Syntax and semantics of dependent types.

Cours MPRI catégories et lambda-calcul.

15 février 2010

Antoine Delignat-Lavaud
École Normale Supérieure de Cachan

Outline

1. **Dependent types**
   - Some motivations
   - The type system
   - Dependent function
   - Natural numbers
   - Dependent sum
   - Identity types
   - Universes

2. **Category-theoretic semantics**
   - Context morphisms
   - Categories with families
   - Interpretation

Dependent types

1. Types that depend on or vary with values.
2. Example : $Vec_\tau(M)$, type of vectors of length M
3. M is a value in the calculus
4. The dependancy is written $\Pi x : N. Vec_\tau(x)$
5. Benefits : types are more accurate (e.g. $N \to List(N)$)
6. More expressive static verification :
   $H : \Pi x : N. Vec_\tau(Suc(x)) \to \tau$
7. Programs on length dependent vectors must satisfy length constraints to type.
8. Another example : ordered vectors

Dependent types

1. Types that depend on or vary with values.
2. Example : $Vec_\tau(M)$, type of vectors of length M
3. M is a value in the calculus
4. The dependancy is written $\Pi x : N.Vec_\tau(x)$
5. Benefits : types are more accurate (e.g. $N \to List(N)$)
6. More expressive static verification :
   $H : \Pi x : N.Vec_\tau(Suc(x)) \to \tau$
7. Programs on length dependent vectors must satisfy length constraints to type.
8. Another example : ordered vectors

Dependent types

1. Types that depend on or vary with values.
2. Example : $Vec_\tau(M)$, type of vectors of length M
3. M is a value in the calculus
4. The dependancy is written $\Pi x : N. Vec_\tau(x)$
5. Benefits : types are more accurate (e.g. $N \rightarrow List(N)$)
6. More expressive static verification :
   $H : \Pi x : N. Vec_\tau(Suc(x)) \rightarrow \tau$
7. Programs on length dependent vectors must satisfy length constraints to type.
8. Another example : ordered vectors

Dependent types

1. Types that depend on or vary with values.
2. Example : $Vec_\tau(M)$, type of vectors of length M
3. M is a value in the calculus
4. The dependancy is written $\Pi x : N.Vec_\tau(x)$
5. Benefits : types are more accurate (e.g. $N \rightarrow List(N)$)
6. More expressive static verification :
   $H : \Pi x : N.Vec_\tau(Suc(x)) \rightarrow \tau$
7. Programs on length dependent vectors must satisfy length constraints to type.
8. Another example : ordered vectors

Dependent types

1. Types that depend on or vary with values.
2. Example : $Vec_\tau(M)$, type of vectors of length M
3. M is a value in the calculus
4. The dependancy is written $\Pi x : N. Vec_\tau(x)$
5. Benefits : types are more accurate (e.g. $N \to List(N)$)
6. More expressive static verification : $H : \Pi x : N. Vec_\tau(Suc(x)) \to \tau$
7. Programs on length dependent vectors must satisfy length constraints to type.
8. Another example : ordered vectors

Dependent types

1. Types that depend on or vary with values.
2. Example : $Vec_{\tau}(M)$, type of vectors of length M
3. M is a value in the calculus
4. The dependancy is written $\Pi x : N. Vec_{\tau}(x)$
5. Benefits : types are more accurate (e.g. $N \rightarrow List(N)$)
6. More expressive static verification :
   $H : \Pi x : N. Vec_{\tau}(Suc(x)) \rightarrow \tau$
7. Programs on length dependent vectors must satisfy length constraints to type.
8. Another example : ordered vectors

Dependent types

1. Types that depend on or vary with values.
2. Example : $Vec_\tau(M)$, type of vectors of length M
3. M is a value in the calculus
4. The dependancy is written $\Pi x : N. Vec_\tau(x)$
5. Benefits : types are more accurate (e.g. $N \rightarrow List(N)$)
6. More expressive static verification :
   $H : \Pi x : N. Vec_\tau(Suc(x)) \rightarrow \tau$
7. Programs on length dependent vectors must satisfy length constraints to type.
8. Another example : ordered vectors

Dependent types

1. Types that depend on or vary with values.
2. Example : $Vec_\tau(M)$, type of vectors of length M
3. M is a value in the calculus
4. The dependancy is written $\Pi x : N. Vec_\tau(x)$
5. Benefits : types are more accurate (e.g. $N \to List(N)$)
6. More expressive static verification :
   $H : \Pi x : N. Vec_\tau(Suc(x)) \to \tau$
7. Programs on length dependent vectors must satisfy length constraints to type.
8. Another example : ordered vectors

Type-checking

1. How to test equality of dependent types ?
2. Computation may be required $Vec_\tau(1)$, $Vec_\tau(0+0+1)$
3. Arbitrary dependance : typing is undecidable.
4. Built-in type equality

$\vdash \Gamma$ ctx	Γ is a valid context
$\Gamma \vdash \sigma$ type	σ is a valid type in context Γ
$\Gamma \vdash M : \sigma$	M is a term of type σ in context Γ
$\vdash \Gamma = \Delta$ ctx	Γ, Δ are definitionally equal contexts
$\Gamma \vdash \sigma = \tau$ type	σ, τ are definitionally equal types in context Γ
$\Gamma \vdash M = N : \sigma$	M, N are equal terms of type σ in context Γ

Type-checking

1. How to test equality of dependent types ?
2. Computation may be required $Vec_\tau(1)$, $Vec_\tau(0+0+1)$
3. Arbitrary dependance : typing is undecidable.
4. Built-in type equality

$\vdash \Gamma$ ctx	Γ is a valid context
$\Gamma \vdash \sigma$ type	σ is a valid type in context Γ
$\Gamma \vdash M : \sigma$	M is a term of type σ in context Γ
$\vdash \Gamma = \Delta$ ctx	Γ, Δ are definitionally equal contexts
$\Gamma \vdash \sigma = \tau$ type	σ, τ are definitionally equal types in context Γ
$\Gamma \vdash M = N : \sigma$	M, N are equal terms of type σ in context Γ

Type-checking

1. How to test equality of dependent types ?
2. Computation may be required $Vec_\tau(1)$, $Vec_\tau(0+0+1)$
3. Arbitrary dependance : typing is undecidable.
4. Built-in type equality

$\vdash \Gamma$ ctx	Γ is a valid context
$\Gamma \vdash \sigma$ type	σ is a valid type in context Γ
$\Gamma \vdash M : \sigma$	M is a term of type σ in context Γ
$\vdash \Gamma = \Delta$ ctx	Γ, Δ are definitionally equal contexts
$\Gamma \vdash \sigma = \tau$ type	σ, τ are definitionally equal types in context Γ
$\Gamma \vdash M = N : \sigma$	M, N are equal terms of type σ in context Γ

Type-checking

1. How to test equality of dependent types ?
2. Computation may be required $Vec_\tau(1)$, $Vec_\tau(0+0+1)$
3. Arbitrary dependance : typing is undecidable.
4. Built-in type equality

$\vdash \Gamma$ ctx	Γ is a valid context
$\Gamma \vdash \sigma$ type	σ is a valid type in context Γ
$\Gamma \vdash M : \sigma$	M is a term of type σ in context Γ
$\vdash \Gamma = \Delta$ ctx	Γ, Δ are definitionally equal contexts
$\Gamma \vdash \sigma = \tau$ type	σ, τ are definitionally equal types in context Γ
$\Gamma \vdash M = N : \sigma$	M, N are equal terms of type σ in context Γ

Type-checking

1. How to test equality of dependent types ?
2. Computation may be required $Vec_\tau(1)$, $Vec_\tau(0+0+1)$
3. Arbitrary dependance : typing is undecidable.
4. Built-in type equality

$\vdash \Gamma$ ctx	Γ is a valid context
$\Gamma \vdash \sigma$ type	σ is a valid type in context Γ
$\Gamma \vdash M : \sigma$	M is a term of type σ in context Γ
$\vdash \Gamma = \Delta$ ctx	Γ, Δ are definitionally equal contexts
$\Gamma \vdash \sigma = \tau$ type	σ, τ are definitionally equal types in context Γ
$\Gamma \vdash M = N : \sigma$	M, N are equal terms of type σ in context Γ

Syntax

$$
\begin{array}{lcl}
\Gamma & ::= & \emptyset \mid \Gamma, x : \sigma \\
\sigma, \tau & ::= & \Pi x : \sigma.\tau \mid \Sigma x : \sigma.\tau \mid Id_\sigma(M, N) \mid \mathbb{N} \\
M, N, H, P & ::= & x \mid \lambda x : \tau.M^\tau \mid App_{[x:\sigma]\tau}(M, N) \mid \\
 & & Pair_{[x:\sigma]\tau}(M, N) \mid R^{\Sigma}_{z:(\Sigma x:\sigma.\tau)\rho}([x : \sigma, y : \tau]H, M) \\
 & & \mid Refl_\sigma(M) \mid R^{Id}_{[x:\sigma, y:\sigma, p:Id_\sigma(x,y)]\tau}([z : \sigma]H, M, N, P) \\
 & & \mid 0 \mid Suc(M) \mid R^{\mathbb{N}}_{[n:\mathbb{N}]\sigma}(H_z, [n : \mathbb{N}, x : \sigma]H_s, M)
\end{array}
$$

Dependent function

Dependent product

1. A dependendent function $\Pi x : \sigma.\tau$ is interpreted as a cartesian product $\prod_{i \in I} B_i$.
2. Formation and equality rules as expected
3. Dependent functions can be eliminated with the dependnt application

$$\frac{\Gamma \vdash M : \Pi x : \sigma.\tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(M, N) : \tau[x \leftarrow N]}$$

Dependent function

Dependent product

1. A dependendent function $\Pi x : \sigma.\tau$ is interpreted as a cartesian product $\prod_{i \in I} B_i$.
2. Formation and equality rules as expected
3. Dependent functions can be eliminated with the dependnt application

$$\frac{\Gamma \vdash M : \Pi x : \sigma.\tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(M, N) : \tau[x \leftarrow N]}$$

Dependent function

Dependent product

1. A dependendent function $\Pi x : \sigma.\tau$ is interpreted as a cartesian product $\prod_{i \in I} B_i$.
2. Formation and equality rules as expected
3. Dependent functions can be eliminated with the dependnt application

$$\frac{\Gamma \vdash M : \Pi x : \sigma.\tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(M, N) : \tau[x \leftarrow N]}$$

Dependent function

Dependent product

1. A dependendent function $\Pi x : \sigma.\tau$ is interpreted as a cartesian product $\prod_{i \in I} B_i$.
2. Formation and equality rules as expected
3. Dependent functions can be eliminated with the dependnt application

$$\frac{\Gamma \vdash M : \Pi x : \sigma.\tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(M, N) : \tau[x \leftarrow N]}$$

Natural numbers

Natural numbers

1. We build numbers from 0 and Succ(M).
2. We use an eliminator $R^{\mathbb{N}}$ to substitute integers in types.
3. The eliminator tests both 0 and Succ(n)

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathbb{N} \\ \Gamma, n : \mathbb{N} \vdash \sigma \text{ type} \\ \Gamma \vdash H_z : \sigma[n \leftarrow 0] \\ \Gamma, n : \mathbb{N}, x : \sigma \vdash H_s : \sigma[n \leftarrow Suc(n)]\end{array}}{\Gamma \vdash R^{\mathbb{N}}_{[n:\mathbb{N}]\sigma}(H_z, [n : \mathbb{N}, x : \sigma]H_s, M) : \sigma[n \leftarrow M]}$$

Natural numbers

Natural numbers

1. We build numbers from 0 and Succ(M).
2. We use an eliminator $R^{\mathbb{N}}$ to substitute integers in types.
3. The eliminator tests both 0 and Succ(n)

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathbb{N} \\ \Gamma, n : \mathbb{N} \vdash \sigma \text{ type} \\ \Gamma \vdash H_z : \sigma[n \leftarrow 0] \\ \Gamma, n : \mathbb{N}, x : \sigma \vdash H_s : \sigma[n \leftarrow Suc(n)]\end{array}}{\Gamma \vdash R^{\mathbb{N}}_{[n:\mathbb{N}]\sigma}(H_z, [n : \mathbb{N}, x : \sigma]H_s, M) : \sigma[n \leftarrow M]}$$

Natural numbers

Natural numbers

1. We build numbers from 0 and Succ(M).
2. We use an eliminator $R^{\mathbb{N}}$ to substitute integers in types.
3. The eliminator tests both 0 and Succ(n)

$$\frac{\Gamma \vdash M : \mathbb{N} \quad \Gamma, n : \mathbb{N} \vdash \sigma \text{ type} \quad \Gamma \vdash H_z : \sigma[n \leftarrow 0] \quad \Gamma, n : \mathbb{N}, x : \sigma \vdash H_s : \sigma[n \leftarrow Suc(n)]}{\Gamma \vdash R^{\mathbb{N}}_{[n:\mathbb{N}]\sigma}(H_z, [n : \mathbb{N}, x : \sigma]H_s, M) : \sigma[n \leftarrow M]}$$

Natural numbers

Natural numbers

1. We build numbers from 0 and Succ(M).
2. We use an eliminator $R^{\mathbb{N}}$ to substitute integers in types.
3. The eliminator tests both 0 and Succ(n)

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathbb{N} \\ \Gamma, n : \mathbb{N} \vdash \sigma \text{ type} \\ \Gamma \vdash H_z : \sigma[n \leftarrow 0] \\ \Gamma, n : \mathbb{N}, x : \sigma \vdash H_s : \sigma[n \leftarrow Suc(n)]\end{array}}{\Gamma \vdash R^{\mathbb{N}}_{[n:\mathbb{N}]\sigma}(H_z, [n : \mathbb{N}, x : \sigma]H_s, M) : \sigma[n \leftarrow M]}$$

Dependent sum

Dependent sum

1. Set family $(B_i)_{i \in I}$, we define $\Sigma_{i \in I} B_i = \{(i, b) \mid i \in I \wedge b \in B_i\}$
2. Type of pairs : $Pair_{[x:\sigma]\tau}(M, N) : \Sigma x : \sigma.\tau$
3. For Σ-elimination, we use an eliminator R^{Σ}
4. R^{Σ} describes the behavior on pairs and serves as projection.

$$\frac{\Gamma \vdash M : \Sigma x : \sigma.\tau \qquad \Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow Pair_{x:\sigma.\tau}(x, y)] \qquad \Gamma, z : \Sigma x : \sigma.\tau \vdash \rho \text{ type}}{\Gamma \vdash R^{\Sigma}_{[z:\Sigma x:\sigma.\tau]\rho}([x : \sigma, y : \tau]H, M) : \rho[z \leftarrow M]}$$

Dependent sum

Dependent sum

1. Set family $(B_i)_{i \in I}$, we define $\Sigma_{i \in I} B_i = \{(i, b) \mid i \in I \wedge b \in B_i\}$
2. Type of pairs : $Pair_{[x:\sigma]\tau}(M, N) : \Sigma x : \sigma.\tau$
3. For Σ-elimination, we use an eliminator R^Σ
4. R^Σ describes the behavior on pairs and serves as projection.

$$\frac{\Gamma \vdash M : \Sigma x : \sigma.\tau \qquad \Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow Pair_{x:\sigma.\tau}(x, y)] \qquad \Gamma, z : \Sigma x : \sigma.\tau \vdash \rho \text{ type}}{\Gamma \vdash R^\Sigma_{[z:\Sigma x:\sigma.\tau]\rho}([x : \sigma, y : \tau]H, M) : \rho[z \leftarrow M]}$$

Dependent sum

Dependent sum

1. Set family $(B_i)_{i \in I}$, we define $\Sigma_{i \in I} B_i = \{(i, b) \mid i \in I \wedge b \in B_i\}$
2. Type of pairs : $Pair_{[x:\sigma]\tau}(M, N) : \Sigma x : \sigma.\tau$
3. For Σ-elimination, we use an eliminator R^{Σ}
4. R^{Σ} describes the behavior on pairs and serves as projection.

$$\frac{\begin{array}{c}\Gamma \vdash M : \Sigma x : \sigma.\tau \\ \Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow Pair_{x:\sigma.\tau}(x, y)] \\ \Gamma, z : \Sigma x : \sigma.\tau \vdash \rho \text{ type}\end{array}}{\Gamma \vdash R^{\Sigma}_{[z:\Sigma x:\sigma.\tau]\rho}([x : \sigma, y : \tau]H, M) : \rho[z \leftarrow M]}$$

Dependent sum

Dependent sum

1. Set family $(B_i)_{i \in I}$, we define $\Sigma_{i \in I} B_i = \{(i, b) \mid i \in I \wedge b \in B_i\}$
2. Type of pairs : $Pair_{[x:\sigma]\tau}(M, N) : \Sigma x : \sigma.\tau$
3. For Σ-elimination, we use an eliminator R^Σ
4. R^Σ describes the behavior on pairs and serves as projection.

$$\frac{\Gamma \vdash M : \Sigma x : \sigma.\tau \qquad \Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow Pair_{x:\sigma.\tau}(x, y)] \qquad \Gamma, z : \Sigma x : \sigma.\tau \vdash \rho \text{ type}}{\Gamma \vdash R^\Sigma_{[z:\Sigma x:\sigma.\tau]\rho}([x : \sigma, y : \tau]H, M) : \rho[z \leftarrow M]}$$

Dependent sum

Dependent sum

1. Set family $(B_i)_{i \in I}$, we define $\Sigma_{i \in I} B_i = \{(i, b) \mid i \in I \land b \in B_i\}$
2. Type of pairs : $Pair_{[x:\sigma]\tau}(M, N) : \Sigma x : \sigma.\tau$
3. For Σ-elimination, we use an eliminator R^{Σ}
4. R^{Σ} describes the behavior on pairs and serves as projection.

$$\frac{\begin{array}{c}\Gamma \vdash M : \Sigma x : \sigma.\tau \\ \Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow Pair_{x:\sigma.\tau}(x, y)] \\ \Gamma, z : \Sigma x : \sigma.\tau \vdash \rho \text{ type}\end{array}}{\Gamma \vdash R^{\Sigma}_{[z:\Sigma x:\sigma.\tau]\rho}([x : \sigma, y : \tau]H, M) : \rho[z \leftarrow M]}$$

Dependent sum : projections

Projections

$$M.1 = R^{\Sigma}_{[z:\Sigma x:\sigma.\tau]\sigma}([x:\sigma, y:\tau]x, M) : \sigma$$

$$M.2 = R^{\Sigma}_{[z:\Sigma x:\sigma.\tau]\tau[x\leftarrow z.1]}([x:\sigma, y:\tau]y, M) : \tau[M.1]$$

Identity types

Identity types

1. Address the problem of dependent type equality
2. For all the previous constructors, we define identity rules, such as :

$$\frac{\Gamma \vdash \lambda x : \sigma . M^{\tau} : \Pi x : \sigma . \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(\lambda x : \sigma . M^{\tau}, N) = M[x \leftarrow N] : \tau[x \leftarrow N]}$$

3. Equality is a judement outside the type theory
4. We introduce an identity constructor to have embedded equality.

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma}{\Gamma \vdash Id_{\sigma}(M, N) \text{ type}} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash Refl_{\sigma}(M) : Id_{\sigma}(M, M)}$$

Identity types

Identity types

1. Address the problem of dependent type equality
2. For all the previous constructors, we define identity rules, such as :

$$\frac{\Gamma \vdash \lambda x:\sigma.M^{\tau} : \Pi x:\sigma.\tau \quad \Gamma \vdash N:\sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(\lambda x:\sigma.M^{\tau}, N) = M[x \leftarrow N] : \tau[x \leftarrow N]}$$

3. Equality is a judement outside the type theory
4. We introduce an identity constructor to have embedded equality.

$$\frac{\Gamma \vdash M:\sigma \quad \Gamma \vdash N:\sigma}{\Gamma \vdash Id_{\sigma}(M,N)\ \text{type}} \qquad \frac{\Gamma \vdash M:\sigma}{\Gamma \vdash Refl_{\sigma}(M) : Id_{\sigma}(M,M)}$$

Identity types

Identity types

1. Address the problem of dependent type equality
2. For all the previous constructors, we define identity rules, such as :

$$\frac{\Gamma \vdash \lambda x : \sigma . M^{\tau} : \Pi x : \sigma . \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(\lambda x : \sigma . M^{\tau}, N) = M[x \leftarrow N] : \tau[x \leftarrow N]}$$

3. Equality is a judement outside the type theory
4. We introduce an identity constructor to have embedded equality.

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma}{\Gamma \vdash Id_{\sigma}(M, N) \text{ type}} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash Refl_{\sigma}(M) : Id_{\sigma}(M, M)}$$

Identity types

Identity types

1. Address the problem of dependent type equality
2. For all the previous constructors, we define identity rules, such as :

$$\frac{\Gamma \vdash \lambda x : \sigma . M^{\tau} : \Pi x : \sigma . \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(\lambda x : \sigma . M^{\tau}, N) = M[x \leftarrow N] : \tau[x \leftarrow N]}$$

3. Equality is a judement outside the type theory
4. We introduce an identity constructor to have embedded equality.

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma}{\Gamma \vdash Id_{\sigma}(M, N) \text{ type}} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash Refl_{\sigma}(M) : Id_{\sigma}(M, M)}$$

Identity types

Identity types

1. Address the problem of dependent type equality
2. For all the previous constructors, we define identity rules, such as :

$$\frac{\Gamma \vdash \lambda x : \sigma . M^{\tau} : \Pi x : \sigma . \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash App_{[x:\sigma]\tau}(\lambda x : \sigma . M^{\tau}, N) = M[x \leftarrow N] : \tau[x \leftarrow N]}$$

3. Equality is a judement outside the type theory
4. We introduce an identity constructor to have embedded equality.

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma}{\Gamma \vdash Id_{\sigma}(M, N) \text{ type}} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash Refl_{\sigma}(M) : Id_{\sigma}(M, M)}$$

Identity elimination

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma \quad \Gamma \vdash P : Id_\sigma(M, N) \qquad \Gamma, z : \sigma \vdash H : \tau[x \leftarrow z, y \leftarrow z, p \leftarrow Refl_\sigma(z)]}{\Gamma \vdash R^{Id}_{[x:\sigma, y:\sigma, p:Id_\sigma(x,y)]\tau}([z : \sigma]H, M, N, P) : \tau[x \leftarrow M, y \leftarrow N, p \leftarrow P]}$$

Universes

Universes

$$\frac{\vdash \Gamma \text{ ctx}}{\Gamma \vdash U \text{ type}} \qquad \frac{\Gamma \vdash M : U}{\Gamma \vdash El(M) \text{ type}}$$

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash T : U}{\Gamma \vdash \forall x : \sigma . T : U}$$

Context morphisms

Context morphisms

If Γ and $\Delta = x : \sigma_1 ... x_n : \sigma_n$ are valid contexts and $f = (M_1 ... M_n)$ is a sequence of syntactic terms, we say that f is a a context morphism from Γ to Δ, denoted $\Gamma \vdash f \Rightarrow \Delta$, if :

$$\Gamma \vdash M_1 : \sigma_1 \quad ... \quad \Gamma \vdash M_n : \sigma_n[x_i \leftarrow M_i, i \leq n]$$

Context-morphism substitution, up to renaming of variables, is denoted $\tau[\Delta \leftarrow f]$.

Categories with families

CwF

1. $\mathcal{C}$ category of semantic contexts and morphisms
2. For $\Gamma \in \mathcal{C}$, a collection $Ty_{\mathcal{C}}(\Gamma)$ of semantic types
3. For $\Gamma \in \mathcal{C}$ and $\sigma \in Ty_{\mathcal{C}}(\Gamma)$, a collection $Tm_{\mathcal{C}}(\Gamma, \sigma)$ of semantic terms

Example

Set has a CwF : sets are contexts, maps are morphisms, elements of $Ty_{Set}(\Gamma)$ are families of sets indexed over Γ, elements of $Tm_{Set}(\Gamma, \sigma)$, with $(\sigma_\gamma)_{\gamma \in \Gamma} \in Ty_{Set}(\Gamma)$, is an assignment of an element $M(\gamma)$ of σ_γ for all $\gamma \in \Gamma$.

Categories with families

CwF

1. $\mathcal{C}$ category of semantic contexts and morphisms
2. For $\Gamma \in \mathcal{C}$, a collection $Ty_{\mathcal{C}}(\Gamma)$ of semantic types
3. For $\Gamma \in \mathcal{C}$ and $\sigma \in Ty_{\mathcal{C}}(\Gamma)$, a collection $Tm_{\mathcal{C}}(\Gamma, \sigma)$ of semantic terms

Example

Set has a CwF : sets are contexts, maps are morphisms, elements of $Ty_{Set}(\Gamma)$ are families of sets indexed over Γ, elements of $Tm_{Set}(\Gamma, \sigma)$, with $(\sigma_\gamma)_{\gamma \in \Gamma} \in Ty_{Set}(\Gamma)$, is an assignment of an element $M(\gamma)$ of σ_γ for all $\gamma \in \Gamma$.

Categories with families

CwF

1. $\mathcal{C}$ category of semantic contexts and morphisms
2. For $\Gamma \in \mathcal{C}$, a collection $Ty_{\mathcal{C}}(\Gamma)$ of semantic types
3. For $\Gamma \in \mathcal{C}$ and $\sigma \in Ty_{\mathcal{C}}(\Gamma)$, a collection $Tm_{\mathcal{C}}(\Gamma, \sigma)$ of semantic terms

Example

Set has a CwF : sets are contexts, maps are morphisms, elements of $Ty_{Set}(\Gamma)$ are families of sets indexed over Γ, elements of $Tm_{Set}(\Gamma, \sigma)$, with $(\sigma_\gamma)_{\gamma \in \Gamma} \in Ty_{Set}(\Gamma)$, is an assignment of an element $M(\gamma)$ of σ_γ for all $\gamma \in \Gamma$.

Category of families of sets

We define the category of families of sets *Fam* with object pairs $B = (B^0, B^1)$ where B^0 is a set and $B^1 = (B^1_b)_{b \in B^0}$ is a family of sets indexed over B^0. A map is a pair (f^0, f^1) where $f^0 : B^0 \to C^0$ is a function and $f^1 = (f^1_b)_{b \in B^0}$.

Types and terms functor

$$\mathcal{F}(\Gamma) = (Ty(\Gamma), (Tm(\Gamma, \sigma))_{\sigma \in Ty(\Gamma)}) : \mathcal{C}^{op} \to Fam$$

Semantic type formers

1. $App_{\sigma,\tau}(\lambda_{\sigma,\tau}(M), N) = M\{\bar{N}\}$
2. $Pi(\sigma,\tau)\{f\} = \Pi(\sigma\{f\}, \tau\{q(f,\sigma)\}) \in Ty(B)$
3. $\lambda_{\sigma,\tau}(M)\{f\} = \lambda_{q\{f\},\tau\{q(f,\sigma)\}}(M\{q(f,\sigma)\})$
4. $App_{\sigma,\tau}(M,N)\{f\} = App_{\sigma\{f\},\tau\{q(f,\sigma)\}}(M\{f\}, N\{f\})$

Semantic type formers

1. $App_{\sigma,\tau}(\lambda_{\sigma,\tau}(M), N) = M\{\bar{N}\}$
2. $Pi(\sigma, \tau)\{f\} = \Pi(\sigma\{f\}, \tau\{q(f, \sigma)\}) \in Ty(B)$
3. $\lambda_{\sigma,\tau}(M)\{f\} = \lambda_{q\{f\},\tau\{q(f,\sigma)\}}(M\{q(f, \sigma)\})$
4. $App_{\sigma,\tau}(M, N)\{f\} = App_{\sigma\{f\},\tau\{q(f,\sigma)\}}(M\{f\}, N\{f\})$

Semantic type formers

1. $App_{\sigma,\tau}(\lambda_{\sigma,\tau}(M), N) = M\{\bar{N}\}$
2. $Pi(\sigma,\tau)\{f\} = \Pi(\sigma\{f\}, \tau\{q(f,\sigma)\}) \in Ty(B)$
3. $\lambda_{\sigma,\tau}(M)\{f\} = \lambda_{q\{f\},\tau\{q(f,\sigma)\}}(M\{q(f,\sigma)\})$
4. $App_{\sigma,\tau}(M,N)\{f\} = App_{\sigma\{f\},\tau\{q(f,\sigma)\}}(M\{f\}, N\{f\})$

Semantic type formers

1. $App_{\sigma,\tau}(\lambda_{\sigma,\tau}(M), N) = M\{\bar{N}\}$
2. $Pi(\sigma,\tau)\{f\} = \Pi(\sigma\{f\}, \tau\{q(f,\sigma)\}) \in Ty(B)$
3. $\lambda_{\sigma,\tau}(M)\{f\} = \lambda_{q\{f\},\tau\{q(f,\sigma)\}}(M\{q(f,\sigma)\})$
4. $App_{\sigma,\tau}(M, N)\{f\} = App_{\sigma\{f\},\tau\{q(f,\sigma)\}}(M\{f\}, N\{f\})$

Interpretation

Interpretation

We define the interpretation by induction on the length of the syntactic contexts, types and terms.

1. [] maps pre-contexts to objects of $\mathcal{C}$.
2. Pairs $\Gamma; \sigma$ to families in $Ty([\Gamma])$.
3. Pairs $\Gamma; M$ to elements of $Tm(\sigma)$ for some $\sigma \in Ty([\Gamma])$

$$[\Gamma; x : \sigma] = [\Gamma].[\Gamma; \sigma] \text{ if } x \notin \Gamma$$

$$[\Gamma; \Pi x : \sigma.\tau] = \Pi([\Gamma; \sigma], [\Gamma, x : \sigma; \tau])$$

$$[\Gamma; x : \sigma, \Delta, y : \tau; x] = [\Gamma, x : \sigma, \Delta; x]\{p([\Gamma, x : \sigma, \Delta; \tau])\}$$

$$[\Gamma; App_{[x:\sigma]\tau}(M, N)] = App_{[\Gamma,\sigma],[\Gamma,x:\sigma.\tau]} \circ \langle [\Gamma; M], [\Gamma; N] \rangle_{[\Gamma;\Pi x:\sigma.\tau]}$$

$$[\Gamma; \lambda x : \sigma.M^\tau] = \lambda_{[\Gamma;\sigma],[\Gamma,x:\sigma;\tau]}([\Gamma, x : \sigma; M])$$

Interpretation

Interpretation

We define the interpretation by induction on the length of the syntactic contexts, types and terms.

1. [] maps pre-contexts to objects of $\mathcal{C}$.
2. Pairs $\Gamma; \sigma$ to families in $Ty([\Gamma])$.
3. Pairs $\Gamma; M$ to elements of $Tm(\sigma)$ for some $\sigma \in Ty([\Gamma])$

$$[\Gamma; x : \sigma] = [\Gamma].[\Gamma; \sigma] \text{ if } x \notin \Gamma$$

$$[\Gamma; \Pi x : \sigma.\tau] = \Pi([\Gamma; \sigma], [\Gamma, x : \sigma; \tau])$$

$$[\Gamma; x : \sigma, \Delta, y : \tau; x] = [\Gamma, x : \sigma, \Delta; x]\{p([\Gamma, x : \sigma, \Delta; \tau])\}$$

$$[\Gamma; App_{[x:\sigma]\tau}(M, N)] = App_{[\Gamma,\sigma],[\Gamma,x:\sigma.\tau]} \circ \langle [\Gamma; M], [\Gamma; N] \rangle_{[\Gamma;\Pi x:\sigma.\tau]}$$

$$[\Gamma; \lambda x : \sigma.M^{\tau}] = \lambda_{[\Gamma;\sigma],[\Gamma,x:\sigma;\tau]}([\Gamma, x : \sigma; M])$$

Interpretation

Interpretation

We define the interpretation by induction on the length of the syntactic contexts, types and terms.

1. [] maps pre-contexts to objects of $\mathcal{C}$.
2. Pairs $\Gamma; \sigma$ to families in $Ty([\Gamma])$.
3. Pairs $\Gamma; M$ to elements of $Tm(\sigma)$ for some $\sigma \in Ty([\Gamma])$

$$[\Gamma; x : \sigma] = [\Gamma].[\Gamma; \sigma] \text{ if } x \notin \Gamma$$

$$[\Gamma; \Pi x : \sigma.\tau] = \Pi([\Gamma; \sigma], [\Gamma, x : \sigma; \tau])$$

$$[\Gamma; x : \sigma, \Delta, y : \tau; x] = [\Gamma, x : \sigma, \Delta; x]\{p([\Gamma, x : \sigma, \Delta; \tau])\}$$

$$[\Gamma; App_{[x:\sigma]\tau}(M, N)] = App_{[\Gamma,\sigma],[\Gamma,x:\sigma.\tau]} \circ \langle [\Gamma; M], [\Gamma; N] \rangle_{[\Gamma;\Pi x:\sigma.\tau]}$$

$$[\Gamma; \lambda x : \sigma.M^{\tau}] = \lambda_{[\Gamma;\sigma],[\Gamma,x:\sigma;\tau]}([\Gamma, x : \sigma; M])$$

Interpretation

Interpretation

We define the interpretation by induction on the length of the syntactic contexts, types and terms.

1. $[]$ maps pre-contexts to objects of $\mathcal{C}$.
2. Pairs $\Gamma; \sigma$ to families in $Ty([\Gamma])$.
3. Pairs $\Gamma; M$ to elements of $Tm(\sigma)$ for some $\sigma \in Ty([\Gamma])$

$$[\Gamma; x : \sigma] = [\Gamma].[\Gamma; \sigma] \text{ if } x \notin \Gamma$$

$$[\Gamma; \Pi x : \sigma.\tau] = \Pi([\Gamma; \sigma], [\Gamma, x : \sigma; \tau])$$

$$[\Gamma; x : \sigma, \Delta, y : \tau; x] = [\Gamma, x : \sigma, \Delta; x]\{p([\Gamma, x : \sigma, \Delta; \tau])\}$$

$$[\Gamma; App_{[x:\sigma]\tau}(M, N)] = App_{[\Gamma,\sigma],[\Gamma,x:\sigma.\tau]} \circ \langle [\Gamma; M], [\Gamma; N] \rangle_{[\Gamma;\Pi x:\sigma.\tau]}$$

$$[\Gamma; \lambda x : \sigma.M^{\tau}] = \lambda_{[\Gamma;\sigma],[\Gamma,x:\sigma;\tau]}([\Gamma, x : \sigma; M])$$