Dependent Types and Irrelevance

Christoph-Simon Senjak

Technische Universität München
Institut für Informatik
Boltzmannstraße 3
85748 Garching

PUMA Workshop September 2012

Dependent Types

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

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

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

```
data _≦_ :  Nat → Nat → Set where
   z :  (n :  Nat) → 0 ≦ n
   q :  (n :  Nat) → (m :  Nat) → n ≦ m →
        (S n) ≦ (S m)
```

Applications

Applications

Vector Multiplication:

```
vecMult : (n : Nat) → List Nat n → List Nat n
          → Nat
```

Applications

Vector Multiplication:

```
vecMult : (n : Nat) → List Nat n → List Nat n
          → Nat
```

Bounds checking:

```
nth : (A : Set) → (n, m : Nat) → (S m) ≦ n
      → List A n → A
```

Automatic unification

Automatic unification

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:

```
cons :  A → List A n → 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:

$\texttt{cons} : \ \texttt{A} \to \texttt{List A n} \to \texttt{List A (S n)}$

$\texttt{z} : \ \texttt{0} \leqq \texttt{n}$

Automatic unification

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

```
cons :  A → List A n → List A (S n)
z :  0 ≤ n
vecMult :  List Nat n → List Nat n → Nat
```

Automatic unification

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

$\texttt{cons : A} \to \texttt{List A n} \to \texttt{List A (S n)}$
$\texttt{z : 0} \leqq \texttt{n}$
$\texttt{vecMult : List Nat n} \to \texttt{List Nat n} \to \texttt{Nat}$
$\texttt{nth : (m : Nat)} \to \texttt{(S m)} \leqq \texttt{n} \to \texttt{List A n} \to \texttt{A}$

Mathematics

- 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.

Mathematics

- 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 $\leq$.

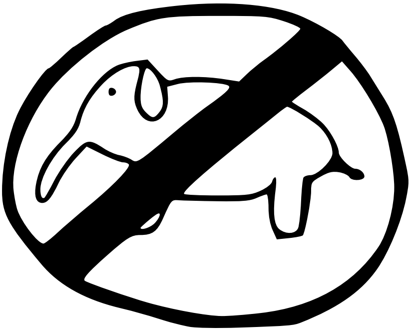
Mathematics

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

Mathematics

- 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 $\leqq$.
- Mathematical implication can be realized by non-dependent abstraction, `A → B` corresponds to $A \to B$.
- Mathematical universal quantification can be realized by dependent abstraction, `(x:A) → 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.

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).

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 `nth`-function by

```
nth :  (A : Set) → (n m :  Nat) → ((S m) ≤ n)
     → (List A n) → A
nth A 0 m () l -- absurd case
nth A (S n) 0 (p .0 .n (z .n)) (cons .A a .n l) = a
nth A (S n) (S m) (p .(S m) .n q) (cons .A a .n l) =
       nth A n m q l
```

Example

We can define the `nth`-function by

```
nth : (A : Set) → (n m : Nat) → ((S m) ≤ n)
     → (List A n) → A
nth A 0 m () l -- absurd case
nth A (S n) 0 (p .0 .n (z .n)) (cons .A a .n l) = a
nth A (S n) (S m) (p .(S m) .n q) (cons .A a .n l) =
       nth A n m q l
```

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

```
nth : (A : Set) → (n m : Nat) →
     (q ÷ ((S m) ≤ n)) → (List A n) → A
```

Example

We can define the `nth`-function by

```
nth :  (A : Set) → (n m :  Nat) → ((S m) ≤ n)
     → (List A n) → A
nth A 0 m () l -- absurd case
nth A (S n) 0 (p .0 .n (z .n)) (cons .A a .n l) = a
nth A (S n) (S m) (p .(S m) .n q) (cons .A a .n l) =
       nth A n m q l
```

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

```
nth :  (A : Set) → (n m :  Nat) →
     (q ÷ ((S m) ≤ n)) → (List A n) → A
```

The rest of the definition stays the same, but the part of the function which is actually compiled now looks like

```
nth A 0 m l
nth A (S n) 0 (cons .A a .n l) = a
nth A (S n) (S m) (cons .A a .n l) = 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

```
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.

```
append1 :  (n ÷ Nat) → List Nat n → List Nat (S n)
append1 0 (nil Nat) = cons 1 Nat 1 (nil Nat)
append1 (S n) (cons (S n) Nat m L) =
        cons (S (S n)) Nat n (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

Shape Irrelevance

Recall our definition of lists:

```
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)
```

Shape Irrelevance

Recall our definition of lists:

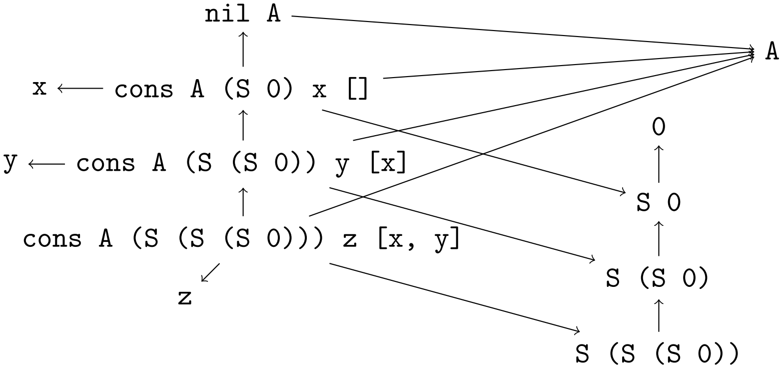
```
data List : Set → Nat → Set1 where
  nil : (A : Set) → List A 0
  cons : (A : Set) → (n : Nat) → A → List A n
          → List A (S n)
```

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

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

Example

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



Example

The second definition removes much of this.

$$
\begin{tikzcd}
 & & \texttt{nil} \\
\texttt{x} & & \texttt{cons x []} \arrow[u] \arrow[ll] \\
\texttt{y} & & \texttt{cons y [x]} \arrow[u] \arrow[ll] \\
 & & \texttt{cons z [x, y]} \arrow[u] \arrow[dl] \\
 & \texttt{z} &
\end{tikzcd}
$$

Shape Irrelevance

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

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

Shape Irrelevance

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

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

Since we have

```
(cons Nat 1 0 (nil Nat)) :  (List Nat 0)
```

and because of irrelevance

```
(List Nat 0) = List ⊥ 0
```

we have an inhabitant of the absurd type

```
head (cons Nat 1 0 (nil Nat)) :  ⊥
```

therefore

```
0 :  ⊥
```

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.