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: \( \text{Vec}_\tau(M) \), type of vectors of length \( M \)
3. \( M \) is a value in the calculus
4. The dependancy is written \( \Pi x : N. \text{Vec}_\tau(x) \)
5. Benefits: types are more accurate (e.g. \( N \rightarrow \text{List}(N) \))
6. More expressive static verification:
   \( H : \Pi x : N. \text{Vec}_\tau(\text{Suc}(x)) \rightarrow \tau \)
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: $\text{Vec}_\tau(M)$, type of vectors of length $M$
3. $M$ is a value in the calculus
4. The dependancy is written $\Pi x : N. \text{Vec}_\tau(x)$
5. Benefits: types are more accurate (e.g. $N \to \text{List}(N)$)
6. More expressive static verification:
   
   $H : \Pi x : N. \text{Vec}_\tau(Suc(x)) \to \tau$
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: $\text{Vec}_\tau(M)$, type of vectors of length $M$
3. $M$ is a value in the calculus
4. The dependency is written $\Pi x : N. \text{Vec}_\tau(x)$
5. Benefits: types are more accurate (e.g. $N \rightarrow \text{List}(N)$)
6. More expressive static verification:
   $$H : \Pi x : N. \text{Vec}_\tau(Suc(x)) \rightarrow \tau$$
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: $\text{Vec}_\tau(M)$, type of vectors of length $M$
3. $M$ is a value in the calculus
4. The dependency is written $\Pi x : N. \text{Vec}_\tau(x)$
5. Benefits: types are more accurate (e.g. $N \rightarrow \text{List}(N)$)
6. More expressive static verification:
   $H : \Pi x : N. \text{Vec}_\tau(\text{Suc}(x)) \rightarrow \tau$
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: $\text{Vec}_\tau(M)$, type of vectors of length $M$
3. $M$ is a value in the calculus
4. The dependency is written $\prod x : N. \text{Vec}_\tau(x)$
5. Benefits: types are more accurate (e.g. $N \rightarrow \text{List}(N)$)
6. More expressive static verification:
   $H : \prod x : N. \text{Vec}_\tau(\text{Suc}(x)) \rightarrow \tau$
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: $\text{Vec}_\tau(M)$, type of vectors of length $M$
3. $M$ is a value in the calculus
4. The dependency is written $\Pi x : N. \text{Vec}_\tau(x)$
5. Benefits: types are more accurate (e.g. $N \rightarrow \text{List}(N)$)
6. More expressive static verification:
   
   $$H : \Pi x : N. \text{Vec}_\tau(Suc(x)) \rightarrow \tau$$

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: $\text{Vec}_\tau(M)$, type of vectors of length $M$
3. $M$ is a value in the calculus
4. The dependency is written $\Pi x : N. \text{Vec}_\tau(x)$
5. Benefits: types are more accurate (e.g. $N \rightarrow \text{List}(N)$)
6. More expressive static verification:
   \[ H : \Pi x : N. \text{Vec}_\tau(\text{Suc}(x)) \rightarrow \tau \]
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: \( \text{Vec}_\tau(M) \), type of vectors of length \( M \)
3. \( M \) is a value in the calculus
4. The dependency is written \( \Pi x : N. \text{Vec}_\tau(x) \)
5. Benefits: types are more accurate (e.g. \( N \rightarrow \text{List}(N) \))
6. More expressive static verification:
   \[ H : \Pi x : N. \text{Vec}_\tau(Suc(x)) \rightarrow \tau \]
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 $\text{Vec}_\tau(1)$, $\text{Vec}_\tau(0 + 0 + 1)$
3. Arbitrary dependance: typing is undecidable.
4. Built-in type equality

<table>
<thead>
<tr>
<th>⊢ Γ ctx</th>
<th>Γ is a valid context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ σ type</td>
<td>σ is a valid type in context Γ</td>
</tr>
<tr>
<td>Γ ⊢ M : σ</td>
<td>M is a term of type σ in context Γ</td>
</tr>
<tr>
<td>⊢ Γ = Δ ctx</td>
<td>Γ, Δ are definitionally equal contexts</td>
</tr>
<tr>
<td>Γ ⊢ σ = τ type</td>
<td>σ, τ are definitionally equal types in context Γ</td>
</tr>
<tr>
<td>Γ ⊢ M = N : σ</td>
<td>M, N are equal terms of type σ in context Γ</td>
</tr>
</tbody>
</table>
Type-checking

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

<table>
<thead>
<tr>
<th>⊢ Γ ctx</th>
<th>Γ is a valid context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ σ type</td>
<td>σ is a valid type in context Γ</td>
</tr>
<tr>
<td>Γ ⊢ M : σ</td>
<td>M is a term of type σ in context Γ</td>
</tr>
<tr>
<td>⊢ Γ = Δ ctx</td>
<td>Γ, Δ are definitionally equal contexts</td>
</tr>
<tr>
<td>Γ ⊢ σ = τ type</td>
<td>σ, τ are definitionally equal types in context Γ</td>
</tr>
<tr>
<td>Γ ⊢ M = N : σ</td>
<td>M, N are equal terms of type σ in context Γ</td>
</tr>
</tbody>
</table>
Type-checking

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

\[
\begin{align*}
\vdash \Gamma \text{ ctx} & \quad \Gamma \text{ is a valid context} \\
\Gamma \vdash \sigma \text{ type} & \quad \sigma \text{ is a valid type in context } \Gamma \\
\Gamma \vdash M : \sigma & \quad M \text{ is a term of type } \sigma \text{ in context } \Gamma \\
\vdash \Gamma = \Delta \text{ ctx} & \quad \Gamma, \Delta \text{ are definitionally equal contexts} \\
\Gamma \vdash \sigma = \tau \text{ type} & \quad \sigma, \tau \text{ are definitionally equal types in context } \Gamma \\
\Gamma \vdash M = N : \sigma & \quad M, N \text{ are equal terms of type } \sigma \text{ in context } \Gamma
\end{align*}
\]
### Type-checking

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

| \( \vdash \Gamma \text{ ctx} \) | \( \Gamma \) is a valid context |
| \( \Gamma \vdash \sigma \text{ type} \) | \( \sigma \) is a valid type in context \( \Gamma \) |
| \( \Gamma \vdash M : \sigma \) | \( M \) is a term of type \( \sigma \) in context \( \Gamma \) |
| \( \vdash \Gamma = \Delta \text{ ctx} \) | \( \Gamma, \Delta \) are definitionally equal contexts |
| \( \Gamma \vdash \sigma = \tau \text{ type} \) | \( \sigma, \tau \) are definitionally equal types in context \( \Gamma \) |
| \( \Gamma \vdash M = N : \sigma \) | \( M, N \) are equal terms of type \( \sigma \) in context \( \Gamma \) |
### Type-checking

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

<table>
<thead>
<tr>
<th>Type-checking Conditions</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \Gamma \text{ ctx}$</td>
<td>$\Gamma$ is a valid context</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \text{ type}$</td>
<td>$\sigma$ is a valid type in context $\Gamma$</td>
</tr>
<tr>
<td>$\Gamma \vdash M : \sigma$</td>
<td>$M$ is a term of type $\sigma$ in context $\Gamma$</td>
</tr>
<tr>
<td>$\Gamma \vdash \Gamma = \Delta \text{ ctx}$</td>
<td>$\Gamma, \Delta$ are definitionally equal contexts</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma = \tau \text{ type}$</td>
<td>$\sigma, \tau$ are definitionally equal types in context $\Gamma$</td>
</tr>
<tr>
<td>$\Gamma \vdash M = N : \sigma$</td>
<td>$M, N$ are equal terms of type $\sigma$ in context $\Gamma$</td>
</tr>
</tbody>
</table>
Syntax

\[
\begin{align*}
\Gamma & ::= \emptyset | \Gamma, x : \sigma \\
\sigma, \tau & ::= \Pi x : \sigma. \tau | \Sigma x : \sigma. \tau | \text{Id}_\sigma(M, N) | \mathbb{N} \\
M, N, H, P & ::= \text{Pair}_{\sigma \tau}^\chi(M, N) | R_{\Sigma z : (\Sigma x : \sigma. \tau) \rho}^{\Sigma}(\text{Pair}_{\sigma \tau}^\chi(M, N) | \text{Refl}_\sigma(M) | R_{\chi : \sigma, y : \tau, p : \text{Id}_\sigma(x, y)}^{\text{Id}}([\text{Pair}_{\sigma \tau}^\chi(M, N) \| 0 | \text{Suc}(M) | R_{n : \mathbb{N}}^{\text{Id}}(H_z, [n : \mathbb{N}, x : \sigma]H_s, M)
\end{align*}
\]

Outline

Dependent types
Some motivations
The type system
Dependent function
Natural numbers
Dependent sum
Identity types
Universes
Category-theoretic semantics
Context morphisms
Categories with families
Interpretation
Dependent function

Dependent product

1. A dependent 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 dependent application.

$$\Gamma \vdash M : \Pi x : \sigma.\tau \quad \Gamma \vdash N : \sigma$$

$$\Gamma \vdash App_{[x : \sigma]}\tau(M, N) : \tau[x \leftarrow N]$$
Dependent function

A dependent function $\prod x : \sigma . \tau$ is interpreted as a cartesian product $\prod_{i \in I} B_i$.

Formation and equality rules as expected

Dependent functions can be eliminated with the dependent application

$$\Gamma \vdash M : \prod x : \sigma . \tau \quad \Gamma \vdash N : \sigma$$

$$\Gamma \vdash App_{[x : \sigma]}(M, N) : \tau[x \leftarrow N]$$
Dependent function

Dependent product

1. A dependent 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 dependent application

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

1. A dependent 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 dependent application

\[
\Gamma \vdash M : \Pi x : \sigma.\tau \quad \Gamma \vdash N : \sigma \\
\hline
\Gamma \vdash App[\chi:\sigma]_\tau(M, N) : \tau[\chi \leftarrow N]
\]
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)

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

\[
\begin{align*}
\Gamma \vdash M : \mathbb{N} \\
\Gamma, n : \mathbb{N} \vdash \sigma \text{ type} \\
\Gamma \vdash H_z : \sigma[n \leftarrow 0] \\
\Gamma, n : \mathbb{N}, x : \sigma \vdash H_s : \sigma[n \leftarrow \text{Suc}(n)] \\
\Gamma \vdash R^\mathbb{N}_{[n:\mathbb{N}]\sigma}(H_z, [n : \mathbb{N}, x : \sigma]H_s, M) : \sigma[n \leftarrow M]
\end{align*}
\]
Natural numbers

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

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

\[
\Gamma \vdash M : \mathbb{N} \\
\Gamma, n : \mathbb{N} \vdash \sigma \text{ type} \\
\Gamma \vdash H_z : \sigma[n \leftarrow 0] \\
\Gamma, n : \mathbb{N}, x : \sigma \vdash H_s : \sigma[n \leftarrow \text{Suc}(n)] \\
\Gamma \vdash R^\mathbb{N}_{[n:\mathbb{N}]}(H_z, [n : \mathbb{N}, x : \sigma]H_s, M) : \sigma[n \leftarrow M]
\]
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: \(\text{Pair}_{[x : \sigma \tau]}(M, N) : \Sigma x : \sigma . \tau\)

3. For \(\Sigma\)-elimination, we use an eliminator \(R^\Sigma\)

4. \(R^\Sigma\) describes the behavior on pairs and serves as projection.

\[\Gamma \vdash M : \Sigma x : \sigma . \tau\]
\[\Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow \text{Pair}_{x : \sigma . \tau}(x, y)]\]
\[\Gamma, z : \Sigma x : \sigma . \tau \vdash \rho \text{ type}\]
\[\Gamma \vdash R^\Sigma_{[z : \Sigma x : \sigma . \tau] \rho}([x : \sigma, y : \tau]H, M) : \rho[z \leftarrow M]\]
Dependent sum

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: \(\text{Pair}_{[x : \sigma]}(M, N) : \Sigma x : \sigma.\tau\)

3. For \(\Sigma\)-elimination, we use an eliminator \(R^\Sigma\)

4. \(R^\Sigma\) describes the behavior on pairs and serves as projection.

\[\Gamma \vdash M : \Sigma x : \sigma.\tau\]
\[
\Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow \text{Pair}_{x : \sigma} (x, y)]
\]
\[
\Gamma, z : \Sigma x : \sigma.\tau \vdash \rho \text{ type}
\]
\[\Gamma \vdash R^\Sigma_{[z : \Sigma x : \sigma.\tau] \rho}([x : \sigma, y : \tau] H, M) : \rho[z \leftarrow M]\]
Dependent sum

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 \(\Sigma\)-elimination, we use an eliminator \(R^\Sigma\)

4. \(R^\Sigma\) describes the behavior on pairs and serves as projection.

\[
\begin{align*}
\Gamma \vdash M : \Sigma x : \sigma.\tau \\
\Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow Pair_{x : \sigma.\tau}(x, y)] \\
\Gamma, z : \Sigma x : \sigma.\tau \vdash \rho \text{ type} \\
\hline
\Gamma \vdash R^\Sigma_{[z : \Sigma x : \sigma.\tau]\rho}([x : \sigma, y : \tau]H, M) : \rho[z \leftarrow M]
\end{align*}
\]
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 : \(\text{Pair}_{[x:\sigma] \tau}(M, N) : \Sigma x : \sigma. \tau\)

3. For \(\Sigma\)-elimination, we use an eliminator \(R^\Sigma\)

4. \(R^\Sigma\) describes the behavior on pairs and serves as projection.

\[
\begin{align*}
\Gamma & \vdash M : \Sigma x : \sigma. \tau \\
\Gamma, x : \sigma, y : \tau & \vdash H : \rho[z \leftarrow \text{Pair}_{x:\sigma. \tau}(x, y)] \\
\Gamma, z : \Sigma x : \sigma. \tau & \vdash \rho \text{ type}
\end{align*}
\]

\[
\Gamma \vdash R^\Sigma_{[z:\Sigma x:\sigma. \tau] \rho}([x: \sigma, y: \tau]H, M) : \rho[z \leftarrow M]
\]
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: \(\text{Pair}_{[x : \sigma]} (M, N) : \Sigma x : \sigma. \tau\)

3. For \(\Sigma\)-elimination, we use an eliminator \(R^\Sigma\)

4. \(R^\Sigma\) describes the behavior on pairs and serves as projection.

\[
\Gamma \vdash M : \Sigma x : \sigma. \tau \\
\Gamma, x : \sigma, y : \tau \vdash H : \rho[z \leftarrow \text{Pair}_{x : \sigma. \tau} (x, y)] \\
\Gamma, z : \Sigma x : \sigma. \tau \vdash \rho \text{ type} \\
\Gamma \vdash R^\Sigma_{[z : \Sigma x : \sigma. \tau] \rho} ([x : \sigma, y : \tau] H, M) : \rho[z \leftarrow M]
\]
Dependent sum: projections

Projections

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

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

\[
\Gamma \vdash \lambda x : \sigma. M^\tau : \prod x : \sigma.\tau \quad \Gamma \vdash N : \sigma
\]

\[
\Gamma \vdash \text{App}_{[x:\sigma]\tau}(\lambda x : \sigma. M^\tau, N) = M[x \leftarrow N] : \tau[x \leftarrow N]
\]

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

\[
\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma
\]

\[
\Gamma \vdash \text{Id}_\sigma(M, N) \text{ type}
\]

\[
\Gamma \vdash \text{Refl}_\sigma(M) : \text{Id}_\sigma(M, M)
\]
Identity types

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

\[ \Gamma \vdash \lambda x : \sigma. M^\tau : \Pi x : \sigma. \tau \quad \Gamma \vdash N : \sigma \]

\[ \Gamma \vdash \text{App}_{[x:\sigma]^{\tau}}(\lambda x : \sigma. M^\tau, N) = M[x ← N] : \tau[x ← N] \]

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

\[ \Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma \]

\[ \Gamma \vdash \text{Id}_\sigma(M, N) \text{ type} \]

\[ \Gamma \vdash \text{Refl}_\sigma(M) : \text{Id}_\sigma(M, M) \]
Identity types

1. Address the problem of dependent type equality
2. For all the previous constructors, we define identity rules, such as:
   \[\Gamma \vdash \lambda x : \sigma. M^\tau : \prod x : \sigma. \tau \quad \Gamma \vdash N : \sigma\]
   \[\Gamma \vdash \text{App}_{[x : \sigma]}(\lambda x : \sigma. M^\tau, N) = M[x \leftarrow N] : \tau[x \leftarrow N]\]
3. Equality is a judgement outside the type theory
4. We introduce an identity constructor to have embedded equality.

\[\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma\]
\[\Gamma \vdash \text{Id}_\sigma(M, N) \text{ type}\]
\[\Gamma \vdash M : \sigma\]
\[\Gamma \vdash \text{Refl}_\sigma(M) : \text{Id}_\sigma(M, M)\]
Identity types

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

\[
\Gamma \vdash \lambda x : \sigma. M^\tau : \prod x : \sigma. \tau \quad \Gamma \vdash N : \sigma
\]

\[
\Gamma \vdash \text{App}_{[x: \sigma]}(\lambda x : \sigma. M^\tau, N) = M[x \leftarrow N] : \tau[x \leftarrow N]
\]

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

\[
\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma \\
\Gamma \vdash Id_{\sigma}(M, N) \text{ type}
\]

\[
\Gamma \vdash M : \sigma \\
\Gamma \vdash \text{Refl}_{\sigma}(M) : Id_{\sigma}(M, M)
\]
Identity types

1. Address the problem of dependent type equality
2. For all the previous constructors, we define identity rules, such as:
   \[\Gamma \vdash \lambda x : \sigma. M^\tau : \Pi x : \sigma. \tau \quad \Gamma \vdash N : \sigma\]
   \[\Gamma \vdash \text{App}_{[x : \sigma]_\tau}(\lambda x : \sigma. M^\tau, N) = M[x \leftarrow N] : \tau[x \leftarrow N]\]
3. Equality is a judgement outside the type theory
4. We introduce an identity constructor to have embedded equality.

\[\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma\]
\[\Gamma \vdash \text{Id}_\sigma(M, N) : \text{type}\]
\[\Gamma \vdash M : \sigma\]
\[\Gamma \vdash \text{Refl}_\sigma(M) : \text{Id}_\sigma(M, M)\]
Identity elimination

\[
\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \sigma \quad \Gamma \vdash P : \text{Id}_\sigma(M, N)
\]
\[
\Gamma, z : \sigma \vdash H : \tau[x \leftarrow z, y \leftarrow z, p \leftarrow \text{Refl}_\sigma(z)]
\]
\[
\Gamma \vdash R^{\text{Id}}_{[x: \sigma, y: \sigma, p: \text{Id}_\sigma(x, y)]\tau}([z : \sigma]H, M, N, P) : \tau[x \leftarrow M, y \leftarrow N, p \leftarrow P]
\]
Universes

\[\Gamma \vdash U \text{ type} \quad \Gamma \vdash M : U \quad \Gamma \vdash \text{El}(M) \text{ type} \]

\[\Gamma \vdash \sigma \text{ type} \]

\[\Gamma \vdash \forall x : \sigma. T : U\]
Context morphisms

If $\Gamma$ 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 context morphism from $\Gamma$ to $\Delta$, 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 $\text{Ty}_\mathcal{C}(\Gamma)$ of semantic types
3. For $\Gamma \in \mathcal{C}$ and $\sigma \in \text{Ty}_\mathcal{C}(\Gamma)$, a collection $\text{Tm}_\mathcal{C}(\Gamma, \sigma)$ of semantic terms

Example

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

<table>
<thead>
<tr>
<th>CwF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $\mathcal{C}$ category of semantic contexts and morphisms</td>
</tr>
<tr>
<td>2. For $\Gamma \in \mathcal{C}$, a collection $\mathcal{Ty}_\mathcal{C}(\Gamma)$ of semantic types</td>
</tr>
<tr>
<td>3. For $\Gamma \in \mathcal{C}$ and $\sigma \in \mathcal{Ty}<em>\mathcal{C}(\Gamma)$, a collection $\mathcal{Tm}</em>\mathcal{C}(\Gamma, \sigma)$ of semantic terms</td>
</tr>
</tbody>
</table>

### Example

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

Outline

- Dependent types
  - Some motivations
  - The type system
  - Dependent function
  - Natural numbers
  - Dependent sum
  - Identity types
  - Universes
- Category-theoretic semantics
  - Context morphisms
- Interpretation
Categories with families

CwF

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

Example

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

We define the category of families of sets $\textit{Fam}$ 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 $f^0 : B^0 \rightarrow C^0$ is a function and $f^1 = (f^1_b)_{b \in B^0}$.

Types and terms functor

$$\mathcal{F}(\Gamma) = (Ty(\Gamma), (Tm(\Gamma, \sigma))_{\sigma \in Ty(\Gamma)}): C^{op} \rightarrow \textit{Fam}$$
Syntax and semantics of dependent types.

Antoine Delignat-Lavaud

Outline

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

Category-theoretic semantics
- Context morphisms
- Categories with families
- Interpretation

Semantic type formers

1. \( \text{App}_{\sigma, \tau}(\lambda_{\sigma, \tau}(M), N) = M\{\tilde{N}\} \)
2. \( \text{Pi}(\sigma, \tau)\{f\} = \prod(\sigma\{f\}, \tau\{q(f, \sigma)\}) \in \text{Ty}(B) \)
3. \( \lambda_{\sigma, \tau}(M)\{f\} = \lambda_{q\{f\}, \tau\{q(f, \sigma)\}}(M\{q(f, \sigma)\}) \)
4. \( \text{App}_{\sigma, \tau}(M, N)\{f\} = \text{App}_{\sigma\{f\}, \tau\{q(f, \sigma)\}}(M\{f\}, N\{f\}) \)
Semantic type formers

1. \( \text{App}_{\sigma,\tau}(\lambda_{\sigma,\tau}(M), N) = M\{\tilde{N}\} \)
2. \( \Pi(\sigma, \tau)\{f\} = \prod(\sigma\{f\}, \tau\{q(f, \sigma)\}) \in \text{Ty}(B) \)
3. \( \lambda_{\sigma,\tau}(M)\{f\} = \lambda_{q\{f\},\tau\{q(f,\sigma)\}}(M\{q(f, \sigma)\}) \)
4. \( \text{App}_{\sigma,\tau}(M, N)\{f\} = \text{App}_{\sigma\{f\},\tau\{q(f,\sigma)\}}(M\{f\}, N\{f\}) \)
Semantic type formers

1. \[ \text{App}_{\sigma,\tau}(\lambda_{\sigma,\tau}(M), N) = M\{\bar{N}\} \]
2. \[ \text{Pi}(\sigma, \tau\{f\}) = \prod(\sigma\{f\}, \tau\{q(f, \sigma)\}) \in \text{Ty}(B) \]
3. \[ \lambda_{\sigma,\tau}(M)\{f\} = \lambda_{q\{f\},\tau\{q(f,\sigma)\}}(M\{q(f,\sigma)\}) \]
4. \[ \text{App}_{\sigma,\tau}(M, N)\{f\} = \text{App}_{\sigma\{f\},\tau\{q(f,\sigma)\}}(M\{f\}, N\{f\}) \]
<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{App}<em>{\sigma,\tau}(\lambda</em>{\sigma,\tau}(M), N) = M{\bar{N}} )</td>
</tr>
<tr>
<td>2</td>
<td>( \Pi(\sigma, \tau){f} = \Pi(\sigma{f}, \tau{q(f, \sigma)}) \in Ty(B) )</td>
</tr>
<tr>
<td>3</td>
<td>( \lambda_{\sigma,\tau}(M){f} = \lambda_{q{f},\tau{q(f, \sigma)}}(M{q(f, \sigma)}) )</td>
</tr>
<tr>
<td>4</td>
<td>( \text{App}<em>{\sigma,\tau}(M, N){f} = \text{App}</em>{\sigma{f},\tau{q(f, \sigma)}}(M{f}, N{f}) )</td>
</tr>
</tbody>
</table>
Interpretation

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

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

$$
\begin{align*}
[\Gamma; x : \sigma] &= [\Gamma].[\Gamma; \sigma] \text{ if } x \not\in \Gamma \\
[\Gamma; \Pi x : \sigma.\tau] &= \Pi((\Gamma; \sigma), [\Gamma, x : \sigma; \tau]) \\
[\Gamma; x : \sigma, \Delta, y : \tau; x] &= [\Gamma, x : \sigma, \Delta; x]\{p([\Gamma, x : \sigma, \Delta; \tau])\} \\
[\Gamma; App_{x:\sigma.\tau} (M, N)] &= App_{[\Gamma, \sigma],[\Gamma, x:\sigma.\tau]} \circ \langle[\Gamma; M], [\Gamma; N]\rangle_{[\Gamma; \Pi x:\sigma.\tau]} \\
[\Gamma; \lambda x : \sigma. M^\tau] &= \lambda_{[\Gamma; \sigma],[\Gamma, x:\sigma;\tau]}([\Gamma, x : \sigma; M])
\end{align*}
$$
Interpretation

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

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

\[
[\Gamma; x : \sigma] = [\Gamma].[\Gamma; \sigma] \text{ if } x \not\in \Gamma \\
[\Gamma; \Pi x : \sigma.\tau] = \Pi([\Gamma; \sigma], [\Gamma, x : \sigma; \tau]) \\
[\Gamma; x : \sigma, \Delta, y : \tau; x] = [\Gamma, x : \sigma, \Delta; x] \{\rho([\Gamma, x : \sigma, \Delta; \tau])\} \\
[\Gamma; App_{x:\sigma}^{\tau}(M, N)] = App_{[\Gamma, \sigma], [\Gamma, x : \sigma.\tau]}(\langle [\Gamma; M], [\Gamma; N] \rangle_{[\Gamma; \Pi x : \sigma.\tau]}) \\
[\Gamma; \lambda x : \sigma. M^\tau] = \lambda_{[\Gamma; \sigma], [\Gamma, x : \sigma; \tau]}([\Gamma, x : \sigma; M])
\]
Interpretation

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 $\text{Ty}([\Gamma])$.
3. Pairs $\Gamma; M$ to elements of $\text{Tm}(\sigma)$ for some $\sigma \in \text{Ty}([\Gamma])$

\[
\begin{align*}
[\Gamma; x : \sigma] &= [\Gamma].[\Gamma; \sigma] \text{ if } x \notin \Gamma \\
[\Gamma; \Pi x : \sigma.\tau] &= \Pi([\Gamma; \sigma], [\Gamma, x : \sigma; \tau]) \\
[\Gamma; x : \sigma, \Delta, y : \tau; x] &= [\Gamma, x : \sigma, \Delta; x]\{\rho([\Gamma, x : \sigma, \Delta; \tau])\} \\
[\Gamma; \text{App}_{\xi:\sigma}^\tau(M, N)] &= \text{App}_{[\Gamma, \sigma], [\Gamma, x: \sigma.\tau]} \circ \langle [\Gamma; M], [\Gamma; N]\rangle [\Gamma; \Pi x: \sigma.\tau] \\
[\Gamma; \lambda x : \sigma. M^\tau] &= \lambda_{[\Gamma; \sigma],[\Gamma, x: \sigma.\tau]}([\Gamma, x : \sigma; M])
\end{align*}
\]
Interpretation

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

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

\[
[\Gamma; x : \sigma] = [\Gamma].[\Gamma; \sigma] \text{ if } x \not\in \Gamma
\]
\[
[\Gamma; \prod x : \sigma.\tau] = \prod([\Gamma; \sigma], [\Gamma, x : \sigma; \tau])
\]
\[
[\Gamma; x : \sigma, \Delta, y : \tau; x] = [\Gamma, x : \sigma, \Delta; x]\{p([\Gamma, x : \sigma, \Delta; \tau])\}
\]
\[
[\Gamma; \text{App}_{x:\sigma}\tau(M, N)] = \text{App}_{[\Gamma,\sigma],[\Gamma, x:\sigma.\tau]} \circ \langle[\Gamma; M], [\Gamma; N]\rangle[\Gamma; \prod x:\sigma.\tau]
\]
\[
[\Gamma; \lambda x : \sigma. M^\tau] = \lambda_{[\Gamma;\sigma],[\Gamma, x:\sigma;\tau]}([\Gamma, x : \sigma; M])
\]