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: Π, Σ.

Today

Basic constructions of Type Theory

- From Logic to Type Theory: Π, Σ.
- Example: Decidability of $=$

Today

Basic constructions of Type Theory

- From Logic to Type Theory: Π, Σ.
- Example: Decidability of $=$
- Proof or program?

Today

Basic constructions of Type Theory

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

Today

Basic constructions of Type Theory

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

Today

Basic constructions of Type Theory

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

Today

Basic constructions of Type Theory

- From Logic to Type Theory: Π, Σ.
- Example: Decidability of $=$
- Proof or program?
- $\mathrm{Nat}, \Sigma, +, =$
  Pattern matching and elimination
- Uniqueness of equality proofs
- Inductive families
- Loose ends

From Logic to Type Theory

From Logic to Type Theory

- 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

From Logic to Type Theory

- We will also make the following indentifications:

From Logic to Type Theory

- We will also make the following indentifications:

Implication ($\rightarrow$) Universal quantifications ($\forall$) Function types ($\rightarrow$)	

From Logic to Type Theory

- We will also make the following indentifications:

Implication ($\rightarrow$) Universal quantifications ($\forall$) Function types ($\rightarrow$)	Pi-types (Π)

From Logic to Type Theory

- We will also make the following indentifications:

Implication ($\to$) Universal quantifications ($\forall$) Function types ($\to$)	Pi-types (Π)
Conjunction ($\wedge$) Existential quantification ($\exists$) Cartesian product ($\times$)	

From Logic to Type Theory

- We will also make the following indentifications:

Implication ($\rightarrow$) Universal quantifications ($\forall$) Function types ($\rightarrow$)	Pi-types (Π)
Conjunction ($\wedge$) Existential quantification ($\exists$) Cartesian product ($\times$)	Sigma-types (Σ)

From Logic to Type Theory

- We will also make the following indentifications:

Implication ($\rightarrow$) Universal quantifications ($\forall$) Function types ($\rightarrow$)	Pi-types (Π)
Conjunction ($\wedge$) Existential quantification ($\exists$) Cartesian product ($\times$)	Sigma-types (Σ)
Disjunction Disjoint union ($+$)	

From Logic to Type Theory

- We will also make the following indentifications:

Implication ($\rightarrow$) Universal quantifications ($\forall$) Function types ($\rightarrow$)	Pi-types (Π)
Conjunction ($\wedge$) Existential quantification ($\exists$) Cartesian product ($\times$)	Sigma-types (Σ)
Disjunction Disjoint union ($+$)	Disjoint union ($+$)

From Logic to Type Theory

- 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.

From Logic to Type Theory

- 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:

$$\begin{array}{rcl} A \times B & = & \{(a, b) \mid a \in A \wedge b \in B\} \\ A \to B & = & \{f \subseteq A \times B \mid \forall a \in A. \exists! b \in B. (a, b) \in f\} \end{array}$$

Set theoretic encodings

Let A, B be sets, then:

$$\begin{aligned} A \times B &= \{(a,b) \mid a \in A \wedge b \in B\} \\ A \to B &= \{f \subseteq A \times B \mid \forall a \in A. \exists! b \in B. (a,b) \in f\} \end{aligned}$$

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

$$\begin{aligned} \Sigma a \in A.B(a) &= \{(a,b) \mid a \in A \wedge b \in B(a)\} \\ \Pi a \in A.B(a) &= \\ &\quad \{f \subseteq \Sigma a \in A.B(a) \mid \forall a \in A. \exists! b \in B(a). (a,b) \in f\} \end{aligned}$$

Example: Decidability of $=$

Example: Decidability of $=$

- In intuitionistic arithmetic we can prove

$$\forall m, n \in \mathrm{Nat}.(m{=}n) \vee (m{\neq}n)$$

where $m{\neq}n = \neg(m{=}n) = (m{=}n) \to \mathrm{False}$

Example: Decidability of $=$

- In intuitionistic arithmetic we can prove

$$\forall m, n \in \mathrm{Nat}.(m{=}n) \vee (m{\neq}n)$$

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

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

Natural Numbers

Natural Numbers

How to form ?

Natural Numbers

How to form ?

$$\frac{}{\mathrm{Nat} \in \mathrm{Type}}$$

Natural Numbers

How to form ?

$$\frac{}{\mathrm{Nat} \in \mathrm{Type}}$$

How to construct ?

Natural Numbers

How to form ?

$$\frac{}{\mathrm{Nat} \in \mathrm{Type}}$$

How to construct ?

$$\frac{}{0 \in \mathrm{Nat}} \qquad \frac{n \in \mathrm{Nat}}{\mathrm{s}\, n \in \mathrm{Nat}}$$

Equality

Equality

How to form ?

Equality

How to form ?

$$\frac{A \in \mathrm{Type} \qquad a, b \in A}{a{=}b \in \mathrm{Type}}$$

Equality

How to form ?

$$\frac{A \in \mathrm{Type} \qquad a, b \in A}{a{=}b \in \mathrm{Type}}$$

How to prove ?

Equality

How to form ?

$$\frac{A \in \mathrm{Type} \qquad a, b \in A}{a{=}b \in \mathrm{Type}}$$

How to prove ?

$$\frac{A \in \mathrm{Type} \qquad a \in A}{\mathrm{refl}\ a \in a{=}a}$$

Proving decidablity

Proving decidablity

- We will present a proof

$$\mathrm{eqN} \in \Pi m, n \in \mathrm{Nat}.(m{=}n){+}(m{\neq}n)$$

using pattern matching.

Proving decidablity

- We will present a proof

$$\mathrm{eqN} \in \Pi m, n \in \mathrm{Nat}.(m{=}n){+}(m{\neq}n)$$

  using pattern matching.
- We will discuss the rules for pattern matching later.

Peano's axioms

Peano's axioms

$$\mathrm{consN} \in \Pi n \in \mathrm{Nat}.(0 = \mathrm{s}\, n) \rightarrow \mathrm{False}$$

Peano's axioms

$$\mathrm{consN} \in \Pi n \in \mathrm{Nat}.(0{=}\mathrm{s}\ n) \to \mathrm{False}$$

where

empty pattern

Peano's axioms

$$\mathrm{consN} \in \Pi n \in \mathrm{Nat}.(0{=}\mathrm{s}\,n){\to}\mathrm{False}$$

where

empty pattern

$$\mathrm{consN}' \in \Pi n \in \mathrm{Nat}.(\mathrm{s}\,n{=}0){\to}\mathrm{False}$$

Peano's axioms

$$\mathrm{consN} \in \Pi n \in \mathrm{Nat}.(0 = \mathrm{s}\, n) \to \mathrm{False}$$

where

empty pattern

$$\mathrm{consN}' \in \Pi n \in \mathrm{Nat}.(\mathrm{s}\, n = 0) \to \mathrm{False}$$

where

empty pattern

Peano's axioms

Peano's axioms

$$\mathrm{resps} \in \Pi m, n \in \mathrm{Nat}.(m{=}n){\to}\mathrm{s}\, m = \mathrm{s}\, n$$

Peano's axioms

$$\mathrm{resps} \in \Pi m, n \in \mathrm{Nat}.(m{=}n) \to \mathrm{s}\, m = \mathrm{s}\, n$$

where

$$\mathrm{resps}\, m\, m\, (\mathrm{refl}\, m) \quad = \quad \mathrm{refl}\, (\mathrm{s}\, m)$$

Peano's axioms

$$\text{resps} \in \Pi m, n \in \text{Nat}.(m{=}n){\to}\text{s}\, m = \text{s}\, n$$

where

$$\text{resps}\, m\, m\, (\text{refl}\, m) \quad = \quad \text{refl}\, (\text{s}\, m)$$

$$\text{injs} \in \Pi m, n \in \text{Nat}.(\text{s}\, m{=}\text{s}\, n){\to}m{=}n$$

Peano's axioms

$$\mathrm{resps} \in \Pi m, n \in \mathrm{Nat}.(m{=}n) \to \mathrm{s}\, m = \mathrm{s}\, n$$

where

$$\mathrm{resps}\, m\, m\, (\mathrm{refl}\, m) \quad = \quad \mathrm{refl}\, (\mathrm{s}\, m)$$

$$\mathrm{injs} \in \Pi m, n \in \mathrm{Nat}.(\mathrm{s}\, m{=}\mathrm{s}\, n) \to m{=}n$$

where

$$\mathrm{injs}\, m\, m\, (\mathrm{refl}\, (\mathrm{s}\, m)) \quad = \quad \mathrm{refl}\, m$$

Proving decidability

Proving decidability

$$\mathrm{eqNs} \in \Pi m, n \in \mathrm{Nat}.((m{=}n){+}(m{\neq}n)){\to}((\mathrm{s}\,m{=}\mathrm{s}\,n){+}(\mathrm{s}\,m{\neq}\mathrm{s}\,n))$$

Proving decidability

$$\mathrm{eqNs} \in \Pi m, n \in \mathrm{Nat}.((m{=}n)+(m{\neq}n))\to((\mathrm{s}\,m{=}\mathrm{s}\,n)+(\mathrm{s}\,m{\neq}\mathrm{s}\,n))$$

where

$$\begin{array}{rcl} \mathrm{eqNs}\,m\,n\,(\mathrm{inl}\,p) & = & \mathrm{inl}\,(\mathrm{resps}\,m\,n\,p) \\ \mathrm{eqNs}\,m\,n\,(\mathrm{inr}\,f) & = & \mathrm{inr}\,(\lambda q.f\,(\mathrm{injs}\,m\,n\,f)) \end{array}$$

Proving decidability

Proving decidability

$$\mathrm{eqN} \in \Pi m, n \in \mathrm{Nat}.(m{=}n){+}(m{\neq}n)$$

Proving decidability

$$\mathrm{eqN} \in \Pi m, n \in \mathrm{Nat}.(m{=}n){+}(m{\neq}n)$$

where

$$\begin{array}{lcl}
\mathrm{eqN}\,0\,0 & = & \mathrm{inl}\,(\mathrm{refl}\,0) \\
\mathrm{eqN}\,0\,(\mathrm{s}\,n) & = & \mathrm{inr}\,(\mathrm{consN}'\,n) \\
\mathrm{eqN}\,(\mathrm{s}\,m)\,0 & = & \mathrm{inr}\,(\mathrm{consN}\,m) \\
\mathrm{eqN}\,(\mathrm{s}\,m)\,(\mathrm{s}\,n) & = & \mathrm{eqNs}\,m\,n\,(\mathrm{eqN}\,m\,n)
\end{array}$$

Proof or program ?

Proof or program ?

- We can use eqN to effectively decide whether two numbers $m, n \in \mathrm{Nat}$ are equal.

Proof or program ?

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

Proof or program ?

- We can use eqN to effectively decide whether two numbers $m, n \in \mathrm{Nat}$ are equal.
- Reduce $\mathrm{eqN}\, m\, n$ to its canonical form.
- If it is $\mathrm{inl}\, p$ then the numbers are equal and $p \in m{=}n$ proves this.

Proof or program ?

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

Proof or program ?

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

Proof or program ?

- We can use eqN to effectively decide whether two numbers $m, n \in \mathrm{Nat}$ are equal.
- Reduce $\mathrm{eqN}\, m\, n$ to its canonical form.
- If it is $\mathrm{inl}\, p$ then the numbers are equal and $p \in m{=}n$ proves this.
- If it is $\mathrm{inr}\, f$ 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 $m, n \in \mathrm{Nat}$ are equal.
- Reduce $\mathrm{eqN}\, m\, n$ to its canonical form.
- If it is $\mathrm{inl}\, p$ then the numbers are equal and $p \in m{=}n$ proves this.
- If it is $\mathrm{inr}\, f$ 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.

Proof or program ?

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

$$\text{Prime} \in \text{Nat} \rightarrow \text{Type}$$

Proof or program ?

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

$$\mathrm{Prime} \in \mathrm{Nat} \to \mathrm{Type}$$

we can implement a primality checker as

$$\mathrm{isPrime} \in \Pi n \in \mathrm{Nat}.(\mathrm{Prime}\, n) + \neg(\mathrm{Prime}\, n)$$

Pattern matching for Nat

Pattern matching for Nat

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

Pattern matching for Nat

- If a pattern variable n has type Nat we can split the pattern into two, replacing n by 0 in the first line and by $\mathrm{s}\,m$ in the second, where $m \in \mathrm{Nat}$ is a fresh variable.
- Since n may appear in the type we have to substitute n by 0 and $\mathrm{s}\,m$ respectively.

Pattern matching for Nat

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

Pattern matching for Nat

- If a pattern variable n has type Nat we can split the pattern into two, replacing n by 0 in the first line and by $\mathrm{s}\, m$ in the second, where $m \in \mathrm{Nat}$ is a fresh variable.
- Since n may appear in the type we have to substitute n by 0 and $\mathrm{s}\, m$ 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.
- Similarily $A+B$ has canonical constants $\mathrm{inl}, \mathrm{inr}$ as $\vee$ 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.
- Similarily $A+B$ has canonical constants $\mathrm{inl}, \mathrm{inr}$ as $\vee$ and hence the same rules for pattern matching apply.
- As a consequence of $\mathrm{Prop} = \mathrm{Type}$ variables ranging over Σ and $+$ types may occur in the type and have to be substituted.

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 Nat

Elimination for Nat

$$\frac{C \in \mathrm{Nat} \to \mathrm{Type} \quad z \in C\, 0 \quad s \in \Pi n \in \mathrm{Nat}.C\, n \to C\, (\mathrm{s}\, n) \quad m \in \mathrm{Nat}}{\mathrm{natElim}\, C\, z\, s\, m \in C\, m}$$

Elimination for Nat

$$\frac{C \in \mathrm{Nat} \to \mathrm{Type} \quad z \in C\,0 \quad s \in \Pi n \in \mathrm{Nat}. C\,n \to C\,(\mathrm{s}\,n) \quad m \in \mathrm{Nat}}{\mathrm{natElim}\,C\,z\,s\,m \in C\,m}$$

where

$$\begin{array}{lcl} \mathrm{natElim}\,z\,s\,0 & = & z \\ \mathrm{natElim}\,z\,s\,(\mathrm{s}\,n) & = & s\,n\,(\mathrm{natElim}\,z\,s\,n) \end{array}$$

One stone, two birds

One stone, two birds

Note that $\mathrm{natElim}$ unifies two different principles:

One stone, two birds

Note that $\mathrm{natElim}$ unifies two different principles:

primitive recursion We obtain simply typed primitive recursion if the *motive* C is constant.

One stone, two birds

Note that $\mathrm{natElim}$ unifies two different principles:

primitive recursion We obtain simply typed primitive recursion if the *motive* C is constant.

induction When reading Type as Prop we obtain the principle of induction.

Elimination for $+$

Elimination for $+$

$$\frac{\begin{array}{l} A, B \in \mathrm{Type} \quad C \in (A+B)\to\mathrm{Type} \\ l \in \Pi a \in A.C\,(\mathrm{inl}\,a) \\ r \in \Pi b \in B.C\,(\mathrm{inr}\,b) \\ p \in A+B \end{array}}{\mathrm{plusElim}\,l\,r\,p \in C\,p}$$

Elimination for $+$

$$\frac{\begin{array}{l} A, B \in \mathrm{Type} \quad C \in (A+B)\to\mathrm{Type} \\ l \in \Pi a \in A.C\,(\mathrm{inl}\,a) \\ r \in \Pi b \in B.C\,(\mathrm{inr}\,b) \\ p \in A+B \end{array}}{\mathrm{plusElim}\,l\,r\,p \in C\,p}$$

where

$$\begin{array}{rcl} \mathrm{plusElim}\,l\,r\,(\mathrm{inl}\,a) & = & l\,a \\ \mathrm{plusElim}\,l\,r\,(\mathrm{inr}\,b) & = & r\,b \end{array}$$

A little quiz

A little quiz

- What is the construct corresponding to $\mathrm{plusElim}$ in programming?

A little quiz

- What is the construct corresponding to $\mathrm{plusElim}$ in programming?
- The type corresponding to True is called Unit, written $\mathbf{1}$. We didn't need an elimination constant for True, do we need one for $\mathbf{1}$?

Elimination for Σ

Elimination for Σ

$$\frac{\begin{array}{l} A \in \mathrm{Type} \quad B \in A{\to}\mathrm{Type} \\ C \in (\Sigma a \in A.B\ a){\to}\mathrm{Type} \\ f \in \Pi a \in A.\Pi b \in B\ a.C\ (a,b) \\ p \in \Sigma a \in A.B\ a \end{array}}{\mathrm{sigmaElim}\ f\ p \in C\ p}$$

Elimination for Σ

$$\frac{\begin{array}{l} A \in \mathrm{Type} \quad B \in A \to \mathrm{Type} \\ C \in (\Sigma a \in A.B\ a) \to \mathrm{Type} \\ f \in \Pi a \in A.\Pi b \in B\ a.C\ (a,b) \\ p \in \Sigma a \in A.B\ a \end{array}}{\mathrm{sigmaElim}\ f\ p \in C\ p}$$

where

$$\mathrm{sigmaElim}\ f\ (a,b) \quad = \quad f\ a\ b$$

Alternative: projections

Alternative: projections

There is an alternative form of elimination for Σ using projections.

Alternative: projections

There is an alternative form of elimination for Σ using projections.

$$\frac{A \in \mathrm{Type} \quad B \in A \to \mathrm{Type} \quad p \in \Sigma a \in A.B\,a}{\mathrm{fst}\,p \in A \qquad \mathrm{snd}\,p \in B\,(\mathrm{fst}\,p)}$$

Alternative: projections

There is an alternative form of elimination for Σ using projections.

$$\frac{A \in \mathrm{Type} \quad B \in A \rightarrow \mathrm{Type} \quad p \in \Sigma a \in A.B\,a}{\mathrm{fst}\,p \in A \qquad \mathrm{snd}\,p \in B\,(\mathrm{fst}\,p)}$$

where

$$\begin{aligned} \mathrm{fst}\,(a,b) &= a \\ \mathrm{snd}\,(a,b) &= b \end{aligned}$$

Comparing sigmaElim vs. fst,snd

Comparing sigmaElim **vs.** fst,snd

- Which form of elimination is better?

Comparing sigmaElim **vs.** fst,snd

- Which form of elimination is better?
- Can we use sigmaElim to implement fst and snd?

The axiom of choice

The axiom of choice

We can use fst and snd to implement the *axiom of choice*.

The axiom of choice

We can use fst and snd to implement the *axiom of choice*.

$$\frac{A, B \in \mathrm{Type} \quad C \in A \to B \to \mathrm{Type} \quad f \in \Pi a \in A.\Sigma b \in B.C\,a\,b}{\mathrm{choice}\,f \in \Sigma g \in A \to B.\Pi a \in A.C\,a\,(g\,a)}$$

The axiom of choice

We can use fst and snd to implement the *axiom of choice*.

$$\frac{A, B \in \mathrm{Type} \quad C \in A \to B \to \mathrm{Type} \quad f \in \Pi a \in A.\Sigma b \in B.C\,a\,b}{\mathrm{choice}\,f \in \Sigma g \in A \to B.\Pi a \in A.C\,a\,(g\,a)}$$

where

$$\mathrm{choice}\,f \quad = \quad (\lambda a \in A.\mathrm{fst}\,(f\,a), \lambda a \in A.\mathrm{snd}\,(f\,a))$$

The axiom of choice

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 $=$

Pattern matching for $=$

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

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:

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 cas we can eliminate the pattern.

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 cas we can eliminate the pattern.
  - The unification problem $a = b$ has a most general solution which is given by a substitution ρ. Then q can be replaced by $\mathrm{refl}\, a\rho$ and the substitution ρ 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 cas we can eliminate the pattern.
  - The unification problem $a = b$ has a most general solution which is given by a substitution ρ. Then q can be replaced by $\mathrm{refl}\, a\rho$ and the substitution ρ 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

Reducing unification problems

We only consider the special case of terms over Nat here.

Reducing unification problems

We only consider the special case of terms over Nat here.

- Problems of the form $m = m$ can be solved trivially.

Reducing unification problems

We only consider the special case of terms over Nat here.

- Problems of the form $m = m$ can be solved trivially.
- Problems of the $0 = \mathrm{s}\, m$ or $n = \mathrm{s}(\mathrm{s}(\ldots(\mathrm{s}\, n)\ldots))$ are unsolvable.

Reducing unification problems

We only consider the special case of terms over Nat here.

- Problems of the form $m = m$ can be solved trivially.
- Problems of the $0 = \mathrm{s}\, m$ or $n = \mathrm{s}(\mathrm{s}(\ldots(\mathrm{s}\, n)\ldots))$ 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 Nat here.

- Problems of the form $m = m$ can be solved trivially.
- Problems of the $0 = \mathrm{s}\, m$ or $n = \mathrm{s}(\mathrm{s}(\ldots(\mathrm{s}\, n)\ldots))$ 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 $\mathrm{s}\, m = \mathrm{s}\, n$ can be reduced to $m = n$.

Reducing unification problems

We only consider the special case of terms over Nat here.

- Problems of the form $m = m$ can be solved trivially.
- Problems of the $0 = \mathrm{s}\, m$ or $n = \mathrm{s}(\mathrm{s}(\ldots(\mathrm{s}\, n)\ldots))$ 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 $\mathrm{s}\, m = \mathrm{s}\, n$ can be reduced to $m = n$.
- All other problems are irreducible.

Question

Question

Can we *generalize* our proof injs to

$$\frac{A, B \in \mathrm{Type} \qquad f \in A \to B}{\mathrm{inj} \in \Pi a, b \in A.(f\ a = f\ b) \to a = b}$$

Question

Can we *generalize* our proof injs to

$$\frac{A, B \in \mathrm{Type} \qquad f \in A \to B}{\mathrm{inj} \in \Pi a, b \in A.(f\ a{=}f\ b){\to}a{=}b}$$

where

$$\mathrm{inj}\ a\ a\ (\mathrm{refl}\ (f\ a)) \quad = \quad \mathrm{refl}\ a \quad ?$$

Elimination for $=$

Elimination for $=$

$$\frac{\begin{array}{l} A \in \mathrm{Type} \quad C \in \Pi a, b \in A.(a{=}b)\to\mathrm{Type} \\ f \in \Pi a \in A.C\; a\; a\; (\mathrm{refl}\; a) \\ a, b \in A \quad p \in a{=}b \end{array}}{\mathrm{eqElim}\; f\; a\; b\; p \in C\; a\; b\; p}$$

Elimination for $=$

$$\frac{A \in \mathrm{Type} \quad C \in \Pi a, b \in A.(a{=}b) \to \mathrm{Type} \\ f \in \Pi a \in A.C\, a\, a\, (\mathrm{refl}\; a) \\ a, b \in A \quad p \in a{=}b}{\mathrm{eqElim}\; f\; a\; b\; p \in C\; a\; b\; p}$$

where

$$\mathrm{eqElim}\; f\; a\; a\; (\mathrm{refl}\; a) \quad = \quad f\; a$$

Pattern matching vs. elimination ?

Does the *Equivalence of pattern matching and elimination* still hold?

Uniqueness of equality proofs.

Uniqueness of equality proofs.

$$\frac{A \in \mathrm{Type} \quad a, b \in A \quad p, q \in a = b}{\mathrm{uneq}\, a\, b\, p\, q \in p = q}$$

Uniqueness of equality proofs.

$$\frac{A \in \mathrm{Type} \quad a, b \in A \quad p, q \in a{=}b}{\mathrm{uneq}\, a\, b\, p\, q \in p{=}q}$$

where

$$\mathrm{uneq}\, a\, a\, (\mathrm{refl}\, a)\, (\mathrm{refl}\, a) \quad = \quad \mathrm{refl}(\mathrm{refl}\, a)$$

Uniqueness of equality proofs.

- In the early 90ies it was an open problem wether uneq could be derived from eqElim.

Uniqueness of equality proofs.

- In the early 90ies it was an open problem wether 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 wether 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.
- However, this can be fixed by introducing another elimination constant.

Another elimination for $=$

Another elimination for $=$

$$\frac{\begin{array}{l} A \in \mathrm{Type} \quad C \in \Pi a \in A.(a{=}a) \to \mathrm{Type} \\ f \in \Pi a \in A.C\,(\mathrm{refl}\ a) \\ a \in A \quad p \in a{=}a \end{array}}{\mathrm{eqElim}'\ f\ a\ p \in C\ a\ p}$$

Another elimination for $=$

$$\frac{\begin{array}{l} A \in \mathrm{Type} \quad C \in \Pi a \in A.(a{=}a)\to\mathrm{Type} \\ f \in \Pi a \in A.C\,(\mathrm{refl}\;a) \\ a \in A \quad p \in a{=}a \end{array}}{\mathrm{eqElim}'\,f\;a\;p \in C\;a\;p}$$

where

$$\mathrm{eqElim}'\,f\;a\,(\mathrm{refl}\;a) \quad = \quad f\;a$$

Uniqueness of equality proofs.

Uniqueness of equality proofs.

$$\frac{A \in \mathrm{Type} \quad a, b \in A \quad p, q \in a = b}{\mathrm{uneq}\, a\, b\, p\, q \in p = q}$$

Uniqueness of equality proofs.

$$\frac{A \in \mathrm{Type} \quad a, b \in A \quad p, q \in a = b}{\mathrm{uneq}\, a\, b\, p\, q \in p = q}$$

where

$$\mathrm{uneq}\, a\, b\, p\, q \quad = \quad \mathrm{eqElim}\, a\, b\, (\lambda q.\mathrm{eqElim}' a(\lambda a.\mathrm{refl}(\mathrm{refl}\, a))q)\, p$$

Conor's result

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.

$\leq$ in logic

$\leq$ in logic

- How to define $\leq \in \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Prop}$?

$\leq$ in logic

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

$\leq$ in logic

- How to define $\leq \in \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Prop}$?
- $m \leq n = \exists i \in \mathrm{Nat}. m + i = n$
- There is an alternative inductive definition.

$$\frac{n \in \mathrm{Nat}}{0 \leq n} \qquad \frac{m \leq n}{\mathrm{s}\, m \leq \mathrm{s}\, n}$$

$\leq$ in Type Theory

$\leq$ in Type Theory

How to form ?

$\leq$ in Type Theory

How to form ?

$$\frac{m, n \in \mathrm{Nat}}{m \leq n \in \mathrm{Type}}$$

$\leq$ in Type Theory

How to form ?

$$\frac{m, n \in \mathrm{Nat}}{m \leq n \in \mathrm{Type}}$$

How to construct?

$\leq$ in Type Theory

How to form ?

$$\frac{m, n \in \mathrm{Nat}}{m{\leq}n \in \mathrm{Type}}$$

How to construct?

$$\frac{n \in \mathrm{Nat}}{\mathrm{le0}\, n \in 0{\leq}n} \qquad \frac{p \in m{\leq}n}{\mathrm{leS}\, p \in (\mathrm{s}\, m){\leq}(\mathrm{s}\, n)}$$

Pattern matching for $\leq$

Pattern matching for $\leq$

$$\mathrm{transLe} \in \Pi i, j, k \in \mathrm{Nat}.(i{\leq}j) \to (j{\leq}k) \to i{\leq}k$$

Pattern matching for $\leq$

$$\mathrm{transLe} \in \Pi i, j, k \in \mathrm{Nat}.(i{\leq}j){\to}(j{\leq}k){\to}i{\leq}k$$

where

$$\begin{array}{lcl} \mathrm{transLe}\, 0\, j\, k\, (\mathrm{le0}\, j)\, q & = & \mathrm{le0}\, k \\ \mathrm{transLe}\, (\mathrm{s}\, i)\, (\mathrm{s}\, j)\, (\mathrm{s}\, k)\, (\mathrm{leS}\, p)\, (\mathrm{leS}\, q) & = & \mathrm{leS}\, (\mathrm{transLe}\, i\, j\, k\, p\, q) \end{array}$$

Leq in LEGO

```
Inductive [Leq : Nat → Nat → Set]
Constructors
          [le0 : {n:Nat}Leq ze n]
          [leS : {m,n|Nat}(Leq m n)
                        → (Leq (su m) (su n))];
```

Elimination for `Leq`

```
decl  Leq_elim :
      {C_Leq:{x1,x2|Nat}(Leq x1 x2)→ TYPE}
          ({n:Nat}C_Leq (le0 n))→
          ({m,n|Nat}{x1:Leq m n}(C_Leq x1)→ C_Leq (leS x1))→
        {x1,x2|Nat}{z:Leq x1 x2}C_Leq z

[[C_Leq:{x1,x2|Nat}(Leq x1 x2)→ TYPE][f_le0:{n1:Nat}C_Leq (le0 n1)]
 [f_leS:{m,n|Nat}{x1:Leq m n}(C_Leq x1)→ C_Leq (leS x1)][n1:Nat][m,n|Nat
 [x1:Leq m n]
    Leq_elim C_Leq f_le0 f_leS (le0 n1)  ⇒  f_le0 n1
 || Leq_elim C_Leq f_le0 f_leS (leS x1)  ⇒
    f_leS x1 (Leq_elim C_Leq f_le0 f_leS x1)]
```

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

- 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