Dependent Types, Twelf and Its Application in Proof

Seyed Mohammad Mehdi Ahmadpanah

smahmadpanah@aut.ac.ir

Supervisor

Dr. Mehran S. Fallah

Formal Security Lab.

CEIT@AUT

Feb. 13, 2018

Outline

- Motivation
- The Curry-Howard Correspondence
- Logical Frameworks
- Pure First-Order Dependent Types
- Dependent Sum Types
- The Calculus of Constructions
- Twelf in Practice
- Programming with Dependent Types
- Conclusion

Motivation

- Dependent types are type-valued functions
  - Those functions which send $terms$ to types
- Type family of vectors (one-dimensional arrays)

$$\texttt{Vector :: Nat} \rightarrow *$$

- **Kinding assertion:** Vector maps a $\mathrm{k{:}Nat}$ to a type
  - type $\mathrm{Vector\ k}$ contains vectors of length k of elements of some fixed type, say data
- Initialization function:

$$\mathrm{init} : \Pi \mathrm{n{:}Nat.\ data} \rightarrow \mathrm{Vector\ n}$$

Motivation (cont.)

$$\text{init} : \Pi \text{n:Nat. data} \rightarrow \text{Vector n}$$

- Dependent Product Type (Pi type)

$$\Pi x : S \, . \, T$$

  - Generalizes the arrow type of the simply typed λ-calculus

$$S \rightarrow [x \mapsto s] \; T$$

  - The result type can vary according to the argument supplied
  - Π-type is almost as old as the lambda calculus

Motivation (cont.)

- **Another way of building up vectors (constructor)**

$$\text{empty} : \text{Vector } 0$$

$$\text{cons} : \Pi\text{n:Nat. data} \rightarrow \text{Vector n} \rightarrow \text{Vector (n+1)}$$

- **Example**

$$\text{v : Vector 5 , x : data } \textbf{then} \text{ cons 5 x v : Vector 6}$$

Πx : S . T (Dependent Product Type)

$$S \rightarrow [x \mapsto s]\ T$$

vs.

Universal Type ∀X. T of System F

(If t: $\forall X.T$ **and** A **is a** **type**, **then** $t\ A : X \rightarrow [X \mapsto A]\ T$**)**

Motivation (cont.)

- Why dependent typing?
  - It reveals more information about the behavior of the term
  - More precious typing
  - Exclude more of the badly behaved terms in a type system
- We can type a function that returns the first element of a non-empty vector:

$$\text{first} : \Pi\text{n:Nat. Vector(n+1)} \to \text{data}$$

  - Non-emptiness is expressed within the type system itself!

Motivation (cont.)

- Another example: sprintf

$$\text{sprintf} : \Pi \text{f:Format. Data(f)} \rightarrow \text{String}$$

  - Format: type of valid print formats
  - Data(f): type of data corresponding to format f

```
Data([])          =  Unit
Data("%d"::cs)    =  Nat * Data(cs)
Data("%s"::cs)    =  String * Data(cs)
Data(c::cs)       =  Data(cs)
```

  - Vectors are uniform, here is non-uniform
    - More challenging!

Curry-Howard Correspondence

- Proposition-as-Type (and Proof-as-Term)
  - A formula has a proof iff the corresponding type is inhabited
- Example:

$$((A \to B) \to A) \to (A \to B) \to B \quad \text{(formula, type)}$$

$$\lambda f.\ \lambda u.\ u\ (f\ u) \quad \text{(proof, term)}$$

- Constructive proof of $A \Rightarrow B$ should be understood as a procedure that transforms any given proof of A into a proof of B
  - A proof of $A \Rightarrow B$ is simply any term of type $A \to B$

Curry-Howard Correspondence (cont.)

- Generalizing the correspondence to first-order predicate logic leads to dependent types
- A proof of the universal quantification $\forall \mathrm{x}{:}\mathrm{A}.\ \mathrm{B}(\mathrm{x})$ is constructively a procedure that given an arbitrary element x of type A produces a proof of $\mathrm{B}(\mathrm{x})$
- Identification of universal quantification with dependent product
  - A proof of $\forall \mathrm{x}{:}\mathrm{A}.\ \mathrm{B}(\mathrm{x})$ is a member of $\Pi \mathrm{x}{:}\mathrm{A}.\ \mathrm{B}(\mathrm{x})$
- Existential quantification $\equiv$ Σ-types
- Equality $\equiv$ Identity types

Curry-Howard Correspondence (cont.)

- Application: freely mixing propositions and types

Example - indexing function

$$\mathrm{ith(n)} : \Pi \mathrm{n{:}Nat}.\ \Pi \mathrm{l{:}Nat}.\ \mathrm{Lt(l,n)} \to \mathrm{Vector(n)} \to \mathrm{T}$$

Example – type of binary, associative operations on some type T

$$\Sigma \mathrm{m}{:}\ \mathrm{T} \to \mathrm{T} \to \mathrm{T}.\ \Pi \mathrm{x} : \mathrm{T}.\ \Pi \mathrm{y} : \mathrm{T}.\ \Pi \mathrm{z} : \mathrm{T}.$$
$$\mathrm{Id}\ (\mathrm{m(x,m(y,z))})\ (\mathrm{m(m(x,y),z)})$$

- A proof of $\exists \mathrm{x{:}A}.\ \mathrm{B(x)}$ would consist of a member a of type A and a proof (a member) of $\mathrm{B(a)}$
  - an element of $\Sigma \mathrm{a{:}A}.\ \mathrm{B(a)}$

Logical Frameworks

- Another application: representation of other type theories and formal systems
- Example – typechecker for simply typed λ-calculus

```
Ty    :: *
Tm    :: Ty → *
base  : Ty
arrow : Ty → Ty → Ty
app   : ΠA:Ty.ΠB:Ty.Tm(arrow A B) → Tm A → Tm B
lam   : ΠA:Ty.ΠB:Ty.(Tm A → Tm B) → Tm(arrow A B)
```

- Higher-order abstract syntax
- e.g. representation of $\mathrm{identity}$ term: $(\mathrm{A}\colon \mathrm{Ty})$

$$\mathrm{idA} = \mathrm{lam}\ \mathrm{A}\ \mathrm{A}\ (\lambda x\colon \mathrm{Tm}\ \mathrm{A}.\ x)$$

Logical Frameworks (cont.)

- **Definition: systems which provide mechanisms for representing** syntax **and** proof systems
  - **which makes up a logic**
- **It provides a means to define a logic as a "signature" in a higher-order type theory**
  - Provability of a formula **in the original logic reduces to a** type inhabitation problem **in the framework type theory**
- **To describe a logical framework, one must provide:**
  - **A characterization of the class of object-logics to be represented**
  - **An appropriate meta-language**
  - **A characterization of the mechanism by which object-logics are represented**

Logical Frameworks (cont.)

- One approach: Edinburgh Logical Framework (LF)
  - Judgments-as-Types
    - Judgments: types o
    - Derivations of judgm
  - Meta-language: dependently typed λ-calculus (λΠ-calculus)
    - Three-level entities: terms, types, kinds
- Twelf is an implementation of the logical framework LF
  - Twelf code describes logical systems
  - written in Standard ML
    - write out a statement
    - use Twelf to write out a proof (justification of why that statement is true)
    - Twelf will check your proof, making sure that what you said actually is true!

Logical Frameworks (cont.)

- Twelf includes:
  - an implementation of the LF logical framework, which can be used to type check LF representations
  - a logic programming language based on LF
  - a metatheorem checker, which can be used to verify proofs of theorems about LF representations
- Other systems that will let you define logical systems and prove things with them:
  - ACL2, AUTOMATH, Coq, HOL, HOL Light, LEGO, Isabelle, MetaPRL, NuPRL PVS, and TPS
  - In Twelf: *programming languages* are also logical systems

Pure First-Order Dependent Types

- λLF
  - Type system based on a simplified variant of the type system underlying LF
  - Generalizes simply typed λ-calculus by replacing the arrow type $S \to T$ with the dependent product type $\Pi x : S \,.\, T$ and by introducing type families
  - Pure
    - Only has Π-types
  - First-Order
    - Does not include higher-order type operators
  - Corresponds to $\forall$, $\to$-fragment of first-order predicate calculus

Pure First-Order Dependent Types (cont.)

λLF

Syntax

$\mathtt{t}$ ::=		*terms:*
	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	*abstraction*
	$\mathtt{t}\ \mathtt{t}$	*application*
$\mathtt{T}$ ::=		*types:*
	$\mathtt{X}$	*type/family variable*
	$\Pi\mathtt{x}{:}\mathtt{T}.\mathtt{T}$	*dependent product type*
	$\mathtt{T}\ \mathtt{t}$	*type family application*
$\mathtt{K}$ ::=		*kinds:*
	$*$	*kind of proper types*
	$\Pi\mathtt{x}{:}\mathtt{T}.\mathtt{K}$	*kind of type families*
Γ ::=		*contexts:*
	$\varnothing$	*empty context*
	$\Gamma, \mathtt{x}{:}\mathtt{T}$	*term variable binding*
	$\Gamma, \mathtt{X}{::}\mathtt{K}$	*type variable binding*

Well-formed kinds $\boxed{\Gamma \vdash \mathtt{K}}$

$$\Gamma \vdash * \qquad \text{(WF-STAR)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{T} \vdash \mathtt{K}}{\Gamma \vdash \Pi\mathtt{x}{:}\mathtt{T}.\mathtt{K}} \qquad \text{(WF-PI)}$$

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X} :: \mathtt{K} \in \Gamma \qquad \Gamma \vdash \mathtt{K}}{\Gamma \vdash \mathtt{X} :: \mathtt{K}} \qquad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \Pi\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2 :: *} \qquad \text{(K-PI)}$$

$$\frac{\Gamma \vdash \mathtt{S} :: \Pi\mathtt{x}{:}\mathtt{T}.\mathtt{K} \qquad \Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \mathtt{S}\ \mathtt{t} : [\mathtt{x} \mapsto \mathtt{t}]\mathtt{K}} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K} \qquad \Gamma \vdash \mathtt{K} \equiv \mathtt{K}'}{\Gamma \vdash \mathtt{T} :: \mathtt{K}'} \qquad \text{(K-CONV)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma \qquad \Gamma \vdash \mathtt{T} :: *}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{S} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{S}.\mathtt{t} : \Pi\mathtt{x}{:}\mathtt{S}.\mathtt{T}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \Pi\mathtt{x}{:}\mathtt{S}.\mathtt{T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{S}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{T}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \Gamma \vdash \mathtt{T} \equiv \mathtt{T}' :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T}'} \qquad \text{(T-CONV)}$$

Pure First-Order Dependent Types (cont.)

λLF

Kind Equivalence $\boxed{\Gamma \vdash K \equiv K'}$

$$\frac{\Gamma \vdash T_1 \equiv T_2 :: * \qquad \Gamma, x{:}T_1 \vdash K_1 \equiv K_2}{\Gamma \vdash \Pi x{:}T_1.K_1 \equiv \Pi x{:}T_2.K_2} \quad \text{(QK-PI)}$$

$$\frac{\Gamma \vdash K}{\Gamma \vdash K \equiv K} \quad \text{(QK-REFL)}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2}{\Gamma \vdash K_2 \equiv K_1} \quad \text{(QK-SYM)}$$

$$\frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma \vdash K_2 \equiv K_3}{\Gamma \vdash K_1 \equiv K_3} \quad \text{(QK-TRANS)}$$

Type Equivalence $\boxed{\Gamma \vdash S \equiv T :: K}$

$$\frac{\Gamma \vdash S_1 \equiv T_1 :: * \qquad \Gamma, x{:}T_1 \vdash S_2 \equiv T_2 :: *}{\Gamma \vdash \Pi x{:}S_1.S_2 \equiv \Pi x{:}T_1.T_2 :: *} \quad \text{(QT-PI)}$$

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: \Pi x{:}T.K \qquad \Gamma \vdash t_1 \equiv t_2 : T}{\Gamma \vdash S_1\, t_1 \equiv S_2\, t_2 : [x \mapsto t_1]K} \quad \text{(QT-APP)}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash T \equiv T :: K} \quad \text{(QT-REFL)}$$

$$\frac{\Gamma \vdash T \equiv S :: K}{\Gamma \vdash S \equiv T :: K} \quad \text{(QT-SYM)}$$

$$\frac{\Gamma \vdash S \equiv U :: K \qquad \Gamma \vdash U \equiv T :: K}{\Gamma \vdash S \equiv T :: K} \quad \text{(QT-TRANS)}$$

Term Equivalence $\boxed{\Gamma \vdash t_1 \equiv t_2 : T}$

$$\frac{\Gamma \vdash S_1 \equiv S_2 :: * \qquad \Gamma, x{:}S_1 \vdash t_1 \equiv t_2 : T}{\Gamma \vdash \lambda x{:}S_1.t_1 \equiv \lambda x{:}S_2.t_2 : \Pi x{:}S_1.T} \quad \text{(Q-ABS)}$$

$$\frac{\Gamma \vdash t_1 \equiv s_1 : \Pi x{:}S.T \qquad \Gamma \vdash t_2 \equiv s_2 : S}{\Gamma \vdash t_1\, t_2 \equiv s_1\, s_2 : [x \mapsto t_2]T} \quad \text{(Q-APP)}$$

$$\frac{\Gamma, x{:}S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x{:}S.t)\, s \equiv [x \mapsto s]t : [x \mapsto s]T} \quad \text{(Q-BETA)}$$

$$\frac{\Gamma \vdash t : \Pi x{:}S.T \qquad x \notin FV(t)}{\Gamma \vdash \lambda x{:}T.t\, x \equiv t : \Pi x{:}S.T} \quad \text{(Q-ETA)}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \equiv t : T} \quad \text{(Q-REFL)}$$

$$\frac{\Gamma \vdash t \equiv s : T}{\Gamma \vdash s \equiv t : T} \quad \text{(Q-SYM)}$$

$$\frac{\Gamma \vdash s \equiv u : T \qquad \Gamma \vdash u \equiv t : T}{\Gamma \vdash s \equiv t : T} \quad \text{(Q-TRANS)}$$

Strong Normalization

$$\frac{\mathtt{t}_1 \longrightarrow_\beta \mathtt{t}_1'}{\lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_1 \longrightarrow_\beta \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_1'} \qquad \text{(Beta-Abs)}$$

$$\frac{\mathtt{t}_1 \longrightarrow_\beta \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow_\beta \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(Beta-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow_\beta \mathtt{t}_2'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow_\beta \mathtt{t}_1\ \mathtt{t}_2'} \qquad \text{(Beta-App2)}$$

$$(\lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_1)\ \mathtt{t}_2 \longrightarrow_\beta [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_1 \qquad \text{(Beta-AppAbs)}$$

- **Reduction does not go inside the type labels of λ abstractions**
- **Theorem - The relation $\rightarrow_\beta$ is** strongly normalizing **on well-typed terms. More precisely, if $\Gamma \vdash \mathrm{t} : \mathrm{T}$ then there is no infinite sequence of terms $(\mathrm{t_i})_{\mathrm{i}\geq 1}$ such that $t = t_1$ and $\mathrm{t_i} \rightarrow_\beta t_{i+1}$ for $i \geq 1$.**

Algorithmic Typing and Equality

- Needed to be formulated closer to an algorithm
  - Syntax-directed rules (going from premises to conclusions)
- It is shown that the typechecking algorithm is sound, complete, and terminates on all inputs
  - This also demonstrates the decidability of the original judgments

- Theorem (Preservation) – If $\Gamma \vdash \mathrm{t} : \mathrm{T}$ and $\mathrm{t} \rightarrow_\beta \mathrm{t}'$, then $\Gamma \vdash \mathrm{t}' : \mathrm{T}$.

Dependent Sum Types

- $\Sigma x : T_1 . T_2$ (Σ-types)
- Generalize ordinary product types $(T_1 \times T_2)$
- If x does not appear in T$_2$
  - $\Sigma x : T_1 . T_2 \equiv T_1 \times T_2$
  - $\Pi x : T_1 . T_2 \equiv T_1 \rightarrow T_2$
- $(t , t \text{:} \Sigma x : T . T)$
  - Typed pair (annotated explicitly)
  - If $S : T \rightarrow *$ and $x : T$ and $y : S\ x$, then the pair (x , y) could have both $\Sigma z : T.\ S\ z$ and $\Sigma z : T.\ S\ x$ as a type.

Dependent Sum Types (cont.)

Extends λLF (2-1 and 2-2)

New syntax

$\mathtt{t}$	::=	...	*terms:*
		$(\mathtt{t},\ \mathtt{t}{:}\Sigma\mathtt{x}{:}\mathtt{T}.\mathtt{T})$	*typed pair*
		$\mathtt{t.1}$	*first projection*
		$\mathtt{t.2}$	*second projection*
$\mathtt{T}$	::=	...	*types:*
		$\Sigma\mathtt{x}{:}\mathtt{T}.\mathtt{T}$	*dependent sum type*

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma \vdash \mathtt{S} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{T} :: *}{\Gamma \vdash \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T} :: *} \quad \text{(K-SIGMA)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T} :: * \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma \vdash \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}}{\Gamma \vdash (\mathtt{t}_1,\mathtt{t}_2{:}\Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}) : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}} \quad \text{(T-PAIR)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}}{\Gamma \vdash \mathtt{t.1} : \mathtt{S}} \quad \text{(T-PROJ1)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}}{\Gamma \vdash \mathtt{t.2} : [\mathtt{x} \mapsto \mathtt{t.1}]\mathtt{T}} \quad \text{(T-PROJ2)}$$

Term Equivalence $\boxed{\Gamma \vdash \mathtt{t}_1 \equiv \mathtt{t}_2 : \mathtt{T}}$

$$\frac{\Gamma \vdash \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T} :: * \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma \vdash \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}}{\Gamma \vdash (\mathtt{t}_1,\mathtt{t}_2{:}\Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}).1 \equiv \mathtt{t}_1 : S} \quad \text{(Q-PROJ1)}$$

$$\frac{\Gamma \vdash \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T} :: * \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma \vdash \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}}{\Gamma \vdash (\mathtt{t}_1,\mathtt{t}_2{:}\Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}).2 \equiv \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}} \quad \text{(Q-PROJ2)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}}{\Gamma \vdash (\mathtt{t.1},\ \mathtt{t.2}{:}\Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}) \equiv \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}} \quad \text{(Q-SURJPAIR)}$$

The Calculus of Construction

- One of the most famous systems of dependent types
- A setting for all of constructive mathematics
- Simple and very expressive

Extends λLF (2-1 and 2-2)

New syntax

`t`	::=	...	*terms:*
		`all x:T.t`	*universal quantification*
`T`	::=	...	*types:*
		`Prop`	*propositions*
		`Prf`	*family of proofs*

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\Gamma \vdash \mathtt{Prop} :: * \qquad \text{(K-PROP)}$$

$$\Gamma \vdash \mathtt{Prf} :: \Pi \mathtt{x}{:}\mathtt{Prop}.\ * \qquad \text{(K-PRF)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{t} : \mathtt{Prop}}{\Gamma \vdash \mathtt{all}\ \mathtt{x}{:}\mathtt{T}.\mathtt{t} : \mathtt{Prop}} \qquad \text{(T-ALL)}$$

Type Equivalence $\boxed{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: K}$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{T} \vdash \mathtt{t} : \mathtt{Prop}}{\Gamma \vdash \mathtt{Prf}\ (\mathtt{all}\ \mathtt{x}{:}\mathtt{T}.\mathtt{t}) \equiv \Pi \mathtt{x}{:}\mathtt{T}.\mathtt{Prf}\ \mathtt{t} :: *} \qquad \text{(QT-ALL)}$$

The Calculus of Construction

Example –

nat = all a:Prop. all z:Prf a. all s: Prf a $\rightarrow$ Prf a. a

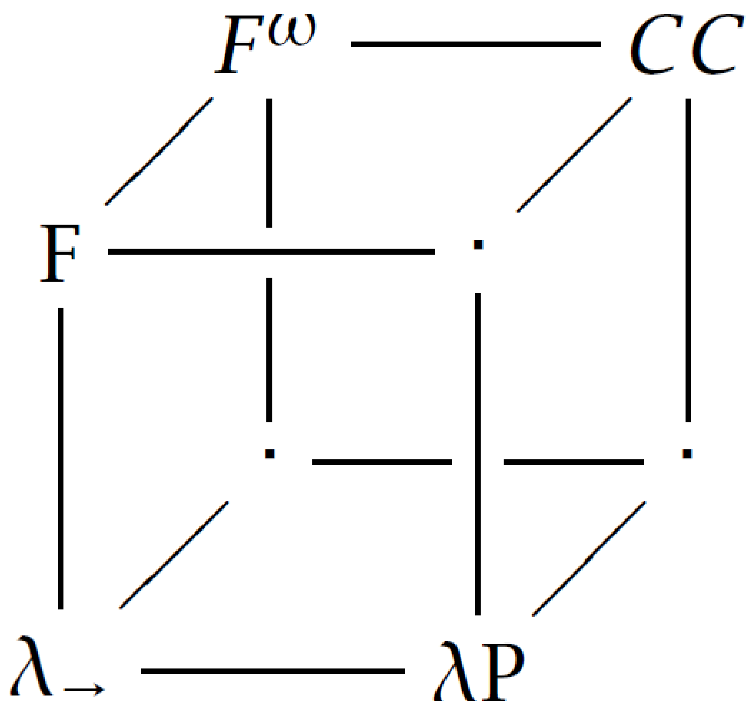
(nat is a member of type Prop)

zero = λa:Prop. λz:Prf a. λs:Prf a $\rightarrow$ Prf a. z : Prf nat

succ = λn:Prf nat. λa:Prop. λz:Prf a. λs:Prf a $\rightarrow$ Prf a.
s (n a z s) : Prf nat $\rightarrow$ Prf nat

add = λm:Nat. λn:Nat. m nat n succ : Prf nat $\rightarrow$ Prf nat $\rightarrow$ Prf nat

Lambda Cube



Twelf in Practice

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

plus: nat -> nat -> nat -> type.
p-z: plus z N N.
p-s: plus (s N1) N2 (s N3)
      <- plus N1 N2 N3.

> 2+1=3 : plus (s (s z)) (s z) (s (s (s z))) = p-s (p-s p-z).
```

http://twelf.plparty.org/live/

Programming with Dependent Types

Languages: Pebble, Cardelli's Quest, Cayenne, Dependent ML, Haskell, Agda

Example - a zip **function which can only be applied to a pair of lists of the same length**

```
datatype 'a list with nat =
    nil(0)
  | {n:nat} cons(n+1) 'a * 'a list(n)
```

The withtype **clause is a type annotation supplied by the programmer.**

```
fun zip ([], []) = []
  | zip (x :: xs, y :: ys) = (x, y) :: zip (xs, ys)
withtype {n:nat} => 'a list(n) * 'b list(n) -> ('a * 'b) list(n)
```

This leads to a safer and faster implementation of zip than a corresponding one in Standard ML.

Conclusion

- Dependent Types
  - Product Type and Sum Type
- Logical Framework LF and λLF
- Algorithms for Typechecking
- Programming in Twelf and DML

References

[1] B. Pierce "Advanced Topics in Types and Programming Languages," Chapter 2, pp. 45-86, MIT Press, 2004.

[2] B. Pierce "Types and Programming Languages," Chapter 29 and 30,, MIT Press, 2002.

[3] H. Xi, "Dependent Types in Practical Programming," PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1998.

[4] F. Pfenning and H. Xi, "Dependent Types in practical programming," In Proc. 26th ACM Symp. on Principles of Prog. Lang., pp 214-227, 1999.

[5] R. Harper, F. Honsell, and G. Plotkin, "A framework for defining logics," Journal of the ACM, vol. 40, pp. 143–184, 1993. Summary in IEEE Symposium on Logic in Computer Science (LICS), Ithaca, New York, 1987.

[6] F. Pfenning and C. Schürmann, "System description: Twelf - a meta-logical framework for deductive systems," International Conference on Automated Deduction, Springer, Berlin, Heidelberg, 1999.

[7] C. Schürmann, "The Twelf proof assistant." International Conference on Theorem Proving in Higher Order Logics. Springer, Berlin, Heidelberg, 2009.

[8] Twelf wiki, http://twelf.org/wiki

[9] DML homepage, https://www.cs.bu.edu/~hwxi/DML/DML.html

Recent Work

[1] R. Eisenberg, "Dependent Types in Haskell Theory and Practice," PhD Thesis, University of Pennsylvania, 2016.

[2] U. Norell, "Towards A Practical Programming Language based on Dependent Type Theory," PhD Thesis, Chalmers University of Technology, 2007.

[3] A. Nanevski, A. Banerjee, and D. Garg. "Dependent type theory for verification of information flow and access control policies." ACM Transactions on Programming Languages and Systems (TOPLAS) 35. no.2, 2013.

[4] De Moura, Leonardo, et al. "Elaboration in dependent type theory." arXiv preprint arXiv:1505.04324, 2015.

[5] R. Atkey, N. Ghani, and P. Johann. "A relationally parametric model of dependent type theory." ACM SIGPLAN Notices. Vol. 49. No. 1. ACM, 2014.

[6] Bizjak, Aleš, et al. "Guarded dependent type theory with coinductive types." International Conference on Foundations of Software Science and Computation Structures. Springer, Berlin, Heidelberg, 2016.

[7] A. Nuyts, A. Vezzosi, and D. Devriese. "Parametric quantifiers for dependent types.", KU Leuven University, 2017.

Any Questions?