

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.

- 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.

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.

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)



- 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.

Constructive Logic

- Twelf is a **constructive** (not classical) proof assistant.
- Proposition is true **iff** there exists a proof of it.
- Law of excluded middle not assumed: $P \vee \neg P$.
 - ▶ Proving $P \vee \neg P$ **requires** either:
 - ▶ Proof of P **OR** Proof of $\neg P$.

Constructive Logic

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

Constructive Logic

- Twelf is a **constructive** (not classical) proof assistant.
- Proposition is true **iff** there exists a proof of it.
- Law of excluded middle not assumed: $P \vee \neg P$.
 - ▶ Proving $P \vee \neg P$ **requires** either:
 - ▶ Proof of P **OR** Proof of $\neg P$.
 - ▶ No case-split on undecidable propositions. Not allowed:
 - ▶ If *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:
definition ln :: real => real where
ln x = **THE** u. exp u = x.

Constructive Logic

- Twelf is a **constructive** (not classical) proof assistant.
- Proposition is true **iff** there exists a proof of it.
- Law of excluded middle not assumed: $P \vee \neg P$.
 - ▶ Proving $P \vee \neg P$ **requires** either:
 - ▶ Proof of P **OR** Proof of $\neg P$.
 - ▶ No case-split on undecidable propositions. Not allowed:
 - ▶ If *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:
definition ln :: real => real where
ln x = **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.

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.

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.

Functions

- Twelf supports defining functions:

```
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

- Twelf supports defining functions:

```
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**:

```
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 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`.

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)`.

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)`.
- A function type is a kind if its return type is also a kind.
 - ▶ `int -> type` is a kind.
 - ▶ `int -> (int -> type)` is a kind.
 - ▶ `int -> int -> type = int -> (int -> type)` is a kind.
- **type** is **not** allowed on the left-hand side of arrow (`->`).

Minilang Syntax in Twelf

- 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`.

Minilang Syntax in Twelf

- 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.`
 - ▶ *Expression* $e : : = +(e_1; e_2)$
 - ▶ `add` takes in **two** arguments.
 - ▶ `exp -> exp -> exp` is curried form of $(exp, exp) \rightarrow exp$.

Terms w/ variables using Higher-Order Abstract Syntax (HOAS)

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

Terms w/ variables using Higher-Order Abstract Syntax (HOAS)

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

- ASTs in **Higher-Order Abstract Syntax (HOAS)**:

- Each t_i in $o(t_1, t_2, \dots, t_n)$ has form:

$x_1, x_2, \dots, x_k.t$

- 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.

HOAS encoding of `let` expression

- First, `let` expression in FOAS:

`let(x; e1; e2)`

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 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**.

Higher-Order Terms are Functions

- 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.
- “ $x.e$ ” represented by **lambda abstraction** “ $\lambda x : \tau.e$ ”.
- Twelf’s syntax of “ $\lambda x : \tau.e$ ”: “ $[x : \tau] \ e$ ”

let expression in Twelf HOAS

- Twelf type signature of let:

$$\text{let} : \text{exp} \rightarrow \underbrace{(\text{exp} \rightarrow \text{exp})}_{x.e_2} \rightarrow \text{exp}.$$

let expression in Twelf HOAS

- Twelf type signature of let:

$$\text{let} : \text{exp} \rightarrow \underbrace{(\text{exp} \rightarrow \text{exp})}_{x.e_2} \rightarrow \text{exp}.$$

- Example HOAS term in Twelf:

<i>Concrete Syntax</i>	<i>Twelf HOAS</i>
$\text{let } x = \underbrace{1 + 2}_{e_1} \text{ in } \underbrace{x + 3}_{e_2}$	$\text{let } \underbrace{(\text{add } 1 \ 2)}_{e_1} \underbrace{([x:\text{exp}] \ \text{add } x \ 3)}_{x.e_2}$

let expression in Twelf HOAS

- Twelf type signature of let:

$$\text{let} : \text{exp} \rightarrow \underbrace{(\text{exp} \rightarrow \text{exp})}_{x.e_2} \rightarrow \text{exp}.$$

- Example HOAS term in Twelf:

<i>Concrete Syntax</i>	<i>Twelf HOAS</i>
$\text{let } x = \underbrace{1 + 2}_{e_1} \text{ in } \underbrace{x + 3}_{e_2}$	$\text{let } \underbrace{(\text{add } 1 \ 2)}_{e_1} \underbrace{([x:\text{exp}] \ \text{add } x \ 3)}_{x.e_2}$

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

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).

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:

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: `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**.
 - ▶ $\text{Vec}(n)$ 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: `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**.
 - ▶ $\text{Vec}(n)$ 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: `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**.
 - ▶ $\text{Vec}(n)$ 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 `(of (enat z) num)`.
- Dependent type `(of e τ)` represents judgment “ $e : \tau$ ”.

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 `(of (enat z) num)`.
- Dependent type `(of e τ)` represents judgment “ $e : \tau$ ”.
- **Derivation/Proof** of “ $e : \tau$ ” represented by **term** of type `(of e τ)`.
- Curry-Howard Correspondence:
Proofs are terms.
Propositions/Judgments are types.

- 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/**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 $\prod x : S. T$:
 - ▶ Takes in a term s of type S .
 - ▶ Returns a term of type $[s/x]T$.

- 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 $\Pi x : S. T \equiv S \rightarrow T$.

- 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.

- 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 $x \notin fv(T)$, then $\Pi x : S. T \equiv S \rightarrow T$.
- If $x \in fv(T)$, then $\Pi x : S. T$ returns term of a dependent type.
- Twelf Syntax for $S \rightarrow T$:
 $S \multimap T$

- 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 $x \notin fv(T)$, then $\Pi x : S. T \equiv S \rightarrow T$.
- If $x \in fv(T)$, 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

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

- `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

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

- `of/nat` : $\{N:\text{nat}\}$ of `(enat N) num`.
- Twelf Convention:
 - ▶ Constants start with lower-case letters.
 - ▶ Variables/parameters start with upper-case letters.
- `(of/nat z) ≠ (of (enat z) num)`.
- `of/nat` **returns terms** not types.

Inference Rules are Functions

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

- `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) ≠ (of (enat z) num).`
- `of/nat` **returns terms** not types.
- `(of/nat z) = term` of type `(of (enat z) num).`
 - ▶ Example legal assignment:
`y : (of (enat z) num) = (of/nat z).`

Inference Rules are Functions

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

- `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) ≠ (of (enat z) num).`
- `of/nat` **returns terms** not types.
- `(of/nat z) = term` of type `(of (enat z) num).`
 - ▶ Example legal assignment:
`y : (of (enat z) num) = (of/nat z).`
- `(of/nat z)` is a **derivation/term** of judgment `z : num` represented by type `of (enat z) num.`

Premises are Inputs

$$\frac{e_1 : \text{ num } \quad e_2 : \text{ num }}{+(e_1; e_2) : \text{ num }} \text{ T.4}$$

- Twelf Encoding:
of/add : of (add E1 E2) num
 <- of E1 num
 <- 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

- `of/nat`: $\{N:\text{nat}\}$ of (enat N) num.
- 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

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

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.

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.

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.
- No type polymorphism implies no **general** logical connectives.
- **Not allowed:**
`conjunction :`
`{P:type} {Q:type} P -> Q -> (and P Q).`

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., `nat`.
 - ▶ Twelf kinds are third-order terms; e.g., `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., nat .
 - ▶ Twelf kinds are third-order terms; e.g., 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 $\underbrace{H \in H \iff H \notin H}_{\text{False}}$.
- 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.

- Create language of numbers with subtyping in Twelf.

Category	Item		Abstract	Concrete
<i>Terms</i>	<i>e</i>	$::=$	zero	0
			pi	π
			img	$\sqrt{-1}$
<i>Types</i>	<i>t</i>	$::=$	number	<i>num</i>
			real	<i>real</i>
			complex	<i>complex</i>
			int	<i>int</i>

- **Subtyping** Rules (not all):

$$\overline{complex <: num}$$

$$\overline{real <: num}$$

$$\overline{int <: real}$$

- **Typing** Rules (not all):

$$\overline{0 : int}$$

$$\overline{\pi : real}$$

$$\overline{\sqrt{-1} : complex}$$

- Define **reflexive** and **transitive** rules for subtyping.
- Define **subsumption** rule for typing judgment.
- **Prove** $0 : num$.
 - ▶ Fill in the blank below:
 - ▶ $D : (\text{of zero number}) = \bullet$

Hypothetical Judgments in Twelf

- What happened to typing context Γ ?

Hypothetical Judgments in Twelf

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

Hypothetical Judgments in Twelf

- What happened to typing context Γ ?
- **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.

Hypothetical Judgments in Twelf

- What happened to typing context Γ ?
- **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)
- Γ does not need to be defined.

Typing let expression in Twelf HOAS

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

■ Twelf Encoding:

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

Typing let expression in Twelf HOAS

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

- Twelf Encoding:

of/let : $(\{x: \text{exp}\} \text{ of } x \text{ T1} \rightarrow \text{of } (E2 \ x) \text{ T2}) \rightarrow$
of E1 T1 \rightarrow
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 \rightarrow exp).
- ([x] E2 x) used for readability: # of inputs explicit.
- ([x] E2 x) is called the **eta-expansion** of E2.

of/let's first type

- 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: $\Gamma, x : \tau_1 \vdash e_2 : \tau_2.$

of/let's first type

- 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 $(\text{of } (E2 \ x) \text{ T2})$.

of/let's first type

- 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 $(\text{of } (E2 \ x) \text{ T2})$.
- dx of type $(\text{of } x \text{ T1})$ can be used in the body of f to return a proof/term of type $(\text{of } (E2 \ x) \text{ T2})$.
- The ability to use a proof (dx) of type $(\text{of } x \text{ T1})$ simulates extending typing context with $x : \tau_1$.

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:

`of (let (enat (s z)) ([x:exp] add x (enat z))) num.`

- Recall important signatures (displaying implicit parameters):

`of/let : {T1:typ} {E2:exp -> exp} {T2:typ} {E1:exp}
 ({x:exp} of x T1 -> of (E2 x) T2) -> of E1 T1
 -> of (let E1 ([x:exp] E2 x)) T2.`

`of/nat : {N:nat} of (enat N) num.`

`of/add : {E2:exp} {E1:exp}
 of E2 num -> of E1 num -> of (add E1 E2) num.`

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 `-`.

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).

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**.

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).

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 “ \rightarrow ” reflects **order that premises are passed to rules/functions** and makes proofs more natural.
- Backward “ \leftarrow ” 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:
of/let : ($\{x: \text{exp}\}$ of x $T1 \rightarrow$ of $(E2\ x)$ $T2$) \rightarrow
of $E1\ T1 \rightarrow$
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 : ($\{x: \text{exp}\}$ 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 $E1\ T1$ ->
($\{x: \text{exp}\}$ of $x\ T1$ -> of $(E2\ x)\ T2$) ->
of (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 “ \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 : $(\{x: \text{exp}\} \text{ of } x \text{ T1} \rightarrow \text{of } (E2 \ x) \text{ T2}) \rightarrow$
 of E1 T1 \rightarrow
 of (let E1 ([x] E2 x)) T2.
- Order of args that **causes error**:
of/let : of E1 T1 \rightarrow
 $(\{x: \text{exp}\} \text{ of } x \text{ T1} \rightarrow \text{of } (E2 \ x) \text{ T2}) \rightarrow$
 of (let E1 ([x] E2 x)) T2.
- Error message:
Occurrence of variable T1 in output (-) argument
not necessarily ground

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.`

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)).`

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`.

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 -> 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.`
- `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).

Ground Term From Output

- 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.

`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.

`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.

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

- Term **T** is **computed**/result of premise/**recursive call** (of **E 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.

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

- Term **T** is **computed**/result of premise/**recursive call** (of **E T**).

Non-ground Term in Conclusion Causing Error

- Output term in conclusion not grounded:

`wrong2 : of E T1 -> of (add E (enat z)) T2.`

- Output term `T2` is universally quantified instead of a grounded result of the input term.
This violates the `%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 **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

- **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 $+(5; 3) : \text{num}$.

$$\frac{\frac{}{5 : \text{num}} \text{ of/num} \quad \frac{}{3 : \text{num}} \text{ of/num}}{+(5; 3) : \text{num}} \text{ of/add}$$

- No need for exhaustive proof search with syntax-directed predicates.

Checking Syntax-Directed

- `%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`.

Printing Proof Terms

- 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`

Proof Term in Sample Output

```
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).

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).

- `%theorem`

`preservation :`

```
forall* {E} {E'} {T}
forall {O:of E T} {S:step E E'}
exists {O':of E' T}
true.
```

- Verbose syntax above.

Desugared, concise alternative on next slide.

Theorems are Function Types w/ Specified Inputs/Outputs

- Preservation theorem is a function returning types (type family):

`preservation:`

`of E T -> step E E' -> of E' T -> 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 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.

Preservation Proof - Addition Case 2 - Informal

Case: (T.4, D.1)

$$\frac{e_1 : \text{num} \quad e_2 : \text{num}}{+(e_1; e_2) : \text{num}} \text{ T.4} \qquad \frac{e_1 \mapsto e'_1}{+(e_1; e_2) \mapsto +(e'_1; e_2)} \text{ D.1}$$

We assume preservation holds for subexpressions. Hence, by the **inductive hypothesis**, $e_1 : \text{num}$ and $e_1 \mapsto e'_1$ implies $e'_1 : \text{num}$. Rule T.4 gives us:

$$\frac{e'_1 : \text{num} \quad e_2 : \text{num}}{+(e'_1; e_2) : \text{num}} \text{ T.4} \quad \square$$

Twelf Proof of Addition Case 2

```
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.

Checking Proof Totality

- 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.

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).
```

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).
```
- No term of type `nat` containing LF variables.
- No such `nat` of form `(s x)`, where `x` of variable of type `nat`.

Terms Containing Binders

- Let expression contains binders.

`add : exp -> exp -> exp.`

`let : exp -> (exp -> exp) -> exp.`

`%worlds () (exp).`

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`

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:** 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:

```
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).
```

Worlds for Relations w/ Outputs

- Specifying world requires specifying how variables are quantified (universal inputs or ground outputs).

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

```
%mode of +E -T.
```

```
...
```

```
of/let : of (let E1 ([x] E2 x)) T2
```

```
  <- of E1 T1
```

```
  <- ({x: exp} of x T1 -> of (E2 x) T2).
```

```
%block of-block :
```

```
  some {T:typ} block {x: exp}{_: of x T}.
```

```
%worlds (of-block) (of _ _).
```

- Number of args specified by pattern in %worlds declaration:
(of _ _)

Checking Proof Totality

- 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.

Checking Proof Totality

- 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

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.

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:

```
- : 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.

My Review of Twelf: The Bad (cont.)

- 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 `nat` for every project using them)
- No polymorphism
 - ▶ Separate definitions for `(int_list)`, `(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.