Dependent Types and Irrelevance

Christoph-Simon Senjak

Technische Universität München
Institut für Informatik
Boltzmannstraße 3
85748 Garching

PUMA Workshop September 2012
Dependent Types

Types may depend on other types and even terms.

Types can be defined inductively.

Usage is still similar to common functional type systems (Haskell, SML).
Dependent Types

- Types may depend on other types and even terms
Dependent Types

- Types may depend on other types and even terms
- Types can be defined inductively
Dependent Types

- Types may depend on other types and even terms
- Types can be defined inductively
- Usage is still similar to common functional type systems (Haskell, SML)
Example

We can define the type of lists of a certain length:

\[
\text{data } \text{List} : \text{Set} \to \text{Nat} \to \text{Set} \quad \text{where}
\]

\[
\text{nil} : (A : \text{Set}) \to \text{List } A \ 0
\]

\[
\text{cons} : (A : \text{Set}) \to (n : \text{Nat}) \to A \to \text{List } A \ n \to \text{List } A \ (S \ n)
\]

We can define the type of pairs \((n, m) | n \leq m\):

\[
\text{data } \leq : \text{Nat} \to \text{Nat} \to \text{Set} \quad \text{where}
\]

\[
z : (n : \text{Nat}) \to 0 \leq n
\]

\[
q : (n : \text{Nat}) \to (m : \text{Nat}) \to n \leq m \to (S \ n) \leq (S \ m)
\]
Example

We can define the Type of lists of a certain length:

data List : Set → Nat → Set1 where
  nil : (A : Set) → List A 0
  cons : (A : Set) → (n : Nat) → A → List A n
       → List A (S n)
Example

We can define the Type of lists of a certain length:

\[
data \text{List} : \text{Set} \to \text{Nat} \to \text{Set}\text{1} \text{ where}
\]
\[
nil : (A : \text{Set}) \to \text{List} A 0
\]
\[
\text{cons} : (A : \text{Set}) \to (n : \text{Nat}) \to A \to \text{List} A n
\]
\[
\to \text{List} A (S n)
\]

We can define the type of pairs \(\{(n, m) | n \leq m\}\):

\[
data \_\_\leq\_ : \text{Nat} \to \text{Nat} \to \text{Set} \text{ where}
\]
\[
z : (n : \text{Nat}) \to 0 \leq n
\]
\[
q : (n : \text{Nat}) \to (m : \text{Nat}) \to n \leq m \to
\]
\[
(S n) \leq (S m)
\]
Applications

Vector Multiplication:
\[ \text{vecMult} : (n : \text{Nat}) \rightarrow \text{List Nat n} \rightarrow \text{List Nat n} \rightarrow \text{Nat} \]

Bounds checking:
\[ \text{nth} : (A : \text{Set}) \rightarrow (n, m : \text{Nat}) \rightarrow (S m) \leq n \rightarrow \text{List A n} \rightarrow A \]
Applications

Vector Multiplication:

\[ \text{vecMult} : (n : \text{Nat}) \rightarrow \text{List Nat n} \rightarrow \text{List Nat n} \rightarrow \text{Nat} \]
Applications

Vector Multiplication:

\[ \text{vecMult} : \ (n : \ Nat) \to \ \text{List Nat n} \to \ \text{List Nat n} \to \ Nat \]

Bounds checking:

\[ \text{nth} : \ (A : \ Set) \to \ (n, m : \ Nat) \to \ (S m) \leq n \to \ \text{List A n} \to \ A \]
Automatic unification

Many of the arguments we gave can be derived automatically, and may be omitted for convenience in most languages:

\[
\text{cons} : \text{A} \rightarrow \text{List A} \\
\text{vecMult} : \text{List Nat} \times \text{List Nat} \rightarrow \text{Nat} \\
\text{nth} : (m : \text{Nat}) \rightarrow (S \text{m}) \leq \text{n} \rightarrow \text{List A} \rightarrow \text{A}
\]
Many of the arguments we gave can be derived automatically, and may be omitted for convenience in most languages:
Automatic unification

Many of the arguments we gave can be derived automatically, and may be omitted for convenience in most languages:

\[
\text{cons} : \ A \rightarrow \text{List} \ A \ n \rightarrow \text{List} \ A \ (S \ n)
\]
Automatic unification

Many of the arguments we gave can be derived automatically, and may be omitted for convenience in most languages:

\[
\text{cons : } A \rightarrow \text{List } A \ n \rightarrow \text{List } A \ (S \ n) \\
z : \ 0 \leq n
\]
Automatic unification

Many of the arguments we gave can be derived automatically, and may be omitted for convenience in most languages:

\[ \text{cons} : \ A \rightarrow \text{List } A \ n \rightarrow \text{List } A \ (S \ n) \]
\[ \text{z} : \ 0 \leq n \]
\[ \text{vecMult} : \ \text{List } \text{Nat } n \rightarrow \text{List } \text{Nat } n \rightarrow \text{Nat} \]
Many of the arguments we gave can be derived automatically, and may be omitted for convenience in most languages:

\[\begin{align*}
\text{cons} : & \quad A \to \text{List } A \, n \to \text{List } A \, (S \, n) \\
\text{z} : & \quad 0 \leq n \\
\text{vecMult} : & \quad \text{List } \text{Nat } n \to \text{List } \text{Nat } n \to \text{Nat} \\
\text{nth} : & \quad (m : \text{Nat}) \to (S \, m) \leq n \to \text{List } A \, n \to A
\end{align*}\]
Dependently typed programming languages implement the BHK-Interpretation of intuitionistic logic (cf. Curry-Howard-Isomorphism). Therefore, they can be used to check constructive mathematical proofs.

Mathematical implication can be realized by non-dependent abstraction, \( A \rightarrow B \) corresponds to \( A \rightarrow B \).

Mathematical universal quantification can be realized by dependent abstraction, \((x:A) \rightarrow B\) corresponds to \(\forall x : A. B\).

Other logical connectives can be defined inductively. Recursion corresponds to induction.
Dependently typed programming languages implement the BHK-Interpretation of intuitionistic logic (cf. Curry-Howard-Isomorphism). Therefore, they can be used to check constructive mathematical proofs.

Conversely, they can be used to formalize mathematical properties of a program, like in the former example $\leq$. 
Dependently typed programming languages implement the BHK-Interpretation of intuitionistic logic (cf. Curry-Howard-Isomorphism). Therefore, they can be used to check constructive mathematical proofs.

Conversely, they can be used to formalize mathematical properties of a program, like in the former example \( \leq \).

Mathematical implication can be realized by non-dependent abstraction, \( A \rightarrow B \) corresponds to \( A \rightarrow B \).

Mathematical universal quantification can be realized by dependent abstraction, \( (x:A) \rightarrow B \) corresponds to \( \forall x:A B \).
Dependently typed programming languages implement the BHK-Interpretation of intuitionistic logic (cf. Curry-Howard-Isomorphism). Therefore, they can be used to check constructive mathematical proofs.

Conversely, they can be used to formalize mathematical properties of a program, like in the former example $\leq$.

Mathematical implication can be realized by non-dependent abstraction, $A \rightarrow B$ corresponds to $A \rightarrow B$.

Mathematical universal quantification can be realized by dependent abstraction, $(x:A) \rightarrow B$ corresponds to $\forall_{x:A} B$.

Other logical connectives can be defined inductively. Recursion corresponds to induction.
Irrelevance

Anything that is unrelated to elephants is irrelephant.
Often, the information given in the types is only necessary to check correctness. It is not important for the actual program to work (comparable to C’s “no type information at runtime”-policy).
Irrelevance

- Often, the information given in the types is only necessary to check correctness. It is not important for the actual program to work (comparable to C’s “no type information at runtime”-policy).
- It is called **computationally irrelevant**.
Irrelevance

- Often, the information given in the types is only necessary to check correctness. It is not important for the actual program to work (comparable to C’s “no type information at runtime”-policy).
- It is called **computationally irrelevant**.
- We want to “prune” the terms before we compile them, to get more efficient programs.
Irrelevance

- Often, the information given in the types is only necessary to check correctness. It is not important for the actual program to work (comparable to C’s “no type information at runtime”-policy).
- It is called **computationally irrelevant**.
- We want to “prune” the terms before we compile them, to get more efficient programs.
- Additionally, we might want to prevent a program from using a certain value.
Example

We can define the $\text{nth}$-function by

\[
\text{nth} : (A : \text{Set}) \rightarrow (n \ m : \text{Nat}) \rightarrow ((S \ m) \leq n) \\
\quad \rightarrow (\text{List } A \ n) \rightarrow A
\]

\[
\text{nth } A \ 0 \ m \ () \ l \ -- \ \text{absurd case}
\]

\[
\text{nth } A \ (S \ n) \ 0 \ (p \ .0 \ .n \ (z \ .n)) \ (\text{cons } A \ a \ .n \ l) = a
\]

\[
\text{nth } A \ (S \ n) \ (S \ m) \ (p \ .(S \ m) \ .n \ q) \ (\text{cons } A \ a \ .n \ l) =
\quad \text{nth } A \ n \ m \ q \ l
\]
Example

We can define the \( \text{nth} \)-function by

\[
\text{nth} : (A: \text{Set}) \to (n, m : \text{Nat}) \to ((S \ m) \leq n) \\
\to (\text{List} \ A \ n) \to A
\]

\[
\text{nth} \ A \ 0 \ m \ () \ l \ -- \ \text{absurd case}
\]

\[
\text{nth} \ A \ (S \ n) \ 0 \ (p.0.n(z.n)) \ (\text{cons} \ .A \ a .n \ l) = a
\]

\[
\text{nth} \ A \ (S \ n) \ (S \ m) \ (p.(S \ m).n \ q) \ (\text{cons} \ .A \ a .n \ l) =
\]

\[
\text{nth} \ A \ n \ m \ q \ l
\]

However, this requires the creation of a proof for \((S \ m) \leq n\) at every call of \(\text{nth}\), though we do not need the actual proof. So we can make it irrelevant:

\[
\text{nth} : (A: \text{Set}) \to (n, m : \text{Nat}) \to
\]

\[
(q \div ((S \ m) \leq n)) \to (\text{List} \ A \ n) \to A
\]
Example

We can define the $\text{nth}$-function by
\[
\text{nth} : (A : \text{Set}) \to (n \ m : \text{Nat}) \to ((S \ m) \leq n) \\
\to (\text{List} A n) \to A
\]
\[
\text{nth} A 0 \ m \ () \ l \ -- \ absurd \ case
\]
\[
\text{nth} A (S \ n) 0 \ (p \ .0 \ .n \ (z \ .n)) \ (\text{cons} \ .A \ a \ .n \ l) = a
\]
\[
\text{nth} A (S \ n) (S \ m) \ (p \ (S \ m) \ .n \ q) \ (\text{cons} \ .A \ a \ .n \ l) =
\]
\[
\text{nth} A n \ m \ q \ l
\]
However, this requires the creation of a proof for $(S \ m) \leq n$ at every call of $\text{nth}$, though we do not need the actual proof. So we can make it irrelevant:

\[
\text{nth} : (A : \text{Set}) \to (n \ m : \text{Nat}) \to
\]
\[
(q \div ((S \ m) \leq n)) \to (\text{List} A n) \to A
\]
The rest of the definition stays the same, but the part of the function which is actually compiled now looks like
\[
\text{nth} A 0 \ m \ l
\]
\[
\text{nth} A (S \ n) 0 \ (\text{cons} \ .A \ a \ .n \ l) = a
\]
\[
\text{nth} A (S \ n) (S \ m) \ (\text{cons} \ .A \ a \ .n \ l) = \text{nth} A n \ m \ l
\]
Example 2

append1' : (n : Nat) -> List Nat n  
          -> List Nat (S n)
append1' n L = map (\x -> if (x == n) then 1  
                       else (nth x L)) [ x | x <- 0..n ]

Appends 1 to the end of the list. Uses the length of the list explicitly, n is relevant.
Example 2

\[
\text{append1}' : (n : \text{Nat}) \to \text{List Nat } n \\
\to \text{List Nat } (S \ n)
\]

\[
\text{append1}' \ n \ L = \text{map } (x \to \text{if } (x == n) \text{ then } 1 \\
\text{else } (\text{nth } x \ L)) \ [ x \mid x \leftarrow 0..n ]
\]

Appends 1 to the end of the list. Uses the length of the list explicitly, \(n\) is relevant.

\[
\text{append1} : (n \div \text{Nat}) \to \text{List Nat } n \to \text{List Nat } (S \ n)
\]

\[
\text{append1} \ 0 \ (\text{nil Nat}) = \text{cons } 1 \ \text{Nat } 1 \ (\text{nil Nat})
\]

\[
\text{append1} \ (S \ n) \ (\text{cons } (S \ n) \ \text{Nat } m \ L) = \\
\text{cons } (S \ (S \ n)) \ \text{Nat } n \ (\text{append1} \ L)
\]

Same guarantees as the first function, but does not make explicit use of the length. This might be desirable when the list is a stream.
Shape Irrelevance

Recall our definition of lists:

\[
data \text{ List} : \text{ Set} \rightarrow \text{ Nat} \rightarrow \text{ Set}
\]

- \( \text{nil} : (A \colon \text{ Set}) \rightarrow \text{ List } A \ 0 \)
- \( \text{cons} : (A \colon \text{ Set}) \rightarrow (n \colon \text{ Nat}) \rightarrow A \rightarrow \text{ List } A \ n \rightarrow \text{ List } A \ (S \ n) \)

Usually, the element type and the length of the list are not needed at runtime. We would like to make them irrelevant:

\[
data \text{ List} : (A \div \text{ Set}) \rightarrow (n \div \text{ Nat}) \rightarrow \text{ Set}
\]

- \( \text{nil} : (A \div \text{ Set}) \rightarrow \text{ List } A \ 0 \)
- \( \text{cons} : (A \div \text{ Set}) \rightarrow (n \div \text{ Nat}) \rightarrow A \rightarrow \text{ List } A \ n \rightarrow \text{ List } A \ (S \ n) \)
Shape Irrelevance

Recall our definition of lists:

\[
\text{data List : Set} \rightarrow \text{Nat} \rightarrow \text{Set1} \text{ where}
\]
\[
\text{nil : (A : Set) } \rightarrow \text{List A 0}
\]
\[
\text{cons : (A : Set) } \rightarrow \text{(n : Nat) } \rightarrow \text{A } \rightarrow \text{List A n}
\]
\[
\rightarrow \text{List A (S n)}
\]
Shape Irrelevance

Recall our definition of lists:

\[
\text{data List} : \text{Set} \to \text{Nat} \to \text{Set1} \text{ where}
\]
\[
\begin{align*}
\text{nil} : & (A : \text{Set}) \to \text{List A} 0 \\
\text{cons} : & (A : \text{Set}) \to (n : \text{Nat}) \to A \to \text{List A n} \\
& \quad \to \text{List A (S n)}
\end{align*}
\]

Usually, the element type and the length of the list are not needed at runtime. We would like to make them irrelevant:

\[
\text{data List} : (A \div \text{Set}) \to (n \div \text{Nat}) \to \text{Set1} \text{ where}
\]
\[
\begin{align*}
\text{nil} : & (A \div \text{Set}) \to \text{List A} 0 \\
\text{cons} : & (A \div \text{Set}) \to (n \div \text{Nat}) \to A \to \text{List A n} \\
& \quad \to \text{List A (S n)}
\end{align*}
\]
The first definition of a list would save its type and its length into every cons cell. A three element list would therefore look like
The second definition removes much of this.

\[
\text{nil} \\
\uparrow \\
x \leftarrow \text{cons } x \ [ ] \\
\uparrow \\
y \leftarrow \text{cons } y \ [x] \\
\uparrow \\
\text{cons } z \ [x, y] \\
\leftarrow \\
z
\]
Shape Irrelevance

The problem with this definition is that it produces a contradiction. We have the rule

\[
\begin{align*}
\vdash t &: T \\
\vdash T &= U : Set
\end{align*}
\]

\[
\frac{t : U}{t : U}
\]
Shape Irrelevance

The problem with this definition is that it produces a contradiction. We have the rule

\[ \vdash t : T \quad \vdash T = U : Set \quad \frac{}{t : U} \]

Since we have

\( (\text{cons} \Nat 1 0 (\text{nil} \Nat)) : (\text{List} \Nat 0) \)

and because of irrelevance

\( (\text{List} \Nat 0) = \text{List} \perp 0 \)

we have an inhabitant of the absurd type

\( \text{head} (\text{cons} \Nat 1 0 (\text{nil} \Nat)) : \perp \)

therefore

\( 0 : \perp \)
Shape Irrelevance

Idea: Define a new kind of irrelevance “between” irrelevance and relevance.
Shape Irrelevance

Idea: Define a new kind of irrelevance "between" irrelevance and relevance.
We may define the shape of a type by disregarding its irrelevant arguments, and include a new mode of equality, only regarding $\beta\eta$-equivalence with shapes.
Shape Irrelevance

Idea: Define a new kind of irrelevance “between” irrelevance and relevance.
We may define the *shape* of a type by disregarding its irrelevant arguments, and include a new mode of equality, only regarding $\beta\eta$-equivalence with shapes.
Also, we might define a more general notion of several kinds of irrelevance regarding other equivalence relations.