Dependently Typed Programming

Shin-Cheng Mu

FLOLAC, 2016

Types as Specifications

- The type of a function specifies properties it should satisfy.
- The type checker verifies our claim that the function does indeed has the property.
- The more expressive the type system is, the more we can say in a type.

Some Notes on Conventions

- Note that "f has type τ" is usually denoted $f : \tau$. Haskell, due to a historical mistake, uses $f :: \tau$. We will use a single colon from now on.
- For reason we will not explain here, a type in some dependently typed languages is called a "Set". We will use the convention here.

Some Specifications

- Monomorphic: $f : [\mathit{Int}] \to \mathit{Int}$.
  - f takes a list of numbers and returns a number.
- Polymorphic: $f : \forall a\ b \to [a] \to [b] \to [(a, b)]$.
  - A correct implementation of f must not inspect the contents of the list.
  - Note that $a : \mathit{Set}$ (type), and $[\_] : \mathit{Set} \to \mathit{Set}$.
- 2nd-rank polymorphism: $f : \forall a \to (\forall s \to s \to (s, a)) \to a$.
  - s must not be shared!

Some More

- If we denote "the type of lists whose elements are of type a and whose length is n" by $[a]_n$, we have $(+\!\!+) : \forall a\ m\ n \to [a]_m \to [a]_n \to [a]_{m+n}$.
  - Notice: $[\_]_\_ : \mathit{Set} \to \mathbb{N} \to \mathit{Set}$.
  - It's a *dependent type* — a type that depends on values!
- The function *sort* typically has type $[\mathbb{N}] \to [\mathbb{N}]$.
  - What about $\mathit{sort} : (\mathit{xs} : [\mathbb{N}]) \to (\mathit{ys}, \mathit{perm}\ \mathit{xs}\ \mathit{ys} \wedge \mathit{ordered}\ \mathit{ys})$?
  - The type says that a correct implementation of *sort* must, of course, sort!

Dependent Type

- So called because a type may depend on a value.
- Very expressive — with it a lot can be said.
- But not that *accessible* yet — we are still learning how to actually use it effectively in programming.
- Thus many theorem provers / programming languages have been developed, aiming to bridge the gap.

Dependently Typed Languages

- Coq ('89-).
- Cayenne (Augustsson '98).
- Dependent ML / ATS (Pfenning & Xi '98).
- Epigram (McBride & McKinna '04).
- Agda 2 (Norell & Danielsson '05).
- Meanwhile, Haskell also gradually supports more and more dep. type-like features. GADT, type families...

1 A Quick Introduction to Agda

A Simple Algebraic Type

- In Haskell we write

$$\textbf{data}\ \mathit{Bool}\ =\ \mathit{True} \mid \mathit{False}\quad .$$

- The constructors have types $\mathit{true} : \mathit{Bool}$ and $\mathit{false} : \mathit{Bool}$.

- In Agda we write

$$\begin{array}{l}\textbf{data}\ \mathit{Bool} : \mathit{Set}\ \textbf{where}\\ \quad \mathit{true} : \mathit{Bool}\\ \quad \mathit{false} : \mathit{Bool}\quad ,\end{array}$$

  - which explicitly says that "*Bool* is a *Set*, with two constructors, whose types are...".
  - This so-called GADT notation may look a bit cumbersome now, but will be useful later.
  - Constructors (and types) need not start with capital letters. Values and types are treated more uniformly in a dependently typed language.

Our First Function

- Given only one *Bool*, this is probably the most interesting function:

$$\begin{array}{lcl}\mathit{not} & : & \mathit{Bool} \to \mathit{Bool}\\ \mathit{not}\ \mathit{false} & = & \mathit{true}\\ \mathit{not}\ \mathit{true} & = & \mathit{false}\quad .\end{array}$$

- See the use of pattern matching in action (and how Agda expand the cases for you).

- The type could also be written in the "named argument" notation: $\mathit{not} : (b : \mathit{Bool}) \to \mathit{Bool}$.
  - It still says that *not* maps *Bool* to *Bool*.
  - However, the first argument now has a name b, which can be used in later parts of the type. We do not need it now, but we will soon.
  - In general, if we write $f : (x : \tau) \to \sigma$, we may mention x in σ.

A Type Parameterised by a Type

- The type [_] in Haskell can be written in Agda as

$$\begin{array}{l}\textbf{data}\ \mathit{List}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where}\\ \quad [\,] \qquad : \mathit{List}\ A\\ \quad \_::\_ \quad : A \to \mathit{List}\ A \to \mathit{List}\ A\quad .\end{array}$$

- The declaration says "*List*, when given a parameter A (which is a *Set*), yields a *Set*. It has two constructors whose types are..."
  - A is in the scope of the constructors of *List*; the constructors may use A.

- $\_::\_$ is an infix operator. The underline _ marks the positions of its arguments.

Type Arguments

- "Polymorphic" functions are treated as a function that takes a type as an argument.

$$\begin{array}{lcl}\mathit{id}' & : & (A : \mathit{Set}) \to A \to A\\ \mathit{id}'\ A\ x & = & x\quad .\end{array}$$

  - Note that we are using the "named argument" notation we introduced just now.
  - "id' is a function that takes a *Set* as its argument. Call it A. It then delivers a function of type $A \to A$."

- To call id' we have to explicitly pass the type:
  - $\mathit{id}'\ \mathit{Bool}\ \mathit{true}$ evaluates to *true*.

Implicit Arguments

- It is rather cumbersome having to explicitly pass the type argument all the time. Agda allows you to declare an argument as implicit, by surrounding it in curly brackets:

$$\begin{array}{lcl}\mathit{id} & : & \{A : \mathit{Set}\} \to A \to A\\ \mathit{id}\ x & = & x\quad .\end{array}$$

- *id* is still a function that takes a type A and yields a function of type $A \to A$. But the argument A need not be given, and Agda will try to infer the value of A from its context.

- We may then call *id* in the Haskell-ish way:

  - $id\ true$ evaluates to $true$. Agda could guess that A must be $Bool$.

- Inference of implicit arguments may not always succeed! In such cases Agda marks the problematic code yellow.

- When Agda cannot infer implicit arguments, or when you just want to be clear, you may explicitly give implicit arguments using the following syntax:

  - $id\ \{Bool\}\ true$ evaluates to $true$.

- An implicit argument need not be a type. It could be a value too. Agda treat them uniformly.

$$\begin{aligned} &f\ :\ \{A : Set\} \to \{x : A\} \to \ldots x \ldots \\ &f\ =\ \ldots \end{aligned}$$

- When we need to mention an explicit argument on the RHS, we could mention it in the LHS.

$$\begin{aligned} &f\ :\ \{A : Set\} \to \{x : A\} \to \ldots \text{ use } x \text{ or } A\ \ldots \\ &f\ \{A\}\ \{x\} \ldots\ =\ \ldots \text{ use } x \text{ or } A\ \ldots \end{aligned}$$

∀-Quantification

- $\forall\ x$ is a shorter syntax for $(x : \tau)$ when τ can be inferred.

- $\forall\ \{A\}$ is a shorter syntax for $\{A : \tau\}$ when τ can be inferred.

$$\begin{aligned} &null && : \forall\ \{A\} \to List\ A \to Bool \\ &null\ [\,] && = true \\ &null\ (x :: xs) && = false\ . \end{aligned}$$

  - From the definition of **data** $List\ (A : Set) : Set \ldots$, we know that A must be a Set.

Natural Numbers

- Such an important type that we spell it out:

$$\begin{aligned} &\textbf{data}\ \mathbb{N}\ :\ Set\ \textbf{where} \\ &\quad zero\ :\ \mathbb{N} \\ &\quad suc\ \ :\ \mathbb{N} \to \mathbb{N}\ . \end{aligned}$$

- Isn't it similar to $List$?

- The function that removes the data in the list:

$$\begin{aligned} &length && :\ \forall\{A\} \to List\ A \to \mathbb{N} \\ &length\ [\,] && = zero \\ &length\ (x :: xs) && = suc\ (length\ xs)\ . \end{aligned}$$

2 Inductive Family

Vectors

- $Vec\ A\ n$ denotes the type of lists whose elements are of type A and whose length is exactly n.

$$\begin{aligned} &\textbf{data}\ Vec\ (A : Set)\ :\ \mathbb{N} \to Set\ \textbf{where} \\ &\quad [\,]\quad\ :\ Vec\ A\ zero \\ &\quad \_::\_\ :\ \{n : \mathbb{N}\} \to A \to Vec\ A\ n \to Vec\ A\ (suc\ n)\ . \end{aligned}$$

- While $List$ defines a datatype inductively, Vec inductively defines a *family* of types

  - $Vec\ A\ 0$ is the base case, with only one value $[\,]$.
  - $Vec\ A\ 1$ is defined in terms of $Vec\ A\ 0$, and $Vec\ A\ 2$ is defined in terms of $Vec\ A\ 1 \ldots$

- Agda allows us to reuse the symbols $[\,]$ and $\_::\_$ (it complains in case of ambiguity).

3 Practicals

- As programming with dependent types is best done through conversation with the computer, teaching dependently typed programming is better done through practicals.

- The rest of the lecture proceeds by walking through the practicals accompanying this course.

A Agda Emacs Mode Key Combinations

Global commands

`C-c C-l`	Load a file
`C-c C-x C-c`	Compile a file
`C-c C-x C-q`	Quit
`C-c C-x C-r`	Kill and restart Agda
`C-c C-x C-d`	Remove goals and highlighting (deactivate)
`C-c C-x C-h`	Toggle display of hidden arguments
`C-c C-=`	Show constraints
`C-c C-s`	Solve constraints
`C-c C-?`	Show goals
`C-c C-f`	Next goal (forward)
`C-c C-b`	Previous goal (back)
`C-c C-d`	Infer (deduce) type
`C-u C-c C-d`	Infer type (normalised)
`C-c C-o`	Module contents
`C-c C-n`	Compute normal form
`C-u C-c C-n`	Compute normal form (ignoring abstract)
`C-c C-x M-;`	Comment/uncomment the rest of the buffer

Commands working in the context of a specific goal

`C-c C-SPC`	Give
`C-c C-r`	Refine
`C-c C-a`	Auto (proof search)
`C-c C-c`	Case
`C-c C-t`	Goal type
`C-u C-c C-t`	Goal type (without normalising)
`C-c C-e`	Context (environment)
`C-u C-c C-e`	Context (without normalising)
`C-c C-d`	Infer (deduce) type
`C-u C-c C-d`	Infer type (normalised)
`C-c C-,`	Goal type and context
`C-u C-c C-,`	Goal type and context (without normalising)
`C-c C-.`	Goal type and inferred type
`C-u C-c C-.`	Goal type and inferred type (without normalising)
`C-c C-o`	Module contents
`C-c C-n`	Compute normal form
`C-u C-c C-n`	Compute normal form (ignoring abstract)

Other commands

`TAB`	Indent the current line (cycles between positions)
`S-TAB`	Indent the current line (cycles in the other direction)
`M-.`	Go to the definition of the identifier under point
Middle mouse button	Go to the definition of the identifier clicked on
`M-*`	Go back