depend on terms).
- Examples:
  - 3 is a term of type `nat`.
Predicates in Twelf

- Predicates defined with **functions returning types (not terms)**.
  - Also, called **type families**.
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \text{exp} \to \text{typ} \to \text{type} \).
- Type families return **dependent types**.
  - Types that contain terms (or **depend on** terms).
- Examples:
  - 3 is a **term** of type \( \text{nat} \).
  - \( \text{Vec}(n) \) is a **type family** returning types of \( n \)-dimensional vectors.
Predicates in Twelf

- Predicates defined with **functions returning types (not terms)**.
  - Also, called **type families**.

- Typing Predicate: \( e : \tau \)

- Twelf Encoding: \( \text{of} : \exp \rightarrow \typ \rightarrow \text{type} \).

- Type families return **dependent types**.
  - Types that contain terms (or depend on terms).

- Examples:
  - \( 3 \) is a **term** of type **nat**.
  - \( \text{Vec}(n) \) is a **type family** returning types of \( n \)-dimensional vectors.
  - \( \text{Vec}(3) \) is a **dependent type** representing 3-dimensional vectors.
Predicates in Twelf

- Predicates defined with **functions returning types (not terms)**.
  - Also, called **type families**.
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \exp \rightarrow \typ \rightarrow \text{type}. \)
- Type families return **dependent types**.
  - Types that contain terms (or depend on terms).
- Examples:
  - \( 3 \) is a **term** of type \( \text{nat} \).
  - \( \text{Vec}(n) \) is a **type family** returning types of \( n \)-dimensional vectors.
  - \( \text{Vec}(3) \) is a **dependent type** representing 3-dimensional vectors.
  - \( [4, 1, 3] \) is a **term** of type \( \text{Vec}(3) \).
Judgments are Dependent Types

- Judgments/Propositions (instances of predicates) represented by dependent types.
- Judgment $z : \text{num}$ represented by type $(\text{of} \ (\text{enat} \ z) \ \text{num})$.
- Dependent type $(\text{of} \ e \ \tau)$ represents judgment “$e : \tau$”.

John Altidor  
Twelf Tutorial  
17/66
Judgments are Dependent Types

- Judgments/Propositions (instances of predicates) represented by dependent types.
- Judgment \( z : \text{num} \) represented by type \((\text{of} (\text{enat} z) \text{num})\).
- Dependent type \((\text{of} e \ \tau)\) represents judgment “\( e : \tau \)”.
- **Derivation/Proof** of “\( e : \tau \)” represented by **term** of type \((\text{of} e \ \tau)\).
Judgments are Dependent Types

- Judgments/Propositions (instances of predicates) represented by dependent types.
- Judgment $z : \text{num}$ represented by type $(\text{of (enat z) num})$.
- Dependent type $(\text{of } e \tau)$ represents judgment “$e : \tau$”.
- Derivation/Proof of “$e : \tau$” represented by term of type $(\text{of } e \tau)$.
- Curry-Howard Correspondence: Proofs are terms. Propositions/Judgments are types.
Pi-Abstractions

- **Function/Lambda Abstraction** “\( \lambda x : S. e \)” of type \( S \rightarrow T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( T \).
Pi-Abstractions

- **Function/Lambda Abstraction** “\( \lambda x : S . e \)” of type \( S \rightarrow T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( T \).

- **Function/Pi-abstraction** of type pi-type \( \Pi x : S . T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \([s/x]T\).
Pi-Abstractions

- **Function/Lambda Abstraction** “\( \lambda x : S.e \)” of type \( S \to T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( T \).

- **Function/Pi-abstraction** of type pi-type \( \Pi x : S.T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( [s/x]T \).

- If \( x \not\in \text{fv}(T) \), then \( \Pi x : S.T \equiv S \to T \).
Pi-Abstractions

- **Function/Lambda Abstraction** “\( \lambda x : S. e \)” of type \( S \rightarrow T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( T \).

- **Function/Pi-abstraction** of type pi-type \( \Pi x : S. T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( [s/x]T \).

- If \( x \notin \text{fv}(T) \), then \( \Pi x : S. T \equiv S \rightarrow T \).
- If \( x \in \text{fv}(T) \), then \( \Pi x : S. T \) returns term of a dependent type.
Pi-Abstractions

- **Function/Lambda Abstraction** \( \lambda x : S. e \) of type \( S \rightarrow T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( T \).
- **Function/Pi-abstraction** of type pi-type \( \Pi x : S. T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \([s/x]T\).
- If \( x \not\in \text{fv}(T) \), then \( \Pi x : S. T \equiv S \rightarrow T \).
- If \( x \in \text{fv}(T) \), then \( \Pi x : S. T \) returns term of a dependent type.
- Twelf Syntax for \( S \rightarrow T \):
  \( S \to T \)
Pi-Abstractions

- **Function/Lambda Abstraction** “\( \lambda x : S. e \)” of type \( S \rightarrow T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( T \).

- **Function/Pi-abstraction** of type pi-type \( \Pi x : S. T \):
  - Takes in a term \( s \) of type \( S \).
  - Returns a term of type \( [s/x]T \).

- If \( x \notin \text{fv}(T) \), then \( \Pi x : S. T \equiv S \rightarrow T \).

- If \( x \in \text{fv}(T) \), then \( \Pi x : S. T \) returns term of a dependent type.

- Twelf Syntax for \( S \rightarrow T \):
  \[
  S \rightarrow T
  \]

- Twelf Syntax for \( \Pi x : S. T \):
  \[
  \{x:S\} T
  \]
Inference Rules are Functions

\[
\begin{align*}
\text{num}[n] &: \text{num} \\
\text{T.1}
\end{align*}
\]

- \text{of/nat} : \{N: \text{nat}\} \text{ of (enat N) num}.
- Twelf Convention:
  - Constants start with lower-case letters.
  - Variables/parameters start with upper-case letters.
Inference Rules are Functions

\[
\text{num}[n] : \text{num} \quad \text{T.1}
\]

- \text{of/nat} : \{N:\text{nat}\} \text{ of \ (enat N) num.}
- Twelf Convention:
  - Constants start with lower-case letters.
  - Variables/parameters start with upper-case letters.
- \((\text{of/nat } z) \neq (\text{of (enat } z \text{) num})\).
- \text{of/nat returns terms} not types.
Inference Rules are Functions

\[
\begin{array}{c}
\text{num}[n] : \text{num} \\
\text{T.1}
\end{array}
\]

- **of/nat**: \(\{N:\text{nat}\} \text{ of (enat N) num}\).
- **Twelf Convention**:
  - Constants start with lower-case letters.
  - Variables/parameters start with upper-case letters.
- \((\text{of/nat z}) \neq (\text{of (enat z) num})\).
- **of/nat** **returns** **terms** **not** **types**.
- \((\text{of/nat z}) = \text{term} \text{ of type (of (enat z) num)}\).
Inference Rules are Functions

\[ \text{num}[n] : \text{num} \quad \text{T.1} \]

- **of/nat**: \{N:nat\} of (enat N) num.
- **Twelf Convention**:
  - Constants start with lower-case letters.
  - Variables/parameters start with upper-case letters.
- \((\text{of/nat } z) \neq (\text{of } (\text{enat } z) \text{ num}).\)
- **of/nat** returns **terms** not types.
- \((\text{of/nat } z) = \text{term} \text{ of type } (\text{of } (\text{enat } z) \text{ num}).\)
- \((\text{of/nat } z)\) is a **derivation/term** of judgment \(z : \text{num}\) represented by type \(\text{of } (\text{enat } z) \text{ num}\).
Premises are Inputs

\[
\frac{e_1: \text{num} \quad e_2: \text{num}}{+\langle e_1; e_2 \rangle: \text{num}} \quad \text{T.4}
\]

- **Twelf Encoding:**
  \[
  \text{of/add} : \text{of (add E1 E2) num}
  \]
  \[
  \quad \leftarrow \text{of E1 num}
  \]
  \[
  \quad \leftarrow \text{of E2 num}.
  \]

- **Given a proof of** (of E2 num) **and**
- **Given a proof of** (of E1 num)
- **of/add returns proof of** (of (add E1 E2) num)
Implicit and explicit parameters

- Parameter N is **explicit** in the above signature.
- Explicit parameters **must be specified** in function applications.
- D : of (enat z) num = of/nat z.
Implicit and explicit parameters

- Parameter N is **explicit** in the above signature.
- Explicit parameters **must be specified** in function applications.

D : of (enat z) num = of/nat z.

- of/nat:  of (enat N) num.
- Parameter N is **implicit** in the above signature.
- Implicit parameters **cannot be specified** by programmer in function applications.

D : of (enat z) num = of/nat.

- Twelf figures out from the context that z is the implicit parameter that of/nat should be applied to.
Can only quantify over first-order terms.

**Allowed:**

- \( \text{add} : \exp \rightarrow \exp \rightarrow \exp. \)
- \( \text{let} : \exp \rightarrow (\exp \rightarrow \exp) \rightarrow \exp. \)

- A **higher-order term** is a function, where one of its inputs is also a function.
First-Order Quantification Only

- Can only quantify over **first-order** terms.

- **Allowed:**
  - add : exp -> exp -> exp.
  - let : exp -> (exp -> exp) -> exp.
    - A **higher-order term** is a function, where one of its inputs is also a function.

- **Not allowed:**
  - quantifyTypes : exp -> type -> exp.
  - allIsTrue : {Prop: type} Prop.

- The kind **type** categorizes Twelf types.
Can only quantify over **first-order** terms.

**Allowed:**
- `add : exp -> exp -> exp.`
- `let : exp -> (exp -> exp) -> exp.`
  - A **higher-order term** is a function, where one of its inputs is also a function.

**Not allowed:**
- `quantifyTypes : exp -> type -> exp.`
- `allIsTrue : {Prop:type} Prop.`

The kind `type` categorizes Twelf types.

No type polymorphism implies no **general** logical connectives.

**Not allowed:**
- `conjunction : {P:type} {Q:type} P -> Q -> (and P Q).`
Different term levels used to restrict quantification.

- Twelf terms are first-order terms; e.g., \((s \ z)\).
- Twelf types are second-order terms; e.g., \(\text{nat}\).
- Twelf kinds are third-order terms; e.g., \(\text{type}\).
Predicativity

- Different term levels used to restrict quantification.
  - Twelf terms are first-order terms; e.g., \((s \ z)\).
  - Twelf types are second-order terms; e.g., \(\text{nat}\).
  - Twelf kinds are third-order terms; e.g., \(\text{type}\).

- Twelf only allows **predicative** definitions:
  - Cannot apply term to itself. (Cannot quantify over oneself.)
  - No term has itself as type. (Not allowed: \(\text{typ} : \text{typ}\).)
  - Disallows Russell’s paradox:
    Let \(H = \{x \mid x \notin x\}\). Then \(H \in H \iff H \notin H\).

- Helps Twelf avoid logical inconsistency
  (i.e. proving false/uninhabited type).

- False implies any proposition (including false ones).

- False/uninhabited types used for constructive proofs by contradiction.
Create language of numbers with subtyping in Twelf.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terms</td>
<td>e</td>
<td>zero</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pi</td>
<td>(\pi)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>img</td>
<td>(\sqrt{-1})</td>
</tr>
<tr>
<td>Types</td>
<td>t</td>
<td>number</td>
<td>num</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complex</td>
<td>complex</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>
**Subtyping** Rules (not all):

- complex <: num
- real <: num
- int <: real

**Typing** Rules (not all):

- 0 : int
- $\pi$ : real
- $\sqrt{-1} : \text{complex}$

Define **reflexive** and **transitive** rules for subtyping.

Define **subsumption** rule for typing judgment.

**Prove** 0 : num.

▶ Fill in the blank below:

▶ D : (of zero number) = •
What happened to typing context $\Gamma$?
Hypothetical Judgments in Twelf

- What happened to typing context $\Gamma$?
- **Hypothetical Judgments**: Judgments made under the assumption of other judgments.
What happened to typing context $\Gamma$?

**Hypothetical Judgments:**
Judgments made under the assumption of other judgments.

Encoded w/ **higher-order types**:
Function types where one of the inputs is also a function type.
What happened to typing context $\Gamma$?

**Hypothetical Judgments:**
Judgments made under the assumption of other judgments.

Encoded w/ **higher-order types**:
Function types where one of the inputs is also a function type.

Input function types represent hypothetical assumptions.

Similar to higher-order terms.
(Another application of HOAS)

$\Gamma$ does not need to be defined.
Typing let expression in Twelf HOAS

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash \text{let}(x; e_1; e_2) : \tau_2 \quad \text{T.6} \]

- Twelf Encoding:
  \[
  \text{of/let} : \{ x : \text{exp} \} \text{ of } x \text{ } T_1 \rightarrow \text{ of } (E_2 \text{ } x) \text{ } T_2 \rightarrow \text{ of } E_1 \text{ } T_1 \rightarrow \text{ of } (\text{let } E_1 ([x] E_2 \text{ } x)) \text{ } T_2.
  \]
Typing let expression in Twelf HOAS

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash \text{let}(x; e_1; e_2) : \tau_2 \quad \text{T.6} \]

- Twelf Encoding:
  of/let : (\{x: exp\} of x T1 -> of (E2 x) T2) ->
  of E1 T1 ->
  of (let E1 ([x] E2 x)) T2.

- First, a Twelf **coding convention**:
  Return type (of (let E1 ([x] E2 x)) T2) could be replaced with (of (let E1 E2) T2).

- E2 in both cases is of type (exp -> exp).

- ([x] E2 x) used for readability: # of inputs explicit.

- ([x] E2 x) is called the **eta-expansion** of E2.
Let $f$ be a function of type

$$\left\{ x : \text{exp} \right\} \text{of} \ x \ T_1 \rightarrow \ \text{of} \ (\text{E2} \ x) \ T_2.$$

Reminder of hypothetical judgment: $\Gamma, x : \tau_1 \vdash e_2 : \tau_2.$
Let $f$ be a function of type

$$(\{x: \text{exp}\} \text{ of } x \text{ T1} \rightarrow \text{of } (E2 \ x) \text{ T2}).$$

Reminder of hypothetical judgment: $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.

- $f$ takes in an exp term bound to LF variable $x$.
- $f$ takes in a term $dx$ of type $(\text{of } x \text{ T1})$.
- $f$ returns a term of type $(\text{of } (E2 \ x) \text{ T2})$. 

The ability to use a proof ($dx$) of type $(\text{of } x \text{ T1})$ simulates extending typing context with $x : \tau_1$. 

John Altidor  Twelf Tutorial  28/ 66
Let $f$ be a function of type
$$(\{x: \text{exp}\} \mathsf{of} \ x \ T1 \rightarrow \mathsf{of} \ (E2 \ x) \ T2).$$

Reminder of hypothetical judgment: $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.

- $f$ takes in an exp term bound to LF variable $x$.
- $f$ takes in a term $dx$ of type $\mathsf{of} \ x \ T1$.
- $f$ returns a term of type $\mathsf{of} \ (E2 \ x) \ T2$.
- $dx$ of type $\mathsf{of} \ x \ T1$ can be used in the body of $f$ to return a proof/term of type $\mathsf{of} \ (E2 \ x) \ T2$.
- The ability to use a proof ($dx$) of type $\mathsf{of} \ x \ T1$ simulates extending typing context with $x : \tau_1$. 
Derive the judgment $\vdash \text{let } x \text{ be } 1 \text{ in } x + 0 : \text{num}$ in Twelf.

Twelf encoding of above judgment:
\[
of (\text{let} \ (\text{enat} \ (s \ z)) \ ([x:exp] \ \text{add} \ x \ (\text{enat} \ z))) \ \text{num}.
\]

Recall important signatures (displaying implicit parameters):
\[
of/\text{let} : \{T1:typ\} \ \{E2:exp \to exp\} \ \{T2:typ\} \ \{E1:exp\}
\quad (\{x:exp\} \ of \ x \ T1 \to of \ (E2 \ x) \ T2) \to of \ E1 \ T1
\quad \to of \ (\text{let} \ E1 \ ([x:exp] \ E2 \ x)) \ T2.
\]
\[
of/\text{nat} : \{N:nat\} \ of \ (\text{enat} \ N) \ \text{num}.
\]
\[
of/\text{add} : \{E2:exp\} \ \{E1:exp\}
\quad of \ E2 \ \text{num} \to of \ E1 \ \text{num} \to of \ (\text{add} \ E1 \ E2) \ \text{num}.
\]
- Inputs/Outputs defined with \texttt{%mode} declaration.
  \begin{verbatim}
  of : exp -> typ -> type.
  \texttt{%mode of +E -T}.
  \end{verbatim}
- Inputs marked with +.
- Outputs marked with -.
- Inputs/Outputs defined with %mode declaration.
  of : exp -> typ -> type.
  %mode of +E -T.
- Inputs marked with +.
- Outputs marked with -.
- Outputs can be derived automatically using Twelf’s logic programming engine (example later).
- Inputs/Outputs defined with `%mode` declaration. Of: \( \text{exp} \rightarrow \text{typ} \rightarrow \text{type} \).
  `%mode` of +E -T.
- Inputs marked with +.
- Outputs marked with -. Outputs can be derived automatically using Twelf’s logic programming engine (example later).
- Not all relations required modes.
- Modes are necessary for specifying theorems.
- Modes used also for checking proofs of theorems.
Inputs/Outputs defined with \%mode declaration.
of : exp -> typ -> type.
\%mode of +E -T.

Inputs marked with +.

Outputs marked with -.

Outputs can be derived automatically using Twelf’s logic programming engine (example later).

Not all relations required modes.

Modes are necessary for specifying theorems.

Modes used also for checking proofs of theorems.

Only ground terms may be applied to relations w/ modes in rules (details later).
Backward Arrow vs. Forward Arrow

- Output terms must be **ground** given ground input terms.
  - Ground terms do not contain **free** variables.
  - Output terms are fixed (ground) wrt (ground) inputs.
- Forward “->” reflects **order that premises are passed to rules/functions** and makes proofs more natural.
- Backward “<-” reflects **order of resolving ground terms**.
Backward Arrow vs. Forward Arrow

- Output terms must be **ground** given ground input terms.
  - Ground terms do not contain **free** variables.
  - Output terms are fixed (ground) wrt (ground) inputs.
- Forward “→” reflects **order that premises are passed to rules/functions** and makes proofs more natural.
- Backward “←” reflects **order of resolving ground terms**.
- **Order of args** allowed by Twelf:
  
  \[
  \text{of/let} : (\{x: \text{exp}\} \text{of } x \ T1 \rightarrow \text{of } (E2 \ x) \ T2) \rightarrow \\
  \text{of } E1 \ T1 \rightarrow \\
  \text{of } (\text{let } E1 ([x] E2 \ x)) \ T2.
  \]

John Altidor Twelf Tutorial 31/66
Backward Arrow vs. Forward Arrow

- Output terms must be **ground** given ground input terms.
  - Ground terms do not contain **free** variables.
  - Output terms are fixed (ground) wrt (ground) inputs.
- Forward “−>” reflects **order that premises are passed to rules/functions** and makes proofs more natural.
- Backward “<-” reflects **order of resolving ground terms**.
- **Order of args** allowed by Twelf:
  
  $\text{of/let} : \ (\{x: \ exp\} \ \text{of} \ x \ T1 \ \text{->} \ \text{of} \ (E2 \ x) \ T2) \ \text{->}
  
  of \ E1 \ T1 \ \text{->}
  
  \text{of} \ (\text{let} \ E1 \ ([x] \ E2 \ x)) \ T2.$

- **Order of args that causes error**:
  
  $\text{of/let} : \ \text{of} \ E1 \ T1 \ \text{->}
  
  (\{x: \ exp\} \ \text{of} \ x \ T1 \ \text{->} \ \text{of} \ (E2 \ x) \ T2) \ \text{->}
  
  \text{of} \ (\text{let} \ E1 \ ([x] \ E2 \ x)) \ T2.$
Backward Arrow vs. Forward Arrow

- Output terms must be **ground** given ground input terms.
  - Ground terms do not contain **free** variables.
  - Output terms are fixed (ground) wrt (ground) inputs.
- Forward “→” reflects **order that premises are passed to rules/functions** and makes proofs more natural.
- Backward “←” reflects **order of resolving ground terms**.
- **Order of args** allowed by Twelf:
  
  of/let : ({x: exp} of x T1 → of (E2 x) T2) →
  of E1 T1 →
  of (let E1 ([x] E2 x)) T2.

- **Order of args that causes error**:
  of/let : of E1 T1 →
  ({x: exp} of x T1 → of (E2 x) T2) →
  of (let E1 ([x] E2 x)) T2.

- **Error message**:
  Occurrence of variable T1 in output (−) argument not necessarily ground
Terms in **input positions** of **return type** are **universally-quantified** inputs to function.

right1 : of E₁ num → of (add E₁ (enat z)) num.
Terms in input positions of return type are universally-quantified inputs to function.

right1 : of E1 num -> of (add E1 (enat z)) num.

Terms in input position of return type: (add E1 (enat z)).
Terms in **input positions** of **return type** are **universally-quantified** inputs to function.

```
right1 : of E1 num -> of (add E1 (enat z)) num.
```

Terms in input position of return type: 

```
(add E1 (enat z)).
```

Tokens starting with capital letters are assumed by Twelf to be variables in type: \( E1 \).
Terms in **input positions** of **return type** are **universally-quantified** inputs to function.

right1 : of E1 num -> of (add E1 (enat z)) num.

Terms in input position of return type: (add E1 (enat z)).

Tokens starting with capital letters are assumed by Twelf to be variables in type: E1.

Free variables in input position of return type, E1, are inferred by Twelf to be **universally**-quantified inputs to function right1.

- Only these terms are allowed to be universal inputs to function right1.
Resolving Ground Terms

- All terms must be **ground** terms: constants or terms **without** free variables assuming that input terms (from return type) are also ground (do not contain free variables).

- Next Step:
  Check that input terms in type preceding return type are ground:

- \texttt{right1 : of E1 num -> of (add E1 (enat z)) num.}
Resolving Ground Terms

- All terms must be **ground** terms: constants or terms **without** free variables assuming that input terms (**from return type**) are also ground (do not contain free variables).

- Next Step:
  Check that input terms in type preceding return type are ground:

- right1 : of E1 num -> of (add E1 (enat z)) num.
Resolving Ground Terms

All terms must be **ground** terms: constants or terms **without** free variables assuming that input terms (**from return type**) are also ground (do not contain free variables).

Next Step:
Check that input terms in type preceding return type are ground:

- right1 : of \( E1 \) num \( ightarrow \) of (add \( E1 \) (enat z)) num.

- \( E1 \) in premise type (of \( E1 \) num) is ground wrt \( E1 \) in return type because they are the same.
Resolving Ground Terms

- All terms must be **ground** terms: constants or terms **without** free variables assuming that input terms (**from return type**) are also ground (do not contain free variables).

- Next Step:
  Check that input terms in type preceding return type are ground:

  - right1 : of E1 num -> of (add E1 (enat z)) num.

  - **E1** in premise type (of E1 num) is ground wrt **E1** in return type because they are the same.

  - **num** in premise type (of E1 num) is ground wrt return type because **num** is a constant.
Non-ground Term in Premise Causing Error

- `wrong1 : of E2 num -> of (add E1 (enat z)) num.`
- **E2** term not coming from conclusion (return type).
Output terms resulting from grounded input terms are also ground.

Second argument of the `of` relation is an **output** argument.

\[
\text{right2} : \quad \text{of} \ E \ T \rightarrow \ \text{of} \ (\text{add} \ E \ (\text{enat} \ z)) \ T.
\]
Output terms resulting from grounded input terms are also
ground.

Second argument of the of relation is an output argument.

\[ \text{right2} : \text{of } E \text{ T} \rightarrow \text{of } (\text{add } E (\text{enat } z)) \text{ T.} \]
Output terms resulting from grounded input terms are also ground.

Second argument of the of relation is an output argument.

right2 : of E T -> of (add E (enat z)) T.
Ground Term From Output

Output terms resulting from grounded input terms are also ground.

Second argument of the of relation is an output argument.

\[
\text{right2} : \text{of } E \ T \rightarrow \text{of } (\text{add } E \ (\text{enat } z)) \ T.
\]

Term \( T \) is \textbf{computed}/result of premise/\textbf{recursive call} (of \( E \ T \)).
Output terms resulting from grounded input terms are also ground.

Second argument of the *of* relation is an *output* argument.

\[
\text{right2} : \text{of } E \ T \rightarrow \text{of } (\text{add } E \text{ (enat } z)) \ T.
\]

Term \( T \) is *computed*/result of premise/recursive call (of \( E \ T \)).
Output term in conclusion not grounded:

\[ \text{wrong2} : \text{of } E \ T_1 \rightarrow \text{of } (\text{add } E \ (\text{enat } z)) \ T_2. \]

Output term \( T_2 \) is universally quantified instead of a grounded result of the input term. This violates the %mode declaration of the of relation.
- right1: of E1 num -> of (add E1 (enat z)) num.
- wrong1: of E2 num -> of (add E1 (enat z)) num.
- right2: of E T -> of (add E (enat z)) T.
- wrong2: of E T1 -> of (add E (enat z)) T2.
Decidable Predicate Definition or Algorithmic Definition: Definition of predicate that gives an algorithm for deciding predicate that halts on all inputs within a finite number of steps.

Constructive Logic Requirement: Proposition is true iff there exists a proof of it.
Decidable Predicate Definitions

- Decidable Predicate Definition or Algorithmic Definition: Definition of predicate that gives an algorithm for deciding predicate that halts on all inputs within a finite number of steps.

- Constructive Logic Requirement: Proposition is true iff there exists a proof of it.

- For every true proposition/instance of predicate, algorithm finds a proof of proposition.

- For every false proposition of predicate, algorithm determines no proof exists.
Termination

- `%terminates` checks a program succeeds or fails in a finite number of steps given ground inputs.
- Modes with termination ensure decidable definitions.
- Termination not guaranteed with transitive rule.

```
subtype : typ -> typ -> type.
%mode subtype +T1 -T2.
```

```
subtype/int/rea : subtype int real.
subtype/rea/num : subtype real number.
subtype/num/num : subtype number number.
subtype/trans:
  subtype T1 T3 <- subtype T1 T2 <- subtype T2 T3.
%terminates T (subtype T _).
```

- **Error**: Termination violation:  ---> (T1) < (T1)
- First input to subtype not **smaller** in premise/recursive call.
Syntax-Directed Definition: For each syntactic form of input, there is at most one applicable rule.

Syntax of input term tells us which rule to use. (or if no rule applies)

Each true proposition of a syntax-directed predicate has exactly one unique derivation.

Only one way to derive \(+(5; 3) : \text{num}.\)

\[
\begin{align*}
5 : \text{num} & \quad \text{of/num} \\
3 : \text{num} & \quad \text{of/num} \\
+(5; 3) : \text{num} & \quad \text{of/add}
\end{align*}
\]

No need for exhaustive proof search with syntax-directed predicates.
%unique checks if outputs are uniquely determined by inputs.
%unique check can also ensure rules are syntax-directed.

subtype : typ -> typ -> type.
%mode subtype +T1 -T2.
subtype/int/rea : subtype int real.
subtype/rea/num : subtype real number.
subtype/num/num : subtype number number.
subtype/trans:
    subtype T1 T3 <- subtype T1 T2 <- subtype T2 T3.
%worlds () (subtype _ _).
%unique subtype +T1 +T2.

■ Error: subtype/rea/num and subtype/trans overlap
■ Both rules could be used to derive subtype real number.
Twelf can derive (search) for proofs:

- \%solve D1 :
  of (estr (a , b , c , a , eps)) string.

- Twelf will save proof term in D1.
To print all (implicit) terms in proofs:

- From Twelf Server:
  "set Print.implicit true"

- From ML (SML) Prompt:
  "Twelf.Print.implicit := true"

- Then just execute "Check File":
  Emacs Key Sequence: ^C ^S
loadFile test_typing.elf
[Opening file test_typing.elf]
%solve
of (estr (, a (, b (, c (, a eps)))))) string.
OK
D1 : of (estr (, a (, b (, c (, a eps)))))) string
    = of/str (, a (, b (, c (, a eps)))))).
Preservation Theorem:
If \((\text{of } E \ T)\) and \((\text{step } E \ E')\), then \((\text{of } E' \ T)\).
Twelf Theorems

- **Preservation Theorem:**
  If (of E T) and (step E E’), then (of E’ T).

- Twelf allows expressing $\forall\exists$-type properties.

- **Preservation, re-formulated:**
  - For every derivation of (of E T) and (step E E’),
  - there exists at least one derivation of (of E’ T).
**Preservation Theorem:**
If \((\text{of } E \ T)\) and \((\text{step } E \ E')\), then \((\text{of } E' \ T)\).

Twelf allows expressing \(\forall \exists\)-type properties.

**Preservation, re-formulated:**
- For every derivation of \((\text{of } E \ T)\) and \((\text{step } E \ E')\),
- there exists at least one derivation of \((\text{of } E' \ T)\).

\[
\text{%theorem}
\text{preservation :}
\begin{align*}
  & \forall \forall* \{E\} \{E'\} \{T\} \\
  & \forall \{O: \text{of } E \ T\} \{S: \text{step } E \ E'\} \\
  & \exists \{O': \text{of } E' \ T\} \\
  & \text{true.}
\end{align*}
\]

Verbose syntax above.
Desugared, concise alternative on next slide.
Preservation theorem is a function returning types (type family):

\[
preservation: \quad \text{of } E \ T \rightarrow \text{step } E \ E' \rightarrow \text{of } E' \ T \rightarrow \text{type}.
\]

Premises are **inputs**. Conclusions are **outputs**.

%mode preservation +O +S -O'.

To prove preservation theorem, need to show *preservation* is a **total relation** on all **possible inputs**.

- For each possible derivation of premises (inputs), need at least one derivation of conclusion (output).
Proofs of Theorems

- Proofs of theorems are **total** relations over inputs.
- Proving theorem
  = Constructing functions **for each case**:
    - For each **constructor** of term to perform structural induction on.
Proofs of Theorems

- Proofs of theorems are **total** relations over inputs.

- Proving theorem
  - Constructing functions **for each case**:
    - For each **constructor** of term to perform structural induction on.

- Note:
  - **No case-split** or **pattern match** construct in Twelf.
    - This is the reason why **multiple functions** are required to prove theorem for multiple cases.
    - Results in smaller proof terms but more of them.
Case: (T.4, D.1)

\[
\begin{align*}
  e_1 &: \text{num} & e_2 &: \text{num} & \Rightarrow & \quad e_1 \mapsto e'_1 \\
  + (e_1; e_2) &: \text{num} & \Rightarrow & \quad + (e_1; e_2) \mapsto + (e'_1; e_2)
\end{align*}
\]

We assume preservation holds for subexpressions. Hence, by the \textbf{inductive hypothesis}, \( e_1 : \text{num} \) and \( e_1 \mapsto e'_1 \) implies \( e'_1 : \text{num} \).

Rule T.4 gives us:

\[
\begin{align*}
  e'_1 &: \text{num} & e_2 &: \text{num} & \Rightarrow & \quad e'_1 \mapsto e'_1 \\
  + (e'_1; e_2) &: \text{num} & \Rightarrow & \quad + (e'_1; e_2) \mapsto + (e'_1; e_2)
\end{align*}
\]
of/add : 
  of (add E1 E2) num <- of E1 num <- of E2 num.

- :
{E1-num : of E1 num }
{E2-num : of E2 num }
{E1=>E1' : step E1 E1' }
{E1'–num : of E1’ num }
preservation E1-num E1=>E1’ E1’–num -> 
preservation
  ((of/add E2-num E1-num) : (of (add E1 E2) num))
  ((step/add1 E1=>E1’) :
    (step (add E1 E2) (add E1’ E2)))
  ((of/add E2–num E1’–num) : (of (add E1’ E2) num)).
Proof Case without Explicit Types

- : preservation
  (of/add E2-num E1-num)
  (step/add1 E1=>E1')
  (of/add E2-num E1'-num)
  <- preservation E1-num E1=>E1' E1'-num.

- Types of terms in proofs: usually not required to specify.
- Allowed to be manually specified.
- Output from Twelf server contains (some) inferred types.
Applying Inductive Hypothesis

- : preservation
  (of/add E2-num E1-num)
  (step/add1 E1=>E1')
  (of/add E2-num E1'-num)
  <- preservation E1-num E1=>E1' E1'-num.

- Applying inductive hypothesis = recursive call.
After proving all cases, ask Twelf to check we covered all cases.

%worlds () (preservation _ _ _).
%total E-T (preservation E-T _ _).

%total E-T tells Twelf to check proof of totality by structural induction on typing derivation E-T.

Details of %world declaration later.
Missing Case

- If we forget to prove a case, \%total command will fail.
- Twelf prints error message to help user “debug” proof:

```
preservation.elf:69.8-69.11 Error: Coverage error --- missing cases:
{E1:exp} {E2:exp} {E3:exp}
{01:of (add E1 E2) num} {S1:step E1 E3}
{02:of (add E3 E2) num}
|  |- preservation 01 (step/add1 S1) 02.
```

- Forgot the case where we could derive:
  - (of (add E1 E2) num)
  - (step (add E1 E2) (add E3 E2))

- Need to construct proof of (of (add E3 E2) num).
Assuming What Needs To Be Proven

- Cannot prove case by just assuming conclusion.
Assuming What Needs To Be Proven

- Cannot prove case by just assuming conclusion.
- Also, cannot assume propositions not derived from premises in proofs.
Assuming What Needs To Be Proven

- Cannot prove case by just assuming conclusion.
- Also, cannot assume propositions not derived from premises in proofs.
- Such a proof will contain a non-ground term.
  - %mode declarations used to check proofs.
Recall Valid Proof of Case

- : {E1-num : of E1 num}
  {E2-num : of E2 num}
  {E1=>E1' : step E1 E1'}
  {E1'-num : of E1' num}

preservation E1-num E1=>E1' E1'-num

-> preservation (of/add E2-num E1-num)
   (step/add1 E1=>E1')
   (of/add E2-num E1'-num).
Invalid Proof of Case

- : {E1-num : of E1 num }
{E2-num : of E2 num }
{E1=>E1’ : step E1 E1’ }
{E1’-num : of E1’ num }
preservation (of/add E2-num E1-num)
   (step/add1 E1=>E1’)
   (of/add E2-num E1’-num).

- Proof above just assumes of E1’ num, which is not one of the assumptions for the case.
- E1’-num is not part of input terms in return type.
- E1’-num is not an output term derived from ground terms.
- Twelf reports error for function above above.
To check totality of function/theorem, need to define all possible inputs or worlds.

- World = Set of terms of a type (inhabitants of a type)

Example world of natural numbers:

```plaintext
nat : type.
z : nat.
s : nat -> nat.
%worlds () (nat).
```
To check totality of function/theorem, need to define all possible inputs or worlds.

- World = Set of terms of a type (inhabitants of a type)

Example world of natural numbers:

```
nat : type.

z : nat.
s : nat -> nat.
%worlds () (nat).
```

- No term of type nat containing LF variables.
- No such nat of form \((s \ x)\), where \(x\) of variable of type nat.
Let expression contains binders.

add : exp → exp → exp.
let : exp → (exp → exp) → exp.
%worlds () (exp).
Let expression contains binders.

add : exp \rightarrow exp \rightarrow exp.
let : exp \rightarrow (exp \rightarrow exp) \rightarrow exp.
\%worlds () (exp).

Error message:
syntax.elf:38.15-38.25 Error:
While checking constant let:
World violation for family exp: {_:exp} <\:/: 1
Terms Containing Binders

- Let expression contains binders.
  
  add : exp -> exp -> exp.
  
  let : exp -> (exp -> exp) -> exp.
  
  %worlds () (exp).

- Error message:
  
  syntax.elf:38.15–38.25 Error:
  
  While checking constant let:
  
  World violation for family exp: {_:exp} </: 1

- Need to tell Twelf about possible variables that can arise from rules.
Blocks

Blocks: Patterns describing fragment of contexts.

Update addressing previous error:

```plaintext
add : exp -> exp -> exp.
let : exp -> (exp -> exp) -> exp.
%block exp-block : block {x:exp}.
%worlds (exp-block) (exp).
```

Informs Twelf that terms of type exp can contain binders of type exp.
Blocks: Patterns describing fragment of contexts.

Update addressing previous error:
add : exp -> exp -> exp.
let : exp -> (exp -> exp) -> exp.
%block exp-block : block {x:exp}.
%worlds (exp-block) (exp).

Informs Twelf that terms of type exp can contain binders of type exp.

Worlds can take in multiple blocks. Syntax:
%worlds (block1 | block2 | ... | blockN) (exp).
Specifying world requires specifying how variables are quantified (universal inputs or ground outputs).

\texttt{of : exp \rightarrow typ \rightarrow type.}
\texttt{\%mode of +E -T.}

\ldots

\texttt{of/let : of (let E1 ([x] E2 x)) T2}
\texttt{<- of E1 T1}
\texttt{<- (\{x: exp\} of x T1 \rightarrow of (E2 x) T2).}
\texttt{\%block of-block :}
\texttt{\hspace{1cm} some \{T:typ\} block \{x: exp\}\{_ : of x T\}.}
\texttt{\%worlds (of-block) (of \_ \_).}

Number of args specified by pattern in \texttt{\%worlds} declaration:
\texttt{(of \_ \_)}
After defining the worlds of all inputs to a theorem/function type, we can ask Twelf to check that the proof/function is total: **defined over the world**.

%worlds () (preservation _ _ _).
%total E-T (preservation E-T _ _).

%total E-T tells Twelf to check proof of totality by structural induction on typing derivation E-T.
After defining the worlds of all inputs to a theorem/function type, we can ask Twelf to check that the proof/function is total: defined over the world.

%worlds () (preservation _ _ _).
%total E-T (preservation E-T _ _).

%total E-T tells Twelf to check proof of totality by structural induction on typing derivation E-T.

Twelf checks proofs of theorems by:

- Type checking – Proof of correct proposition
- Grounds checking – Valid assumptions
- Coverage checking – Proved all cases
Ask Twelf to derive proof of totality:
\%prove 3 E-T (preservation E-T \_ \_).

- by structural induction on typing derivation E-T
- 3 is bound on the size of proof terms.
Totality Proof Automation

- Ask Twelf to **derive** proof of totality:
  \%prove 3 E-T (preservation E-T _ _).
  - by structural induction on typing derivation E-T
  - 3 is bound on the size of proof terms.

- Twelf fails to find proof of **progress** theorem because it requires nested case analysis.
  - Need extra theorems for sub-cases (no case-split construct).
  - See Twelf page on **Output Factoring** for more details:
    http://twelf.org/wiki/Output_factoring
My Review of Twelf: The Good

- **Language Simplicity**: Fewer language constructs
  - Functions encode many language elements
    (e.g., grammar, judgments, theorems, etc.)

- Good tool to **start with** for learning about proof assistants
  because of language simplicity and less syntactic sugar
  (my opinion).
My Review of Twelf: The Good

- **Language Simplicity**: Fewer language constructs
  - Functions encode many language elements
    (e.g., grammar, judgments, theorems, etc.)

- Good tool to **start with** for learning about proof assistants
  because of language simplicity and less syntactic sugar
  (my opinion).

- Language support (HOAS) for **variable binding**
  - Do not need to define substitution and prove substitution
    lemmas (sometimes).

- Language support for **context-sensitive propositions**
  (hypothetical judgments).
  - Do not need to define context of judgments and related
    lemmas (e.g. weakening) (sometimes).
My Review of Twelf: The Bad

- Language sometimes too simple
  - Missing support for frequent use-cases (e.g., nested case analysis, no case-split construct).

- Less verbosity can lead to cryptic code:
  Intent and meaning of code not clear without significant background:
  - No text suggesting this is a proof case:
    \[
    _ : \text{preservation (of/len _) (step/lenV _)} \text{ of/nat}.
    \]
  - No text suggesting this checks a proof of a theorem:
    \[
    \%\text{worlds () (preservation _ _ _)}. \\
    \%\text{total E-T (preservation E-T _ _)}. \\
    \]

- Error messages could be improved (e.g. missing cases messages):
  - Type annotations of function applications and defined names desired.
No support for stepping through proof instead of just reading proof trees.

Lack of automation: Many proofs require manual specification (e.g. proofs requiring nested case analysis).

No libraries.

▶ No standard library
▶ No import statements – All code must be included (repeat definition of \texttt{nat} for every project using them)

No polymorphism

▶ Separate definitions for \texttt{(int\_list)}, \texttt{(str\_list)}, etc.
▶ Each type needs its own definition of equality.

Many contexts require explicit definition (HOAS not always sufficient).
Summary

- Twelf is a proof assistant tool for checking and deriving proofs of properties of languages and deductive logics.
- A tool for language design and implementation.
- Imposes healthy reality and sanity check on language designs.
- Exposes, and helps correct, subtle design errors early in the process.