An Introduction to Type Theory

Part 2

Tallinn, September 2003

http://www.cs.nott.ac.uk/~txa/tallinn/

Thorsten Altenkirch
University of Nottingham
Plan of the course

1. Intuitionistic logic from a type theoretic perspective
2. Basic constructions of Type Theory
3. Programming with dependent types
Today

Basic constructions of Type Theory
Today

Basic constructions of Type Theory

- From Logic to Type Theory: $\Pi$, $\Sigma$.  

Today

Basic constructions of Type Theory

- From Logic to Type Theory: $\Pi, \Sigma$.
- Example: Decidability of $=$
Today

Basic constructions of Type Theory

- From Logic to Type Theory: $\Pi, \Sigma$.
- Example: Decidability of $=$
- Proof or program?
Today

Basic constructions of Type Theory

- From Logic to Type Theory: $\Pi, \Sigma$.
- Example: Decidability of $=$
- Proof or program?
- $\text{Nat}, \Sigma, +, =$
- Pattern matching and elimination
Today

Basic constructions of Type Theory

- From Logic to Type Theory: $\Pi, \Sigma$.
- Example: Decidability of $=$
- Proof or program?
- $\text{Nat}, \Sigma, +, =$
  - Pattern matching and elimination
- Uniqueness of equality proofs
Today

Basic constructions of Type Theory

- From Logic to Type Theory: $\Pi, \Sigma$.
- Example: Decidability of $=$
- Proof or program?
- $\mathsf{Nat}, \Sigma, +, =$
  Pattern matching and elimination
- Uniqueness of equality proofs
- Inductive families
Basic constructions of Type Theory

From Logic to Type Theory: $\Pi, \Sigma$.

Example: Decidability of $=$

Proof or program?

$\text{Nat}, \Sigma, +, =$

Pattern matching and elimination

Uniqueness of equality proofs

Inductive families

Loose ends
In intuitionistic logic constructing proofs is very much like writing functional programs. In Type Theory we go one step further: Proofs = Programs
Propositions = Types
In intuitionistic logic constructing proofs is very much like writing functional programs.
From Logic to Type Theory

- In intuitionistic logic constructing proofs is very much like writing functional programs.
- In Type Theory we go one step further:
  - Proofs = Programs
  - Propositions = Types
From Logic to Type Theory

We will also make the following identifications:
- Implication (→)
- Universal quantifications (∀)
- Pi-types (π)
- Function types (→)
- Conjunction (∧)
- Existential quantification (∃)
- Sigma-types (σ)
- Cartesian product (×)
- Disjunction (∨)
- Disjoint union (⊔)
From Logic to Type Theory

We will also make the following indentifications:

- Implication (\( \rightarrow \))
- Universal quantifications (\( \forall \))
- Pi-types (\( \Pi \))
- Function types (\( \rightarrow \))
- Conjunction (\( \land \))
- Existential quantification (\( \exists \))
- Sigma-types (\( \Sigma \))
- Cartesian product (\( \times \))
- Disjunction (\( \lor \))
We will also make the following identifications:

- Implication ($\rightarrow$)
- Universal quantifications ($\forall$)
- Function types ($\rightarrow$)
- Conjunction
- Existential quantification
- Sigma-types
- Cartesian product
- Disjunction
- Disjoint union
We will also make the following identifications:

<table>
<thead>
<tr>
<th>Implication ( (\rightarrow) )</th>
<th>Pi-types ( (\Pi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal quantifications ( (\forall) )</td>
<td></td>
</tr>
<tr>
<td>Function types ( (\rightarrow) )</td>
<td></td>
</tr>
</tbody>
</table>
We will also make the following identifications:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>Implication</td>
</tr>
<tr>
<td>$\forall$</td>
<td>Universal quantifications</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>Pi-types</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>Function types</td>
</tr>
<tr>
<td>$\land$</td>
<td>Conjunction</td>
</tr>
<tr>
<td>$\exists$</td>
<td>Existential quantification</td>
</tr>
<tr>
<td>$\times$</td>
<td>Cartesian product</td>
</tr>
</tbody>
</table>
We will also make the following identifications:

<table>
<thead>
<tr>
<th>Implication ($\rightarrow$)</th>
<th>Pi-types ($\Pi$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal quantifications ($\forall$)</td>
<td></td>
</tr>
<tr>
<td>Function types ($\rightarrow$)</td>
<td></td>
</tr>
<tr>
<td>Conjunction ($\land$)</td>
<td>Sigma-types ($\Sigma$)</td>
</tr>
<tr>
<td>Existential quantification ($\exists$)</td>
<td></td>
</tr>
<tr>
<td>Cartesian product ($\times$)</td>
<td></td>
</tr>
</tbody>
</table>
We will also make the following identifications:

<table>
<thead>
<tr>
<th>Implication ($\rightarrow$)</th>
<th>Pi-types ($\Pi$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal quantifications ($\forall$)</td>
<td></td>
</tr>
<tr>
<td>Function types ($\rightarrow$)</td>
<td></td>
</tr>
<tr>
<td>Conjunction ($\land$)</td>
<td></td>
</tr>
<tr>
<td>Existential quantification ($\exists$)</td>
<td>Sigma-types ($\Sigma$)</td>
</tr>
<tr>
<td>Cartesian product ($\times$)</td>
<td></td>
</tr>
<tr>
<td>Disjunction</td>
<td></td>
</tr>
<tr>
<td>Disjoint union ($\dot{+}$)</td>
<td></td>
</tr>
</tbody>
</table>
We will also make the following identifications:

<table>
<thead>
<tr>
<th>Logic Symbols</th>
<th>Type Theory Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implication</td>
<td>(→)</td>
</tr>
<tr>
<td>Universal quantifications</td>
<td>(∀)</td>
</tr>
<tr>
<td>Function types</td>
<td>(→)</td>
</tr>
<tr>
<td>Conjunction</td>
<td>(˄)</td>
</tr>
<tr>
<td>Existential quantification</td>
<td>(∃)</td>
</tr>
<tr>
<td>Cartesian product</td>
<td>(×)</td>
</tr>
<tr>
<td>Disjunction</td>
<td></td>
</tr>
<tr>
<td>Disjoint union</td>
<td>(+)</td>
</tr>
</tbody>
</table>
Elimination becomes more subtle:
From Logic to Type Theory

Elimination becomes more subtle:

Pattern matching  We have to keep track of the steps in the type.
Elimination becomes more subtle:

**Pattern matching**  We have to keep track of the steps in the type.

**Elimination constants**  Get replaced by their dependent versions.
Set theoretic encodings
Set theoretic encodings

Let $A, B$ be sets, then:

$$A \times B = \{(a, b) \mid a \in A \land b \in B\}$$

$$A \rightarrow B = \{f \subseteq A \times B \mid \forall a \in A. \exists! b \in B. (a, b) \in f\}$$
Set theoretic encodings

Let $A, B$ be sets, then:

$$A \times B = \{(a, b) \mid a \in A \land b \in B\}$$
$$A \to B = \{f \subseteq A \times B \mid \forall a \in A. \exists! b \in B. (a, b) \in f\}$$

Let $A$ be a set and for each $a \in A$ let $B(a)$ be a set, then:

$$\Sigma a \in A. B(a) = \{(a, b) \mid a \in A \land b \in B(a)\}$$
$$\Pi a \in A. B(a) =$$
$$\{f \subseteq \Sigma a \in A. B(a) \mid \forall a \in A. \exists! b \in B(a). (a, b) \in f\}$$
Example: Decidability of \( = \)
In intuitionistic arithmetic we can prove

$$\forall m, n \in \text{Nat.}(m = n) \lor (m \neq n)$$

where $m \neq n = \neg (m = n) = (m = n) \rightarrow \text{False}$
Example: Decidability of $=$

- In intuitionistic arithmetic we can prove

  $$\forall m, n \in \text{Nat}. (m = n) \lor (m \neq n)$$

  where $m \neq n = \neg (m = n) = (m = n) \rightarrow \text{False}$

- We will use this example to motivate the idea that proofs $=$ programs.
Natural Numbers
Natural Numbers

How to form?
Natural Numbers

How to form?

Nat ∈ Type
Natural Numbers

How to form?

\[ \text{Nat} \in \text{Type} \]

How to construct?
Natural Numbers

How to form?

\[ \text{Nat} \in \text{Type} \]

How to construct?

\[ \begin{align*}
0 & \in \text{Nat} \\
\text{sn} & \in \text{Nat} \\
\end{align*} \]
Equality
Equality

How to form?
Equality

How to form?

\[ A \in \text{Type} \quad a, b \in A \]

\[ a = b \in \text{Type} \]
Equality

How to form?

\[ A \in \text{Type} \quad a, b \in A \]
\[ a = b \in \text{Type} \]

How to prove?
Equality

How to form?

\[ A \in \text{Type} \quad a, b \in A \]
\[
\frac{}{a = b \in \text{Type}}
\]

How to prove?

\[ A \in \text{Type} \quad a \in A \]
\[
\frac{}{\text{refl} \ a \in a = a}
\]
Proving decidability
Proving decidability

We will present a proof

\[ \text{eqN} \in \Pi m, n \in \text{Nat.} (m = n) + (m \neq n) \]

using pattern matching.
Proving decidability

- We will present a proof

\[ \text{eqN} \in \Pi m, n \in \text{Nat}. (m=n) + (m\neq n) \]

using pattern matching.

- We will discuss the rules for pattern matching later.
Peano’s axioms
Peano’s axioms

\[ \text{consN} \in \Pi n \in \text{Nat}. (0 = s n) \rightarrow \text{False} \]
Peano’s axioms

\[ \text{consN} \in \prod n \in \text{Nat}. (0 = s n) \rightarrow \text{False} \]

where

empty pattern
Peano’s axioms

\[ \text{consN} \in \Pi n \in \text{Nat}. (0 = \text{s}\ n) \rightarrow \text{False} \]

where

*empty pattern*

\[ \text{consN}' \in \Pi n \in \text{Nat}. (\text{s}\ n = 0) \rightarrow \text{False} \]
Peano’s axioms

\[ \text{consN} \in \Pi n \in \text{Nat}. (0 = s n) \rightarrow \text{False} \]

where

\textit{empty pattern}

\[ \text{consN'} \in \Pi n \in \text{Nat}. (s n = 0) \rightarrow \text{False} \]

where

\textit{empty pattern}
Peano’s axioms
Peano’s axioms

\[ \text{resps} \in \Pi m, n \in \text{Nat}. (m=n) \rightarrow s\ m = s\ n \]
Peano’s axioms

\[ \text{resps} \in \Pi m, n \in \text{Nat.} (m = n) \rightarrow s \ m = s \ n \]

where

\[ \text{resps} \ m \ m \ (\text{refl} \ m) = \text{refl} \ (s \ m) \]
Peano’s axioms

\[ \text{resps} \in \Pi m, n \in \text{Nat}. (m = n) \rightarrow s\, m = s\, n \]

where

\[ \text{resps}\, m\, m\, (\text{refl}\, m) = \text{refl}\, (s\, m) \]

\[ \text{injs} \in \Pi m, n \in \text{Nat}. (s\, m = s\, n) \rightarrow m = n \]
Peano’s axioms

\[ \text{resps} \in \Pi m, n \in \text{Nat}. (m = n) \rightarrow s\ m = s\ n \]

where

\[ \text{resps} \ m \ m \ \text{(refl} \ m) = \text{refl} \ (s \ m) \]

\[ \text{injs} \in \Pi m, n \in \text{Nat}. (s\ m = s\ n) \rightarrow m = n \]

where

\[ \text{injs} \ m \ m \ \text{(refl} \ (s\ m)) = \text{refl} \ m \]
Proving decidability
Proving decidability

\[ \text{eqNs} \in \prod_{m, n \in \text{Nat.}} ((m=n) + (m \neq n)) \rightarrow ((s \, m = s \, n) + (s \, m \neq s \, n)) \]
Proving decidability

\[
\text{eqNs} \in \prod m, n \in \text{Nat.} ((m=n)+(m\neq n)) \rightarrow ((s\ m=s\ n)+(s\ m\neq s\ n))
\]

where

\[
\begin{align*}
\text{eqNs} \ m \ n \ (\text{inl} \ p) & = \ \text{inl} \ (\text{resps} \ m \ n \ p) \\
\text{eqNs} \ m \ n \ (\text{inr} \ f) & = \ \text{inr} \ (\lambda q. f (\text{injs} \ m \ n \ f))
\end{align*}
\]
Proving decidability
Proving decidability

\[ \text{eqN} \in \Pi m, n \in \text{Nat}. (m=n) + (m \neq n) \]
Proving decidability

\[ \text{eqN} \in \Pi m, n \in \text{Nat}. (m=n) + (m \neq n) \]

where

\[
\begin{align*}
\text{eqN } 0 \ 0 & \quad = \quad \text{inl } (\text{refl } 0) \\
\text{eqN } 0 \ (s \ n) & \quad = \quad \text{inr } (\text{consN}' \ n) \\
\text{eqN } (s \ m) \ 0 & \quad = \quad \text{inr } (\text{consN } m) \\
\text{eqN } (s \ m) \ (s \ n) & \quad = \quad \text{eqNs } m \ n \ (\text{eqN } m \ n)
\end{align*}
\]
Proof or program?

We can use `reduce` to effectively decide whether two numbers are equal. If `reduce` is its canonical form, then the numbers are equal and proves this. If `reduce` is not its canonical form, then the numbers are not equal and proves this.

`reduce` is a program whose specification is in its type. Equality proofs contain no information, hence they do not have to be calculated at run time. Hence `reduce` is not less efficient than an ordinary program to determine equality of natural numbers.
Proof or program?

We can use \( \text{eqN} \) to effectively decide whether two numbers \( m, n \in \text{Nat} \) are equal.
Proof or program?

- We can use $\text{eqN}$ to effectively decide whether two numbers $m, n \in \text{Nat}$ are equal.
- Reduce $\text{eqN} \, m \, n$ to its canonical form.
We can use $\text{eqN}$ to effectively decide whether two numbers $m, n \in \text{Nat}$ are equal.

Reduce $\text{eqN} \; m \; n$ to its canonical form.

If it is $\text{inl} \; p$ then the numbers are equal and $p \in m \equiv n$ proves this.
Proof or program?

- We can use \( \text{eqN} \) to effectively decide whether two numbers \( m, n \in \text{Nat} \) are equal.
- Reduce \( \text{eqN} \ m \ n \) to its canonical form.
- If it is \( \text{inl} \ p \) then the numbers are equal and \( p \in m = n \) proves this.
- If it is \( \text{inr} \ f \) then the numbers are not equal and \( f \in m \neq n \) proves this.
Proof or program?

- We can use \texttt{eqN} to effectively decide whether two numbers \( m, n \in \text{Nat} \) are equal.
- Reduce \texttt{eqN \ m \ n} to its canonical form.
- If it is \texttt{inl \ p} then the numbers are equal and \( p \in m = n \) proves this.
- If it is \texttt{inr \ f} then the numbers are not equal and \( f \in m \neq n \) proves this.
- \texttt{eqN} is a program whose specification is in its type.
Proof or program?

- We can use eqN to effectively decide whether two numbers $m, n \in \text{Nat}$ are equal.
- Reduce eqN $m n$ to its canonical form.
- If it is inl $p$ then the numbers are equal and $p \in m = n$ proves this.
- If it is inr $f$ then the numbers are not equal and $f \in m \neq n$ proves this.
- eqN is a program whose specification is in its type.
- Equality proofs contain no information, hence they do not have to be calculated at run time.
Proof or program?

- We can use eqN to effectively decide whether two numbers \( m, n \in \text{Nat} \) are equal.
- Reduce eqN \( m \, n \) to its canonical form.
- If it is \( \text{inl} \, p \) then the numbers are equal and \( p \in m = n \) proves this.
- If it is \( \text{inr} \, f \) then the numbers are not equal and \( f \in m \neq n \) proves this.
- eqN is a program whose specification is in its type.
- Equality proofs contain no information, hence they do not have to be calculated at run time.
- Hence eqN is not less efficient than an ordinary program to determine equality of natural numbers.
The same principle can be applied to other problems, e.g. once we have specified

\[ \text{Prime} \in \text{Nat} \to \text{Type} \]
Proof or program?

The same principle can be applied to other problems, e.g. once we have specified

\[
\text{Prime} \in \text{Nat} \rightarrow \text{Type}
\]

we can implement a primality checker as

\[
\text{isPrime} \in \prod n \in \text{Nat}. (\text{Prime} n) + \neg (\text{Prime} n)
\]
Pattern matching for \textit{Nat}
Pattern matching for \texttt{Nat}

If a pattern variable $n$ has type \texttt{Nat} we can split the pattern into two, replacing $n$ by $0$ in the first line and by $s m$ in the second, where $m \in \texttt{Nat}$ is a fresh variable.
Pattern matching for \texttt{Nat}

- If a pattern variable $n$ has type \texttt{Nat} we can split the pattern into two, replacing $n$ by 0 in the first line and by $sm$ in the second, where $m \in \texttt{Nat}$ is a fresh variable.

- Since $n$ may appear in the type we have to substitute $n$ by 0 and $sm$ respectively.
Pattern matching for \( \text{Nat} \)

- If a pattern variable \( n \) has type \( \text{Nat} \) we can split the pattern into two, replacing \( n \) by 0 in the first line and by \( sm \) in the second, where \( m \in \text{Nat} \) is a fresh variable.

- Since \( n \) may appear in the type we have to substitute \( n \) by 0 and \( sm \) respectively.

- We may use the function \( f \) we are defining recursively on a subpattern, (e.g. \( m \) above).
If a pattern variable $n$ has type $\text{Nat}$ we can split the pattern into two, replacing $n$ by $0$ in the first line and by $sm$ in the second, where $m \in \text{Nat}$ is a fresh variable.

Since $n$ may appear in the type we have to substitute $n$ by $0$ and $sm$ respectively.

We may use the function $f$ we are defining recursively on a subpattern, (e.g. $m$ above).

The precise rules governing structural recursion in the presence of other variables and mutual recursive definitions are more involved.
Pattern matching for $\Sigma$, $+$
Pattern matching for $\Sigma$, $+$

$\Sigma x \in A.B$ has the same canonical constant $(a,b)$ as $\exists$ hence the same rules for pattern matching apply.
Pattern matching for $\Sigma, +$

- $\Sigma x \in A . B$ has the same canonical constant $(a, b)$ as $\exists$ hence the same rules for pattern matching apply.

- Similarly $A + B$ has canonical constants $\text{inl}, \text{inr}$ as $\lor$ and hence the same rules for pattern matching apply.
Pattern matching for \( \Sigma, + \)

- \( \Sigma x \in A.B \) has the same canonical constant \((a,b)\) as \( \exists \) hence the same rules for pattern matching apply.

- Similarly \( A+B \) has canonical constants \( \text{inl}, \text{inr} \) as \( \lor \) and hence the same rules for pattern matching apply.

- As a consequence of \( \text{Prop} = \text{Type} \) variables ranging over \( \Sigma \) and \( + \) types may occur in the type and have to be substituted.
Elimination constants

As a special instance of the pattern matching rules we will derive elimination constants. The principle Equivalence of pattern matching and elimination still holds. That is every pattern matching proof can be replaced by one only using elimination constants.
Elimination constants

As a special instance of the pattern matching rules we will derive elimination constants.
Elimination constants

- As a special instance of the pattern matching rules we will derive elimination constants.
- The principle *Equivalence of pattern matching and elimination* still holds.
Elimination constants

- As a special instance of the pattern matching rules we will derive elimination constants.
- The principle *Equivalence of pattern matching and elimination* still holds.
- That is every pattern matching proof can be replaced by one only using elimination constants.
Elimination for $\text{Nat}$
Elimination for \( \text{Nat} \)

\[
\begin{align*}
C & \in \text{Nat} \to \text{Type} & z & \in C \ 0 & s & \in \Pi n \in \text{Nat}.C \ n \to C \ (s \ n) & m & \in \text{Nat} \\
\hline
\text{natElim} & C \ z \ s \ m & \in C \ m
\end{align*}
\]
Elimination for \( \text{Nat} \)

\[
C \in \text{Nat} \rightarrow \text{Type} \quad z \in C \, 0 \quad s \in \Pi n \in \text{Nat}.C \, n \rightarrow C \, (s \, n) \quad m \in \text{Nat} \\
\text{natElim} \, C \, z \, s \, m \in C \, m
\]

where

\[
\text{natElim} \, z \, s \, 0 = z \\
\text{natElim} \, z \, s \, (s \, n) = s \, n \, (\text{natElim} \, z \, s \, n)
\]
One stone, two birds
One stone, two birds

Note that \texttt{natElim} unifies two different principles:
One stone, two birds

Note that \texttt{natElim} unifies two different principles:

- **primitive recursion**  We obtain simply typed primitive recursion if the \textit{motive} \( C \) is constant.
One stone, two birds

Note that \texttt{natElim} unifies two different principles:

\textbf{primitive recursion} We obtain simply typed primitive recursion if the \textit{motive} \( C \) is constant.

\textbf{induction} When reading \texttt{Type} as \texttt{Prop} we obtain the principle of induction.
Elimination for +
Elimination for \(\top\)

\[
\begin{align*}
A, B &\in \text{Type} \quad C \in (A+B) \rightarrow \text{Type} \\
l &\in \Pi a \in A. C \ (\text{inl } a) \\
r &\in \Pi b \in B. C \ (\text{inr } b) \\
p &\in A+B \\
\hline
\text{plusElim} \ l \ r \ p &\in C \ p
\end{align*}
\]
Elimination for $+$

$A, B \in \text{Type} \quad C \in (A+B) \rightarrow \text{Type}$

$l \in \Pi a \in A. C \ (\text{inl } a)$

$r \in \Pi b \in B. C \ (\text{inr } b)$

$p \in A+B$

\[ \text{plusElim } l \ r \ p \in C \ p \]

where

\[ \text{plusElim } l \ r \ (\text{inl } a) = l \ a \]

\[ \text{plusElim } l \ r \ (\text{inr } b) = r \ b \]
A little quiz

What is the construct corresponding to \( x \) in programming? The type corresponding to \( x \) is called Unit, written \( \mathbf{1} \).

We didn't need an elimination constant for \( x \), do we need one for \( x \)?
A little quiz

What is the construct corresponding to \texttt{plusElim} in programming?
A little quiz

What is the construct corresponding to `plusElim` in programming?

The type corresponding to `True` is called `Unit`, written `1`. We didn’t need an elimination constant for `True`, do we need one for `1`?
Elimination for $\Sigma$
Elimination for $\Sigma$

\[
A \in \text{Type} \quad B \in A \rightarrow \text{Type} \\
C \in (\Sigma a \in A.B \ a) \rightarrow \text{Type} \\
f \in \Pi a \in A.\Pi b \in B \ a.C \ (a,b) \\
p \in \Sigma a \in A.B \ a \\
\text{sigmaElim} \ f \ p \in C \ p
\]
Elimination for $\Sigma$

\[
\begin{align*}
A & \in \text{Type} \quad B \in A \rightarrow \text{Type} \\
C & \in (\Sigma a \in A. B \ a) \rightarrow \text{Type} \\
f & \in \Pi a \in A. \Pi b \in B \ a. C \ (a,b) \\
p & \in \Sigma a \in A. B \ a \\
\hline
\text{sigmaElim} \ f \ p & \in C \ p
\end{align*}
\]

where

\[
\text{sigmaElim} \ f \ (a,b) \ = \ f \ a \ b
\]
Alternative: projections
Alternative: projections

There is an alternative form of elimination for $\Sigma$ using projections.
There is an alternative form of elimination for $\Sigma$ using projections.

$$A \in \text{Type} \quad B \in A \to \text{Type} \quad p \in \Sigma a \in A. B a$$

$$\text{fst } p \in A \quad \text{snd } p \in B (\text{fst } p)$$
Alternative: projections

There is an alternative form of elimination for $\Sigma$ using projections.

\[
A \in \text{Type} \quad B \in A \rightarrow \text{Type} \quad p \in \Sigma a \in A. B a
\]

\[
\begin{align*}
\text{fst } p & \in A \\
\text{snd } p & \in B (\text{fst } p)
\end{align*}
\]

where

\[
\begin{align*}
\text{fst } (a, b) & = a \\
\text{snd } (a, b) & = b
\end{align*}
\]
Comparing `sigmaElim` vs. `fst, snd`
Comparing \texttt{sigmaElim} vs. \texttt{fst,snd}

Which form of elimination is better?
Comparing $\textit{sigmaElim}$ vs. $\textit{fst, snd}$

- Which form of elimination is better?
- Can we use $\textit{sigmaElim}$ to implement $\textit{fst}$ and $\textit{snd}$?
The axiom of choice
The axiom of choice

We can use \texttt{fst} and \texttt{snd} to implement the \textit{axiom of choice}.
The axiom of choice

We can use \texttt{fst} and \texttt{snd} to implement the \textit{axiom of choice}.

\[
\begin{align*}
A, B \in \text{Type} & \quad C \in A \to B \to \text{Type} & \quad f \in \Pi a \in A. \Sigma b \in B. C a b \\
\text{choice} f \in \Sigma g \in A \to B. \Pi a \in A. C a (g a)
\end{align*}
\]
The axiom of choice

We can use \( \text{fst} \) and \( \text{snd} \) to implement the *axiom of choice*.

\[
A, B \in \text{Type} \quad C \in A \rightarrow B \rightarrow \text{Type} \quad f \in \Pi a \in A. \Sigma b \in B. C \ a \ b \\
\text{choice } f \in \Sigma g \in A \rightarrow B. \Pi a \in A. C \ a \ (g \ a)
\]

where

\[
\text{choice } f = (\lambda a \in A. \text{fst} \ (f \ a), \lambda a \in A. \text{snd} \ (f \ a))
\]
The axiom of choice

This shows that the axiom of choice is justified constructively. However, in the presence of the principle of excluded middle it is a sign of non-constructive reasoning.
The axiom of choice

This shows that the axiom of choice is justified constructively.
The axiom of choice

- This shows that the axiom of choice is justified constructively.

- However, in the presence of the principle of excluded middle it is a sign of non-constructive reasoning.
Pattern matching for $=$

The rules for pattern matching for equality proofs involve unification problems. Given a pattern variable, there are the following cases:

- The unification problem is unsolvable, in this case we can eliminate the pattern.
- The unification problem has a most general solution which is given by a substitution. Then can be replaced by and the substitution has to be applied to the type as well.
- The unification problem is irreducible, in this case we cannot reduce the pattern.
Pattern matching for $=$

The rules for pattern matching for equality proofs involve unification problems.
The rules for pattern matching for equality proofs involve unification problems.

Given a pattern variable $q \in a = b$, there are the following cases:
Pattern matching for $=$

- The rules for pattern matching for equality proofs involve unification problems.

- Given a pattern variable $q \in a = b$, there are the following cases:
  - The unification problem $a = b$ is unsolvable, in this case we can eliminate the pattern.
The rules for pattern matching for equality proofs involve unification problems.

Given a pattern variable $q \in a = b$, there are the following cases:

- The unification problem $a = b$ is unsolvable, in this case we can eliminate the pattern.
- The unification problem $a = b$ has a most general solution which is given by a substitution $\rho$. Then $q$ can be replaced by $\text{refl } a \rho$ and the substitution $\rho$ has to be applied to the type as well.
Pattern matching for $=$

- The rules for pattern matching for equality proofs involve unification problems.

- Given a pattern variable $q \in a = b$, there are the following cases:
  - The unification problem $a = b$ is unsolvable, in this case we can eliminate the pattern.
  - The unification problem $a = b$ has a most general solution which is given by a substitution $\rho$. Then $q$ can be replaced by $\text{refl } a \rho$ and the substitution $\rho$ has to be applied to the type as well.
  - The unification problem $a = b$ is irreducible, in this case we cannot reduce the pattern.
Reducing unification problems

We only consider the special case of terms over \( \cdot \) here.

Problems of the form \( \cdot \) can be solved trivially.

Problems of the form \( \cdot \) or \( \cdot \) are unsolvable.

Problems of the form \( \cdot \), where \( \cdot \) does not occur in \( \cdot \) can be solved and give rise to the substitution \( \cdot \).

The problem \( \cdot \) can be reduced to \( \cdot \).

All other problems are irreducible.
Reducing unification problems

We only consider the special case of terms over $\text{Nat}$ here.
Reducing unification problems

We only consider the special case of terms over \( \text{Nat} \) here.

Problems of the form \( m = m \) can be solved trivially.
Reducing unification problems

We only consider the special case of terms over $\text{Nat}$ here.

- Problems of the form $m = m$ can be solved trivially.
- Problems of the form $0 = sm$ or $n = s(s(\ldots(s\ n)\ldots))$ are unsolvable.
Reducing unification problems

We only consider the special case of terms over $\text{Nat}$ here.

- Problems of the form $m = m$ can be solved trivially.
- Problems of the form $0 = sm$ or $n = s(s(\ldots(s\,n)\ldots))$ are unsolvable.
- Problems of the form $x = m$, where $x$ does not occur in $m$ can be solved and give rise to the substitution $\rho(x) = m$. 
Reducing unification problems

We only consider the special case of terms over $\text{Nat}$ here.

- Problems of the form $m = m$ can be solved trivially.
- Problems of the form $0 = s\,m$ or $n = s(s(\ldots(s\,n)\ldots))$ are unsolvable.
- Problems of the form $x = m$, where $x$ does not occur in $m$ can be solved and give rise to the substitution $\rho(x) = m$.
- The problem $s\,m = s\,n$ can be reduced to $m = n$. 
Reducing unification problems

We only consider the special case of terms over $\text{Nat}$ here.

- Problems of the form $m = m$ can be solved trivially.
- Problems of the form $0 = sm$ or $n = s(s(\ldots(s\,n)\ldots))$ are unsolvable.
- Problems of the form $x = m$, where $x$ does not occur in $m$ can be solved and give rise to the substitution $\rho(x) = m$.
- The problem $sm = sn$ can be reduced to $m = n$.
- All other problems are irreducible.
Question
Question

Can we generalize our proof \texttt{injs} to

\[
\begin{align*}
A, B \in \text{Type} & \quad f \in A \to B \\
\text{inj} & \in \Pi a, b \in A. (f \ a = f \ b) \to a = b
\end{align*}
\]
Question

Can we generalize our proof \texttt{injs} to

\[
\frac{A, B \in \text{Type} \quad f \in A \to B}{\text{inj} \in \prod a, b \in A. (f \ a = f \ b) \to a = b}
\]

where

\[
\text{inj} \ a \ a \ (\text{refl} \ (f \ a)) = \text{refl} \ a \quad ?
\]
Elimination for =
Elimination for $\_ = \_$

$$A \in \text{Type} \quad C \in \prod a, b \in A.(a = b) \rightarrow \text{Type}$$

$$f \in \prod a \in A.C \ a \ a \ (\text{refl} \ a)$$

$$a, b \in A \quad p \in a = b$$

$$\text{eqElim} \ f \ a \ b \ p \ \in \ C \ a \ b \ p$$
Elimination for $=$

\[
\begin{align*}
A &\in \text{Type} \quad C &\in \Pi a, b \in A. (a=b) \rightarrow \text{Type} \\
f &\in \Pi a \in A. C \ a \ a \ (\text{refl} \ a) \\
a, b \in A \quad p &\in a=b \\
\hline
\text{eqElim} \ f \ a \ b \ p &\in C \ a \ b \ p
\end{align*}
\]

where

\[
\text{eqElim} \ f \ a \ a \ (\text{refl} \ a) = f \ a
\]
Pattern matching vs. elimination?

Does the *Equivalence of pattern matching and elimination* still hold?
Uniqueness of equality proofs.
Uniqueness of equality proofs.

\[ A \in \text{Type} \quad a, b \in A \quad p, q \in a=b \]
\[ \text{uneq} \ a \ b \ p \ q \in p=q \]
Uniqueness of equality proofs.

\[
A \in \text{Type} \quad a, b \in A \quad p, q \in a = b \\
\text{uneq } a \ b \ p \ q \in p = q
\]

where

\[\text{uneq } a \ a \ (\text{refl } a) \ (\text{refl } a) = \text{refl } (\text{refl } a)\]
In the early 90ies it was an open problem whether \texttt{uneq} could be derived from \texttt{eqElim}. 
Uniqueness of equality proofs.

- In the early 90ies it was an open problem whether `uneq` could be derived from `eqElim`.
- In 1993 Hofmann and Streicher showed that `uneq` does not hold in the *groupoid model* of Type Theory, although `eqElim` can be interpreted.
Uniqueness of equality proofs.

In the early 90ies it was an open problem whether \texttt{uneq} could be derived from \texttt{eqElim}.

In 1993 Hofmann and Streicher showed that \texttt{uneq} does not hold in the \textit{groupoid model} of Type Theory, although \texttt{eqElim} can be interpreted.

However, this can be fixed by introducing another elimination constant.
Another elimination for \( = \)
Another elimination for $=$

$$
A \in \text{Type} \quad C \in \prod a \in A. (a = a) \to \text{Type} \\
f \in \prod a \in A. C \; (\text{refl} \; a) \\
a \in A \quad p \in a = a \\
\overline{\text{eqElim}' \; f \; a \; p \in C \; a \; p}
$$
Another elimination for $=$

\[
A \in \text{Type} \quad C \in \prod a \in A. (a = a) \rightarrow \text{Type} \\
f \in \prod a \in A. C (\text{refl } a) \\
a \in A \quad p \in a = a \\
\hline \\
eq \text{Elim'} f a p \in C a p \\
\]

where

\[
eq \text{Elim'} f a (\text{refl } a) = f a
\]
Uniqueness of equality proofs.
Uniqueness of equality proofs.

\[ A \in \text{Type} \quad a, b \in A \quad p, q \in a = b \]

\[ \text{uneq} \ a \ b \ p \ q \in p = q \]
Uniqueness of equality proofs.

\[ A \in \text{Type} \quad a, b \in A \quad p, q \in a = b \]

\[ \text{uneq } a \ b \ p \ q \ \in \ p = q \]

where

\[ \text{uneq } a \ b \ p \ q \ = \ \text{eqElim } a \ b \ (\lambda q.\text{eqElim}' a (\lambda a.\text{refl (refl } a)) q) \ p \]
Conor’s result

In 1999 Conor McBride showed as part of his PhD that equivalence of pattern matching and elimination holds, when using inductive families, of which is a special case.
Conor’s result

In 1999 Conor McBride showed as part of his PhD that *Equivalence of pattern matching and elimination* holds, when using `eqElim'`. 
Conor’s result

In 1999 Conor McBride showed as part of his PhD that *Equivalence of pattern matching and elimination* holds, when using `eqElim'`.

In fact he showed this in the presence of *inductive families*, of which `=` is a special case.
\leq \text{ in logic}
How to define $\leq\in \text{Nat}\rightarrow\text{Nat}\rightarrow\text{Prop}$?
\( \leq \) in logic

- How to define \( \leq \in \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Prop} \)?
- \( m \leq n = \exists i \in \text{Nat}. m + i = n \)
How to define $\leq \in \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Prop}$?

$m \leq n = \exists i \in \mathbf{Nat}. m + i = n$

There is an alternative inductive definition.

$$\frac{n \in \mathbf{Nat}}{0 \leq n} \quad \frac{m \leq n}{s \, m \leq s \, n}$$
≤ in Type Theory
≤ in Type Theory

How to form?
\begin{align*}
& \text{\leq in Type Theory} \\
& \text{How to form?} \\
& \frac{m, n \in \text{Nat}}{m \leq n \in \text{Type}}
\end{align*}
≤ in Type Theory

How to form?

\[
m, n \in \text{Nat} \\
\frac{}{m \leq n \in \text{Type}}
\]

How to construct?
≤ in Type Theory

How to form?

\[ m, n \in \text{Nat} \]
\[ m \leq n \in \text{Type} \]

How to construct?

\[ n \in \text{Nat} \]
\[ \text{le0} \ n \in 0 \leq n \]

\[ p \in m \leq n \]
\[ \text{leS} \ p \in (s \ m) \leq (s \ n) \]
Pattern matching for \( \leq \)
Pattern matching for $\leq$

\[ \text{transLe} \in \Pi i, j, k \in \text{Nat} . (i \leq j) \rightarrow (j \leq k) \rightarrow i \leq k \]
Pattern matching for $\leq$

\[
\text{transLe} \in \Pi i, j, k \in \text{Nat.}(i \leq j) \rightarrow (j \leq k) \rightarrow i \leq k
\]

where

\[
\begin{align*}
\text{transLe} 0 j k (\text{le}0 \ j) \ q & = \ \text{le}0 \ k \\
\text{transLe} (s \ i) (s \ j) (s \ k) (\text{leS} \ p) (\text{leS} \ q) & = \ \text{leS} \ (\text{transLe} \ i \ j \ k \ p \ q)
\end{align*}
\]
\textbf{Leq in LEGO}

Inductive \[\text{Leq} \ : \ \text{Nat} \to \text{Nat} \to \text{Set}\]
Constructors

\[\text{le0} \ : \ \{n: \text{Nat}\} \text{Leq} \ n \ n\]
\[\text{leS} \ : \ \{m,n|\text{Nat}\} (\text{Leq} \ m \ n)
\rightarrow (\text{Leq} \ (\text{su} \ m) \ (\text{su} \ n))\};\]
Elimination for $\text{Leq}$

decl Leq_elim :
\[
\{C_{\text{Leq}}:\{x_1,x_2|\text{Nat}\}(\text{Leq } x_1 x_2) \to \text{TYPE}\}
\]
\[
\{n:\text{Nat}\}C_{\text{Leq}} (\text{le0 } n)\to
\]
\[
\{m,n|\text{Nat}\}\{x_1:\text{Leq } m n\}(C_{\text{Leq}} x_1) \to C_{\text{Leq}} (\text{leS } x_1))\to
\]
\[
\{x_1,x_2|\text{Nat}\}\{z:\text{Leq } x_1 x_2\}C_{\text{Leq}} z
\]

[[C_{\text{Leq}}:\{x_1,x_2|\text{Nat}\}(\text{Leq } x_1 x_2) \to \text{TYPE}] [f_\text{le0}:{n_1:\text{Nat}}C_{\text{Leq}} (\text{le0 } n_1)]
[f_\text{leS}:{m,n|\text{Nat}}\{x_1:\text{Leq } m n\}(C_{\text{Leq}} x_1) \to C_{\text{Leq}} (\text{leS } x_1)] [n_1:\text{Nat}] [m,n|\text{Nat}][x_1:\text{Leq } m n]
\]

\[
\text{Leq}_\text{elim} \ C_{\text{Leq}} \ f_\text{le0} \ f_\text{leS} (\text{le0 } n_1) \Rightarrow f_\text{le0} \ n_1
\]
\[
\| \text{Leq}_\text{elim} \ C_{\text{Leq}} \ f_\text{le0} \ f_\text{leS} (\text{leS } x_1) \Rightarrow
\]
\[
f_\text{leS} \ x_1 (\text{Leq}_\text{elim} \ C_{\text{Leq}} \ f_\text{le0} \ f_\text{leS} \ x_1)]
\]
Inductive definitions
Inductive definitions

Inductive definitions are a basic concept of Type Theory.
Inductive definitions

- Inductive definitions are a basic concept of Type Theory
- Inductive types can be imagined as defining a collection of trees.
Inductive definitions are a basic concept of Type Theory.

Inductive types can be imagined as defining a collection of trees.

We can have infinitary constructors, but they have to be strictly positive.
Inductive definitions

- Inductive definitions are a basic concept of Type Theory.
- Inductive types can be imagined as defining a collection of trees.
- We can have infinitary constructors, but they have to be strictly positive.
- There are also size restrictions: there is no type of all types.
Loose ends
Loose ends

The role of equality in Type Theory
extensional vs intensional
Loose ends

- The role of equality in Type Theory
  - extensional vs intensional

- Universes and reflection
  - predicative
  - impredicative
  - inconsistent