Twelf Tutorial
Twelf Encoding of Minilang

John Altidor
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.
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. \( \Delta; \Gamma \vdash e. \mathcal{P}m(e) : T \mid \Delta' \)
  b. \( \emptyset \vdash \Delta \text{ OK} \)
  c. \( \Delta \vdash \Delta' \text{ OK} \)
  d. \( \forall x \in \text{dom}(\Gamma) : \Delta \vdash \Gamma(x) \text{ OK} \)

then:
  there exists \( \Delta_n \)

where:
  \( \Delta', \Delta_n = \Delta'', \overline{\Delta} \)
  \( \Delta \vdash \Delta', \Delta_n \text{ OK} \)
  \( \Delta; \Gamma \vdash e : \exists \Delta''.N \mid \emptyset \)
  \( mType(m, N) = \langle Y < B > U \mapsto U \)
  \( \Delta; \Gamma \vdash e : \exists \Delta. R \mid \emptyset \)
  \( \text{match}(sift(R, U, Y), P, Y, T) \)
  \( \Delta \vdash P \text{ OK} \)
  \( \Delta', \Delta'' \vdash \overline{\Delta} \vdash R <: [T/Y]U \)
  \( \Delta', \Delta'' \vdash \exists 0. R <: [T/Y]U \)
  \( \Delta', \Delta_n \vdash [T/Y]U <: T \)

Proof by structural induction on the derivation of \( \Delta; \Gamma \vdash e. \mathcal{P}m(e) : T \mid \Delta' \)
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)
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.
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)
Twelf Proof Assistant

- 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 $\neg P$.  
  
No case-split on undecidable propositions. Not allowed:

$\text{If } \text{halts}(\text{TuringMachine}) \text{ then proof of } A \text{ else proof of } B$.

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

$$\ln(x) = u \text{ such that } x = e^u.$$ definition in Isabelle/HOL:

\[
\text{definition } \text{ln} :: \text{real} \Rightarrow \text{real} \text{ where}\\
\text{ln } x = \text{THE } u. \exp u = x.
\]

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 $B$. 

\[ \ln(x) = u \text{ such that } x = e^u. \]

Definition in Isabelle/HOL:
\[ \text{definition} \ \ln :: \text{real} \Rightarrow \text{real} \ where \ \
\ln x = \text{THE } u. \exp u = x. \]
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 \( \ln :: \text{real} \Rightarrow \text{real} \) where
  \( \ln x = \text{THE } u . \exp u = x \).
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 \( \ln :: \text{real} \Rightarrow \text{real} \) where
    \( \ln x = \text{THE u. exp u = x} \).

- In Twelf: Writing a proof = Writing a program.
  - Proofs are programs.
Twelf Live Server

- 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.
Kinds: Category of Types

- 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**.
- **ArrayInt** is of kind **Array**.

The kind type is a pre-defined kind in Twelf.
Kinds: Category of Types

- 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)
Kinds: Category of Types

- 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.
Kinds: Category of Types

- 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:

\[
\begin{align*}
\text{int} &: \text{type.} \quad \text{one} &: \text{int.} \\
\text{plusOne} &: \text{int} \to \text{int}. \\
\end{align*}
\]

- **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 \(\to\) int.
- \((\text{plusOne one})\) has type int.
Twelf supports defining functions:

\[
\text{int} : \text{type}. \quad \text{one} : \text{int}.
\]

\[
\text{plusOne} : \text{int} \to \text{int}.
\]

- 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 \to int.
- \((\text{plusOne one})\) has type int.

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

\[
\text{plus} : \text{int} \to \text{int} \to \text{int}.
\]

- int \to int \to int is curried form of \((\text{int, int}) \to \text{int}\).
- int \to int \to int = int \to (\text{int} \to \text{int})
- \((\text{plus one})\) has type int \to int.
- \((\text{plus one one})\) has type 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`. 

Functions Returning Types

- 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.
- oneIsOne : (equalsOne one).
  - Defines a new term oneIsOne of type (equalsOne one).
Recall that **type** is a **kind** (type of types).

Functions can also return **types**:

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

- **oneIsOne**: \((\text{equalsOne \ one})\).
  - Defines a new **term** **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`.

Grammar productions encoded w/ functions between syntactic categories.

- `add : exp -> exp -> exp.`
- `add` takes in two arguments.
- `exp -> exp -> exp` is curried form of `(exp, exp) -> 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.

- \( \text{add} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp} \).
- \( \text{Expression} \ e \ : \ := \ + (e_1 ; e_2) \).
- \( \text{add} \) 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:
    - $+(\text{num}[3]; \text{num}[4])$
    - $\text{add} \ (\text{enat} \ 3) \ (\text{enat} \ 4)$

- 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 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 **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.
Functions are really terms with holes/unknowns: (3 + ●).
Holes are represented by variables.
Holes filled in by applying (terms w/ holes)/functions.
Holes abstract details.
“x.e” represented by lambda abstraction “λx : τ.e”.
Twelf’s syntax of “λx : τ.e”: “[x : τ] e”
let expression in Twelf HOAS

- Twelf type signature of `let`:

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

- Example HOAS term in Twelf:

  Concrete Syntax

  Twelf HOAS

  `let x = 1 + 2`

  `in x + 3`

  `let (add 1 2)`

  `\{x:exp\} add x 3`

  `x.e2`

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 let:

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

- Example HOAS term in Twelf:

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

John Altidor
Twelf Tutorial 15/68
let expression in Twelf HOAS

- Twelf type signature of let:

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

- Example HOAS term in Twelf:

<table>
<thead>
<tr>
<th>Concrete Syntax</th>
<th>Twelf HOAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>let (x = 1 + 2) in (x + 3)</td>
<td>let (\text{add}\ 1\ 2) ([x:exp] 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 are defined with **type families**: Functions that return types (not terms).
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \exp \rightarrow \text{typ} \rightarrow \text{type} \).
Predicates in Twelf

- Predicates are defined with **type families**: Functions that return types (not terms).
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \ exp \to \ typ \to \text{type}. \)
- Type families return **dependent types**.
  - Types that contain terms (or **depend on** terms).

Examples:
- \( 3 \) is a term of type \( \text{nat} \).
- Let \( \text{Vec} \) be a function that given \( \text{nat} \) \( n \), \( \text{Vec}(n) \) returns the type of \( n \)-dimensional vectors.
- \( \text{Vec}(3) \) is a dependent type representing 3-dimensional vectors.
- \( [4, 1, 3] \) is a term of type \( \text{Vec}(3) \).
- \( \text{Vec} \) is a type family because it is a function that returns dependent types.
Predicates in Twelf

- Predicates are defined with **type families**: Functions that return types (not terms).

- Typing Predicate: $e : \tau$

- Twelf Encoding: \texttt{of : exp -> typ -> type.}

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

- Examples:
  - $3$ is a **term** of type $\text{nat}$.
Predicates in Twelf

- Predicates are defined with **type families**: Functions that return types (not terms).
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \exp \to \typ \to \text{type}. \)
- Type families return **dependent types**.
  - Types that contain terms (or depend on terms).
- Examples:
  - \( 3 \) is a term of type \( \text{nat} \).
  - Let \( \text{Vec} \) be a function that given \( \text{nat} \) \( n \), \( \text{Vec}(n) \) returns the type of \( n \)-dimensional vectors.
Predicates in Twelf

- Predicates are defined with **type families**: Functions that return types (not terms).
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \exp \rightarrow \typ \rightarrow \type. \)
- Type families return **dependent types**.
  - Types that contain terms (or depend on terms).
- Examples:
  - 3 is a **term** of type **nat**.
  - Let \( \text{Vec} \) be a function that given **nat** \( n \), \( \text{Vec}(n) \) returns the type of \( n \)-dimensional vectors.
  - \( \text{Vec}(3) \) is a **dependent type** representing 3-dimensional vectors.
Predicates in Twelf

- Predicates are defined with **type families**: Functions that return types (not terms).
- Typing Predicate: \( e : \tau \)
- Twelf Encoding: \( \text{of} : \exp \rightarrow \typ \rightarrow \type. \)
- Type families return **dependent types**.
  - Types that contain terms (or depend on terms).
- Examples:
  - 3 is a **term** of type **nat**.
  - Let Vec be a function that given nat \( n \), Vec\( (n) \) returns the type of \( n \)-dimensional vectors.
  - Vec\( (3) \) is a **dependent type** representing 3-dimensional vectors.
  - [4, 1, 3] is a **term** of type Vec\( (3) \).
Predicates in Twelf

- Predicates are defined with **type families**: Functions that return types (not terms).
- Typing Predicate: $e : \tau$
- Twelf Encoding: $\text{of} : \exp \to \typ \to \text{type}$.
- Type families return **dependent types**.
  - Types that contain terms (or depend on terms).
- Examples:
  - 3 is a **term** of type **nat**.
  - Let $\text{Vec}$ be a function that given **nat** $n$, $\text{Vec}(n)$ returns the type of $n$-dimensional vectors.
  - $\text{Vec}(3)$ is a **dependent type** representing 3-dimensional vectors.
  - $[4, 1, 3]$ is a **term** of **type** $\text{Vec}(3)$.
  - $\text{Vec}$ is a **type family** because it is a function that returns dependent types.
Judgments are Dependent Types

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

- Judgments/Propositions (instantiations 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 (instantiations 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 \((\text{of} \ e \ \tau)\).
- 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 $T$. 

Twelf Syntax for $S \rightarrow T$: $S \rightarrow T$

Twelf Syntax for $\Pi x : S. T$: $\{x : S\} T$
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 \([s/x]T\).
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 $\Pi x : S. T \equiv S \rightarrow T$. 
Pi-Abstractions

- **Function/Lambda Abstraction** “λx : 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

- **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-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** \( \Pi x : S. T \):
  \[ \{x:S\}T \]
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

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

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

- \((\text{of/nat } z) \neq (\text{of } (\text{enat } z) \text{ num})\).

- Function of/nat **returns terms** not types.

- Function of returns **types**.
Inference Rules are Functions

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

- \textbf{of/nat} : \{N:nat\} of (enat N) num.

- Twelf Convention:
  - Constants start with lower-case letters.
  - Variables/parameters start with upper-case letters.

- \((\text{of/nat } z) \neq (\text{of (enat } z) \text{ num})\).

- Function \text{of/nat} returns \textbf{terms} not \textbf{types}.

- Function of returns \textbf{types}.

- \((\text{of/nat } z) = \text{term} \text{ of type (of (enat } z) \text{ num})\).
  - Example legal assignment:
    \(y : (\text{of (enat } z) \text{ num}) = (\text{of/nat } z)\).
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.

- (of/nat z) \(\neq\) (of (enat z) num).

- Function of/nat returns **terms** not types.

- Function of returns **types**.

- (of/nat z) = term of type (of (enat z) num).
  - Example legal assignment:
    - \(y : (\text{of (enat z) num}) = (\text{of/nat z})\).

- (of/nat z) is a **derivation/term** of judgment \(z : \text{num}\) represented by type (of (enat z) num).
Premises are Inputs

\[
\begin{array}{c}
e_1: \text{num} \quad e_2: \text{num} \\
\hline
+(e_1; e_2): \text{num} \quad \text{T.4}
\end{array}
\]

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

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

- $\text{of/nat}: \{N:\text{nat}\} \text{ of (enat } N\text{) num}$.  
  - Parameter $N$ is **explicit** in the above signature.
  - Explicit parameters **must be specified** in function applications.

- D : $\text{of (enat } z\text{) num} = \text{of/nat } z$.  
  - Twelf figures out from the context (type of left-hand side of assignment) that $z$ is the implicit parameter that $\text{of/nat}$ should be applied to.
Implicit and explicit parameters

- Parameter \(N\) is \textbf{explicit} in the above signature.
- Explicit parameters \textbf{must be specified} in function applications.
- \(D : \text{of (enat } z\text{) num = of/nat } z\).

- of/nat: \text{of (enat } N\text{) num}.
- Parameter \(N\) is \textbf{implicit} in the above signature.
- Implicit parameters \textbf{cannot be specified} by programmer in function applications.
- \(D : \text{of (enat } z\text{) num = of/nat}\).
- Twelf figures out from the context (type of left-hand side of assignment) that \(z\) is the implicit parameter that \text{of/nat} should be applied to.
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.
- 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., \(\text{type}\).

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 \(H \in H \iff H \notin H \implies \text{False}\).

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.
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>$\mathbin{:=}$ zero</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\pi$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\sqrt{-1}$</td>
</tr>
<tr>
<td>Types</td>
<td>$t$</td>
<td>$\mathbin{:=}$ number</td>
<td>$num$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$real$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$complex$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$int$</td>
</tr>
</tbody>
</table>
More Exercises

- **Subtyping** Rules (not all):
  
  \[
  \text{complex} <: \text{num} \quad \text{real} <: \text{num} \quad \text{int} <: \text{real}
  \]

- **Typing** Rules (not all):
  
  \[
  0 : \text{int} \quad \pi : \text{real} \quad \sqrt{-1} : \text{complex}
  \]

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

Define **subsumption** rule for typing judgment.

**Prove** \(0 : \text{num}\).

- Fill in the blank below:
- \(D : (\text{of zero number}) = \bullet\)
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 with **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

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

Twelf Encoding:

\[
\text{of/let} : (\{x: \text{exp}\} \text{of } x \ 	ext{T1} \to \text{of } (E2 \ x) \ 	ext{T2}) \to \\
\text{of } E1 \ 	ext{T1} \to \\
\text{of } (\text{let } E1 ([x] E2 x)) \ 	ext{T2}.
\]
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:**
  \[
  \text{of/let} : \left( \{x: \text{exp}\} \text{of } x \ T_1 \to \text{of } (E_2 \ x) \ T_2 \right) \to \\
  \text{of } E_1 \ T_1 \to \\
  \text{of } (\text{let } E_1 \ ([x] \ E_2 \ x)) \ T_2.
  \]

- **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 \to \text{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 $\text{of/let}$’s first input type:

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

That type models hypothetical judgment: $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. 

$dx$ of type $\text{of (E2 x) } T_2$ can be used in the definition of $f$ to return a proof/term of type $\text{of (E2 x) } T_2$. 

The ability to use a proof ($dx$) of type $\text{of (E2 x) } T_2$ to derive a proof of type $\text{of (E2 x) } T_2$ simulates the ability to use an assumption $x : \tau_1$ to prove $e_2 : \tau_2$. 

John Altidor

Twelf Tutorial 28/68
Let $f$ be a function of of/let’s first input type:

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

That type models hypothetical judgment: $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.

$f$’s first input is an exp term bound to LF variable $x$. $dx$ of type $(\text{of } x \ T_1)$ can be used in the definition of $f$ to return a proof/term of type $(\text{of } (E2 \ x) \ T_2)$.

The ability to use a proof ($dx$) of type $(\text{of } x \ T_1)$ to derive a proof of type $(\text{of } (E2 \ x) \ T_2)$ simulates the ability to use an assumption $x : \tau_1$ to prove $e_2 : \tau_2$. 

John Altidor
Twelf Tutorial 28/68
Let $f$ be a function of of/let’s first input type:
$$
\{x: \text{exp}\} \text{ of } x \text{ T1 } \rightarrow \text{ of } (E2 \text{ x}) \text{ T2}.
$$
That type models hypothetical judgment: $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.

- $f$’s first input is an exp term bound to LF variable $x$.
- $f$’s second input is a term $dx$ of type $(\text{of } x \text{ T1})$.
  $dx =$ proof that $f$’s first input has type $T1$.

The ability to use a proof ($dx$) of type $(\text{of } x \text{ T1})$ to derive a proof of type $(\text{of } (E2 \text{ x}) \text{ T2})$ simulates the ability to use an assumption $x : \tau_1$ to prove $e_2 : \tau_2$. 

John Altidor
Twelf Tutorial
28/68
Let $f$ be a function of of/let’s first input type:
$$(\{x: \text{exp}\} \text{ of } x \ T1 \to \text{of} \ (E2 \ x) \ T2).$$
That type models hypothetical judgment: $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.
f’s first input is an exp term bound to LF variable $x$.
f’s second input is a term $dx$ of type $(\text{of} \ x \ T1)$.
$dx = \text{proof that } f’s \text{ first input has type } T1$
f’s output is a term of type $(\text{of} \ (E2 \ x) \ T2)$:
a proof that $E2$ instantiated with exp term $x$ has type $T2$. 
of/let’s first input type

Let \( f \) be a function of of/let’s first input type:
\[
\{x: \text{exp}\} \text{ of } x \text{ T1 } \rightarrow \text{ of } (E2 \ x) \text{ T2}.
\]
That type models hypothetical judgment: \( \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \).

\( f \)’s first input is an exp term bound to LF variable \( x \).

\( f \)’s second input is a term \( dx \) of type \( \text{of } x \text{ T1} \).
\( dx = \) proof that \( f \)’s first input has type \( T1 \).

\( f \)’s output is a term of type \( \text{of } (E2 \ x) \text{ T2} \):
a proof that \( E2 \) instantiated with exp term \( x \) has type \( T2 \).

\( dx \) of type \( \text{of } x \text{ T1} \) can be used in the definition of \( f \) to return a proof/term of type \( \text{of } (E2 \ x) \text{ T2} \).

The ability to use a proof \( (dx) \) of type \( \text{of } x \text{ T1} \) to derive a proof of type \( \text{of } (E2 \ x) \text{ T2} \) simulates the ability to use an assumption \( x : \tau_1 \) to prove \( e_2 : \tau_2 \).
Exercise Applying Hypothetical Judgment

- Derive the judgment $\vdash \text{let } x \text{ be } 1 \text{ in } x + 0 : \text{num}$ in Twelf.
- Twelf encoding of above judgment:
  \[ \text{of (let (enat (s z)) ([x:exp] add x (enat z))) num.} \]
- Recall important signatures (displaying implicit parameters):
  \[ \text{of/let : } \{T1:typ\} \{E2:exp -> exp\} \{T2:typ\} \{E1:exp\} (\{x:exp\} \text{ of } x \text{ T1} -> \text{ of (E2 } x \text{) T2} \leftarrow \text{ of E1 } T1 \leftarrow \text{ of (let E1 ([x:exp] E2 } x \text{)) T2.}\]
  \[ \text{of/nat : } \{N:nat\} \text{ of (enat N) num.} \]
  \[ \text{of/add : } \{E2:exp\} \{E1:exp\} \{E2:exp\} \{E1:exp\} \text{ of E2 } \text{ num} -> \text{ of E1 } \text{ num} -> \text{ of (add E1 E2) num.} \]
Solution to Previous Exercise

- Derive the judgment \( \vdash \text{let } x \text{ be } 1 \text{ in } x + 0 : \text{num} \) in Twelf.

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

- Recall important signatures (without implicit parameters):
  
  \[
  \begin{align*}
  &\text{of/let} : \{\{x:\text{exp}\} \; \text{of } x \; T_1 \rightarrow \text{of } (E_2 \; x) \; T_2 \} \rightarrow \\
  &\quad \text{of } E_1 \; T_1 \rightarrow \text{of } (\text{let } E_1 ([x:\text{exp}] \; E_2 \; x)) \; T_2. \\
  &\text{of/nat} : \text{of } (\text{enat } N) \; \text{num}. \\
  &\text{of/add} : \\
  &\quad \text{of } E_2 \; \text{num} \rightarrow \text{of } E_1 \; \text{num} \rightarrow \text{of } (\text{add } E_1 \; E_2) \; \text{num}.
  \end{align*}
  \]

- Twelf proof of above judgment:
  
  \[
  \text{of/let} \\
  ([x:\text{exp}] \; [dx:\text{of } x \; \text{num}] \; \text{of/}add \; (\text{of/nat } z) \; dx). \\
  (\text{of/nat } (s \; z))
  \]

  \]
Relations w/ Inputs and Outputs (Modes)

- Inputs/Outputs defined with `%mode` declaration.
  of : exp -> typ -> type.
  `%mode` of +E -T.

- Inputs marked with +.

- Outputs marked with −.
Inputs/Outputs defined with \%mode declaration.
\%mode of exp \rightarrow typ \rightarrow type.
\%mode of +E -T.

Inputs marked with +.

Outputs marked with –.

Outputs can be derived automatically using Twelf’s logic programming engine (example later).
Relations w/ Inputs and Outputs (Modes)

- Inputs/Outputs defined with \%mode declaration.
  \[
  \text{of : exp \to typ \to type.}
  \%
  \text{mode of +E \, -T.}
  \]
- 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.
Relations w/ Inputs and Outputs (Modes)

- 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).
- 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 “\( \rightarrow \)” reflects **order that premises are passed to rules/functions** and makes proofs more natural.
- Backward “\( \leftarrow \)” reflects **order of resolving ground terms**.

---

Order of args allowed by Twelf:

\[
of/let : (\{x: \text{exp}\} of x T1 \to of (E2 x) T2) \to of E1 T1 \to of \text{let} E1 ([x] E2 x) T2.
\]

Order of args that causes error:

\[
of/let : of E1 T1 \to (\{x: \text{exp}\} of x T1 \to of (E2 x) T2) \to of \text{let} E1 ([x] E2 x) T2.
\]

Error message:

Occurrence of variable T1 in output (-) argument not necessarily ground
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:

\[
\text{of/let} : (\{x : \text{exp}\} \text{of } x \text{ T1 } \rightarrow \text{of } (E2 \text{ x}) \text{ T2}) \rightarrow \\
\text{of } E1 \text{ T1 } \rightarrow \\
\text{of } (\text{let } E1 ([x] E2 \text{ x})) \text{ T2}.
\]
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 “\(\rightarrow\)” reflects **order that premises are passed to rules/functions** and makes proofs more natural.

Backward “\(\leftarrow\)” reflects **order of resolving ground terms**.

**Order of args** allowed by Twelf:

\[
\text{of/let} : \ (\{x: \exp\} \ \text{of} \ x \ T_1 \rightarrow \ \text{of} \ (E_2 \ x) \ T_2) \rightarrow \\
\text{of} \ E_1 \ T_1 \rightarrow \\
\text{of} \ (\text{let} \ E_1 ([x] E_2 x)) \ T_2.
\]

**Order of args that causes error**:

\[
\text{of/let} : \ \text{of} \ E_1 \ T_1 \rightarrow \\
(\{x: \exp\} \ \text{of} \ x \ T_1 \rightarrow \ \text{of} \ (E_2 \ x) \ T_2) \rightarrow \\
\text{of} \ (\text{let} \ E_1 ([x] E_2 x)) \ T_2.
\]
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:
  
  
  \[
  \text{of/let} : (\{x \mathbin{:} \text{exp}\} \text{of } x \ T_1 \to \text{of } (E_2 \ x) \ T_2) \to \\
  \text{of } E_1 \ T_1 \to \\
  \text{of } (\text{let } E_1 ([x] E_2 \ x)) \ T_2.
  \]

- **Order of args that causes error**:
  
  
  \[
  \text{of/let} : \text{of } E_1 \ T_1 \to \\
  (\{x \mathbin{:} \text{exp}\} \text{of } x \ T_1 \to \text{of } (E_2 \ x) \ T_2) \to \\
  \text{of } (\text{let } E_1 ([x] E_2 \ x)) \ T_2.
  \]

- **Error message**:
  Occurrence of variable $T_1$ 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)).
```
Universally-Quantified Inputs

- 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`. 
Universally-Quantified Inputs

- 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.
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.
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:

\[
\text{right1 : of E1 num -> of (add E1 (enat z)) num.}
\]
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.
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 \texttt{of} relation is an \texttt{output} argument.

\begin{verbatim}
right2 : of E T -> of (add E (enat z)) T.
\end{verbatim}
Output terms resulting from grounded input terms are also ground.

Second argument of the \texttt{of} relation is an \texttt{output} argument.

\begin{verbatim}
right2 : of E T -> of (add E (enat z)) T.
\end{verbatim}
Output terms resulting from grounded input terms are also ground.

Second argument of the \texttt{of} relation is an \texttt{output} argument.

\hspace{1cm} \texttt{right2 : of E T \rightarrow of (add E (enat z)) T.}
Output terms resulting from grounded input terms are also ground.

Second argument of the of relation is an output argument.

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

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.

right2 : of E T -> of (add E (enat z)) T.

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

\[
\text{wrong2 : of } E \text{ T1 } \to \text{ of (add E (enat z)) T2.}
\]

Output term \( \text{T2} \) is universally quantified instead of a grounded result of the input term.
This violates the \( \%\text{mode} \) declaration of the of relation.
Previous Examples for Grounds Checking

- 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 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 if and only if (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.
Checking Syntax-Directed

- Check that rules of relation/type family (e.g. subtype) are syntax-directed by passing relation to \%\texttt{unique} declaration.
- \%\texttt{unique} checks if output arguments of relation are uniquely determined by input arguments.
- \%\texttt{unique} also checks if two rules overlap or can derive the same judgment.

```plaintext
subtype : typ -> typ -> type.
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 \texttt{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))))).
Twelf Theorems

- **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 (of \( E \ T \)) and (\( \text{step } E \ E' \)), then (of \( E' \ T \)).

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

Preservation, re-formulated:

For every derivation of (of \( E \ T \)) and (\( \text{step } E \ E' \)),
there exists at least one derivation of (of \( E' \ T \)).

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

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

Premises are inputs. Conclusions are outputs.

\[ \%\text{mode}\ \text{preservation} \ +0 \ +S \ -0'. \]

To prove preservation theorem, need to show \text{preservation} is a \textbf{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: \((T.4, D.1)\)

\[
\begin{array}{c}
e_1 : \text{num} \quad e_2 : \text{num} \\
+ (e_1; e_2) : \text{num}
\end{array}
\]

\[
\begin{array}{c}
e_1 \mapsto e'_1
\end{array}
\]

\[
\begin{array}{c}
+ (e_1; e_2) \mapsto + (e'_1; e_2)
\end{array}
\]

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:

\[
\begin{array}{c}
e'_1 : \text{num} \\
+ (e'_1; e_2) : \text{num}
\end{array}
\]

\[
\begin{array}{c}
\square
\end{array}
\]
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)).
- : 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

\[-: \text{preservation} \]
\[(\text{of/add E2-num E1-num})\]
\[(\text{step/add1 E1=} \Rightarrow \text{E1'})\]
\[(\text{of/add E2-num E1'}-\text{num})\]
\[<- \text{preservation E1-num E1=} \Rightarrow \text{E1'} \text{ E1'}-\text{num}.\]

- 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.
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).
Cannot prove theorem by just assuming conclusion of theorem holds.
Cannot prove theorem by just assuming conclusion of theorem holds.

Also, cannot assume propositions not derived from premises of theorem.
Cannot prove theorem by just assuming conclusion of theorem holds.

Also, cannot assume propositions not derived from premises of theorem.

Such a proof will contain a non-ground term.
  ▶ %mode declarations used to check proofs.
Recall Valid Proof of Case

- : \{E1\text{-num} : \text{of E1 num}\}
  \{E2\text{-num} : \text{of E2 num}\}
  \{E1\text{=}E1' : \text{step E1 E1'}\}
  \{E1'\text{-num} : \text{of E1' num}\}

preservation E1\text{-num} E1\text{=}E1' E1'\text{-num}

\rightarrow \text{preservation (of/add E2\text{-num} E1\text{-num})}
  \quad \text{(step/add1 E1\text{=}E1')}
  \quad \text{(of/add E2\text{-num} E1'\text{-num}).}
Invalid Proof of Case

- \{E1\text{-num} : \text{of } E1 \text{ num} \}
  \{E2\text{-num} : \text{of } E2 \text{ num} \}
  \{E1 \Rightarrow E1' : \text{step } E1 \text{ E1'} \}
  \{E1'\text{-num} : \text{of } E1' \text{ num} \}

\text{preservation (of/add } E2\text{-num } E1\text{-num)}
  (\text{step/add1 } E1 \Rightarrow E1')
  (\text{of/add } E2\text{-num } E1'\text{-num}).

- Proof above just assumes \text{of } E1' \text{ num}, which is not one of the assumptions for the case.
- \text{E1'\text{-num} is not an input term in the conclusion (third) argument of preservation.}
- \text{E1'\text{-num is not an output term derived from ground terms.}
- Twelf reports error for function above.
Twelf checks proofs of theorems by verifying three key aspects:

- Type checking – Proof of correct proposition
- Grounds checking – Valid assumptions
- Coverage checking – Proved all cases of theorem

Next few slides describes Twelf’s coverage checking of proofs
Specifying Worlds – Possible Inputs

- 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).
  ```
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).
```

- 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.

\[\text{add} : \exp \to \exp \to \exp.\]
\[\text{let} : \exp \to (\exp \to \exp) \to \exp.\]
\[\%\text{worlds} () (\exp).\]
Let expression contains binders.

```plaintext
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
```
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: 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.
Blocks: Patterns describing fragment of contexts.

Update addressing previous error:

\[\text{add} : \text{exp} \to \text{exp} \to \text{exp}.\]
\[\text{let} : \text{exp} \to (\text{exp} \to \text{exp}) \to \text{exp}.\]
\[\%\text{block} \ \text{exp-block} : \text{block} \{x:\text{exp}\}.\]
\[\%\text{worlds} \ (\text{exp-block}) \ (\text{exp}).\]

Informs Twelf that terms of type \text{exp} can contain binders of type \text{exp}.

Worlds can take in multiple blocks. Syntax:

\[\%\text{worlds} \ (\text{block1} \mid \text{block2} \mid \ldots \mid \text{blockN}) \ (\text{exp}).\]
Specifying world requires specifying how variables are quantified (universal inputs or ground outputs).

\[ \text{of : exp -> typ -> type.} \]
\[ \%\text{mode of +E -T.} \]

\[ \ldots \]
\[ \text{of/let : of (let E1 ([x] E2 x)) T2} \]
\[ \quad \leftarrow \text{of E1 T1} \]
\[ \quad \leftarrow (\{x: \text{exp}\} \text{of x T1} \rightarrow \text{of (E2 x) T2}). \]
\[ \%\text{block of-block :} \]
\[ \quad \text{some \{T:typ\} block \{x: \text{exp}\}\{_\: \text{of x T}\}.} \]
\[ \%\text{worlds (of-block) (of \_ \_).} \]

Number of args specified by pattern in \%\text{worlds} declaration:

\( \text{(of \_ \_)} \)
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.

```plaintext
%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 of theorem
Ask Twelf to **derive** proof for all cases of theorem: 
%prove 3 E-T (preservation E-T _ _).

- by structural induction on typing derivation E-T
- 3 is bound on the size of proof terms.
Ask Twelf to **derive** proof for all cases of theorem:
\[
\%prove 3 \text{ E-T (preservation E-T \_ \_).}
\]
  - 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).

**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 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:
    - : preservation (of/len _) (step/lenV _) of/nat.
  - No text suggesting this checks a proof of a theorem:
    - %worlds () (preservation _ _ _).
    - %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).
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.