Dependent types

Lecture 12 of CSE 3100
Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

Lecture plan

- What’s a dependent type?
- Dependent function types
- The Vector and Fin types
- Well-typed syntax

What’s a dependent type?

Cooking with dependent types (1/3)

Suppose we are implementing a cooking assistant that can help with preparing three kinds of food:

```
data Food : Set where
  pizza : Food
  cake  : Food
  bread : Food
```

We want to implement a function amountOfCheese : Food $\rightarrow$ Nat that computes how much cheese is needed.

Problem: How can we make sure this function is never called with argument cake?

Cooking with dependent types (2/3)

Solution. We can make the type Food more precise making it into an indexed datatype:

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Flavour} : \mathsf{Set}\ \mathsf{where}\\
&\quad \mathsf{cheesy} : \mathsf{Flavour}\\
&\quad \mathsf{chocolatey} : \mathsf{Flavour}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Food} : \mathsf{Flavour} \to \mathsf{Set}\ \mathsf{where}\\
&\quad \mathsf{pizza} : \mathsf{Food}\ \mathsf{cheesy}\\
&\quad \mathsf{cake} : \mathsf{Food}\ \mathsf{chocolatey}\\
&\quad \mathsf{bread} : \{f : \mathsf{Flavour}\} \to \mathsf{Food}\ f
\end{aligned}
$$

This defines two types Food cheesy and Food chocolatey.

Cooking with dependent types (3/3)

We can now rule out invalid inputs by using the more precise type Food cheesy:

```
amountOfCheese : Food cheesy → Nat
amountOfCheese pizza  = 100
amountOfCheese bread = 20
```

The coverage checker of Agda knows that cake is not a valid input!

Dependent type theory (1972)

Per
Martin-Löf

A dependent type is a family of types, depending on a term of a base type.

Dependent type theory (1972)

Per
Martin-Löf

A dependent type is a family of types, depending on a term of a base type.

Example. Food is a dependent type indexed over the base type Flavour.

Dependent types vs. parametrized types

Question. What is the difference between a dependent type such as $\mathsf{Food}\ f$ and a parametrized type such as $\mathsf{Maybe}\ a$?

Dependent types vs. parametrized types

Question. What is the difference between a dependent type such as Food f and a parametrized type such as Maybe a?

Answer. All types Maybe a have the same constructors (Nothing and Just) for all values of a is, while Food f has different constructors depending on f.

The Vec type

Vectors: lists that know their length

Vec A n is the type of vectors with exactly n arguments of type A:

```
myVec1 : Vec Nat 4
myVec1 = 1 :: 2 :: 3 :: 4 :: []

myVec2 : Vec Nat 0
myVec2 = []

myVec3 : Vec (Bool → Bool) 2
myVec3 = not :: id :: []
```

Definition of the `Vec` type

Vec A n is a dependent type indexed over the base type Nat:

```
data Vec (A : Set) : Nat → Set where
  []    : Vec A 0
  _::_ : {n : Nat} →
    A → Vec A n → Vec A (suc n)
```

This has two constructors [] and _::_ like List, but the constructors specify the length in their types.

Parameters vs. indices

The argument ($A : Set$) in the definition of Vec is a parameter, and has to be *the same in the type of each constructor*.

The argument of type Nat in the definition of Vec is an index, and must be *determined individually for each constructor*.

Quiz question

Question. How many elements are there in the type Vec Bool 3?

Quiz question

Question. How many elements are there in the type Vec Bool 3?

Answer. 8 elements:

- true :: true :: true :: []
- true :: true :: false :: []
- true :: false :: true :: []
- true :: false :: false :: []
- false :: true :: true :: []
- false :: true :: false :: []
- false :: false :: true :: []
- false :: false :: false :: []

Type-level computation

During type-checking, Agda will evaluate expressions in types:

```
myVec4 : Vec Nat (2 + 2)
myVec4 = 1 :: 2 :: 3 :: 4 :: []
```

Every type is equal to its normal form:
Vec Nat (2 + 2) is the same type as Vec Nat 4.

Since Agda is total, every type has a unique normal form!

Checking the length of a vector

Constructing a vector of the wrong length in any way is a type error:

```
myVec5 : Vec Nat 0
myVec5 = 1 :: 2 :: []
```

```
suc _n_46 != zero of type Nat
when checking that the inferred
type of an application
Vec Nat (suc _n_46)
matches the expected type
Vec Nat 0
```

Dependent function types

A dependent function type is a type of the form $(x : A) \to B\ x$ where the *type* of the output depends on the *value* of the input.

Example.

$$\begin{aligned} &\mathrm{zeroes} : (n : \mathrm{Nat}) \to \mathrm{Vec}\ \mathrm{Nat}\ n \\ &\mathrm{zeroes}\ \mathrm{zero} \quad\;\; = [\,] \\ &\mathrm{zeroes}\ (\mathrm{suc}\ n) = 0 :: \mathrm{zeroes}\ n \end{aligned}$$

E.g. zeroes 3 has type Vec Nat 3 and evaluates to $0 :: 0 :: 0 :: [\,]$.

Quiz question

Question. What's 'dependent' about a dependent function type?

1. The value of the output depends on the value of the input
2. The value of the output depends on the type of the input
3. The type of the output depends on the value of the input
4. The type of the output depends on the type of the input

Concatenation of vectors

We can pattern match on Vec just like on List:

$$\begin{aligned}
&\mathsf{mapVec} : \{A\ B : \mathsf{Set}\}\ \{n : \mathsf{Nat}\} \to \\
&\quad (A \to B) \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ B\ n \\
&\mathsf{mapVec}\ f\ [] \qquad\qquad = [] \\
&\mathsf{mapVec}\ f\ (x :: xs) = f\ x :: \mathsf{mapVec}\ f\ xs
\end{aligned}$$

Note. The type of mapVec specifies that the output has the same length as the input.

A safe `head` function

By making the input type of a function more precise, we can rule out certain cases statically (= during type checking):

$$\mathsf{head} : \{A : \mathsf{Set}\}\{n : \mathsf{Nat}\} \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A$$
$$\mathsf{head}\ (x :: xs) = x$$

Agda knows the case for head [] is impossible!
(just like for amountOfCheese cake)

A safe `tail` function

Question. What should be the type of tail on vectors with the following definition?

```
tail (x :: xs) = xs
```

A safe `tail` function

Question. What should be the type of tail on vectors with the following definition?

$$\mathsf{tail}\ (x :: xs) = xs$$

Answer.

$$\mathsf{tail} : \{A : \mathsf{Set}\}\ \{n : \mathsf{Nat}\} \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to \mathsf{Vec}\ A\ n$$
$$\mathsf{tail}\ (x :: xs) = xs$$

Live coding

Exercise. Define a function zipVec that only accepts vectors of the same length.

The Fin type

A safe lookup

By combining head and tail, we can get the 1st, 2nd, 3rd,… element of a vector with at least that many elements.

How can we define a function lookupVec that get the element at position i of a Vec A n where $i < n$?

Note. We want to get an element of A, *not* of Maybe A!

The `Fin` type

We need a type of indices that are *safe* for a vector of length n, i.e. numbers between 0 and $n - 1$.

This is the type Fin n of finite numbers:

```
zero3 one3 two3 : Fin 3
zero3 = zero
one3  = suc zero
two3  = suc (suc zero)
```

Definition of the `Fin` type

```
data Fin : Nat → Set where
  zero : {n : Nat} → Fin (suc n)
  suc  : {n : Nat} → Fin n → Fin (suc n)
```

An empty type

Fin *n* has *n* elements, so in particular Fin 0 has *zero* elements: it is an empty type.

This means there are *no valid indices* for a vector of length 0.

Note. Unlike in Haskell, we cannot even construct an expression of Fin 0 using `undefined` or an infinite loop.

The family of `Fin` types

$$\begin{tikzcd}
 & & & & \mathsf{suc} \arrow[dl] & \ldots \\
 & & & \mathsf{suc} \arrow[dl] & \mathsf{suc} \arrow[dl] & \ldots \\
 & & \mathsf{suc} \arrow[dl] & \mathsf{suc} \arrow[dl] & \mathsf{suc} \arrow[dl] & \ldots \\
 & \mathsf{zero} & \mathsf{zero} & \mathsf{zero} & \mathsf{zero} & \ldots \\
\mathrm{Fin}\ 0 & \mathrm{Fin}\ 1 & \mathrm{Fin}\ 2 & \mathrm{Fin}\ 3 & \mathrm{Fin}\ 4 & \cdots
\end{tikzcd}$$

A safe lookup (1/5)

```
lookupVec : {A : Set} {n : Nat} →
  Vec A n → Fin n → A
lookupVec xs i = {!  !}
```

A safe lookup (2/5)

```
lookupVec : {A : Set} {n : Nat} →
  Vec A n → Fin n → A
lookupVec (x :: xs) i = {!  !}
```

A safe lookup (3/5)

```
lookupVec : {A : Set} {n : Nat} →
  Vec A n → Fin n → A
lookupVec (x :: xs) zero    = {!  !}
lookupVec (x :: xs) (suc i) = {!  !}
```

A safe lookup (4/5)

```
lookupVec : {A : Set} {n : Nat} →
  Vec A n → Fin n → A
lookupVec (x :: xs) zero    = x
lookupVec (x :: xs) (suc i) = {!  !}
```

A safe lookup (5/5)

$$
\begin{aligned}
&\mathsf{lookupVec} : \{A : \mathsf{Set}\}\ \{n : \mathsf{Nat}\} \to \\
&\quad \mathsf{Vec}\ A\ n \to \mathsf{Fin}\ n \to A \\
&\mathsf{lookupVec}\ (x :: xs)\ \mathsf{zero} \quad\ \ = x \\
&\mathsf{lookupVec}\ (x :: xs)\ (\mathsf{suc}\ i) = \mathsf{lookupVec}\ xs\ i
\end{aligned}
$$

We now have a safe and total version of the Haskell `(!!)` function, without having to change the return type in any way.

Live coding exercise (1/2)

Define a datatype Expr of expressions of a small programming language with:

- Number literals $0, 1, 2, \ldots$
- Arithmetic expressions $e_1 + e_2$ and $e_1 * e_2$
- Booleans true and false
- Comparisons $e_1 < e_2$ and $e_1 == e_2$
- Conditionals `if` e_1 `then` e_2 `else` e_3

Expr should be a *dependent type* indexed over the type Ty of possible types of this language:

```
data Ty : Set where
  tInt  : Ty
  tBool : Ty
```

Live coding exercise (2/2)

Next, write a function $\mathsf{El} : \mathsf{Ty} \to \mathsf{Set}$ that interprets a type of this language as an Agda type.

Finally, define $\mathsf{eval} : \{t : \mathsf{Ty}\} \to \mathsf{Expr}\ t \to \mathsf{El}\ t$ that evaluates a given expression to an Agda value.

Dependent types: Summary

A dependent type is a type that *depends on* a value of some base type.

With dependent types, we can specify the allowed inputs of a function more precisely, ruling out invalid inputs at compile time.

Examples of dependent types.

- Food f, indexed over f : Flavour
- Vec A n, indexed over n : Nat
- Fin n, indexed over n : Nat
- Expr t, indexed over t : Ty

What's next?

Next lecture: Using Agda as a theorem prover

To do:

- Read the lecture notes:
    - This lecture: section 2 of Agda lecture notes
    - Next lecture: section 3 of Agda lecture notes
- Do Weblab exercises on dependent types
- Continue hacking on the project and asking questions on TU Delft Answers