CS3110 Spring 2016 Lecture 8: OCaml Type Theory Continued

Topics

1. Review Lecture 7, OCaml type theory
2. Look at typing rules and type checking
3. Defining the `void` type and propositional logic
4. Look closer at logic, we already have &, $\vee$, $\Rightarrow$ $\forall$
5. The challenge of *dependent types*

   $\{x : \alpha | \beta(x)\}$

   $\exists x : \alpha . \beta(x)$

   Specifications with dependent types:

   – GCD

- We started the course with the *computational type system*. That required a precise syntax and computational rules to define canonical and noncanonical *values*.

  We gave a precise semantics for a small subset of the OCaml language based on step-by-step reductions for several expressions. There are many rules and no complete account outside of the compiler.

  We explained the OCaml *type system* for these values. It is "very rich," and we estimated there would be over 200 rules. Today we will look at the *format* of these rules and give *examples*.

  We ended with the comparison to the number of rules to explain modern mathematics:

- First-Order Logic and ZFC set theory (10 axioms)
- $\&, \vee, \Rightarrow, \sim, \forall, \exists$ $\quad (6 * 2 = 12$ axioms)

  What explains this?

- We'll examine some *typing rules* and look at type inference.

  The folk wisdom about OCaml is "*if it type checks it works!*"

  Can the types provide an adequate precise description of programming tasks?

  OCaml, Haskell, $F^{\#}$, etc. *Almost.*

  The type systems of the proof assistants Agda, Coq, and Nuprl can define precisely most (if not all current) programming tasks as well as most modern mathematics.

  How close is the OCaml type theory to the theory used by these proof assistants – called "constructive or computational type theory?"

  Closer than most people think! Just missing bits of logic that come from *dependent types.*

Summary of the OCaml (Functional) Type System

Atomic Types

1. *unit*	5. *string*
2. *bool*	6. *float*
3. *int*	7. *exn*
4. *char*	8. *Big_int*

The exception type, *exn*, is discussed on page 129. We want to include *Big_int* in the standard theory since they are essential to mathematics.

Type Constructors

9. $*$ product type, e.g. $bool * int$. The elements are pairs, and repeated pairing gives tuples, $float * float * float$.

10. $\rightarrow$ function space e.g. $bool \rightarrow int$. Elements can be *anonymous functions* such as $fun\ x \rightarrow exp$, e.g. $fun\ x \rightarrow x$.

11. options (also called variants, p.103), e.g. $L\alpha || R\beta$.

12. lists, e.g. α-list, elements $[e_1; e_2; ...; e_n]$, p.11, Chapter 3.

13. $\{id_1 : ty_1;\ id_2 : ty_2\,; ...;\ id_n : ty_n\}$, records, Ch. 5, p.87.

14. $< id_1 : ty_1;\ id_2 : ty_2\,; ...;\ id_n : ty_n >$, objects, p.212.

15. $[>\ 'id_1\ :\ val_1;\ 'id_2\ :\ val_2; ...;\ 'id_n\ :\ val_n]$, polymorphic variants (we won't use these in lecture).

16. ```
    'a ty = | case 1 of ty
            | case 2 of ty
                 ...
            | case n of ty
    ```
    recursive type, p.111

17. ```
    module type T = sig ... end
    ```
    module signature (i.e. type), Ch. 4

18. ```
    module Name (M:ty_in) : ty_out = structure CODE end
    ```
    functors, p.176.

19. monitors, Ch. 18
    ```

Type Inference Examples

Type of `fun x -> x * x`?

Type of `fun x -> hd x`?

Type of `((fun x -> x * x) -1, -1 < 0)`?

Type of `tl [3;5;7;11]`?

Type of

```
majors = | CS | ECE | Math | Hist
    major_to_start = function
         | CS -> 100
         | ECE -> 90
         | Math -> 200
         | Hist -> 50;;
```

See *Real World OCaml* p.103-105.

Note *union* or *disjoint union* is another name for the following type:
`N1 of ty1 | N2 of ty2 | N3 of ty3`

Consider these two expressions:

```
{item : 'a; time : float}.item
{item : 'a; time : float}.time
```

What do they mean? See page 88 of the book.

Sample Typing Rules

$f \in \alpha \rightarrow \beta,\ a \in \alpha \vdash f\,a \in \beta$

$p \in \alpha * \beta \vdash fst\,p \in \alpha$

$p \in \alpha * \beta \vdash snd\,p \in \beta$

$$\begin{aligned} exp_1 \,\&\&\, exp_2 = true &\vdash exp_1 \in bool \\ &\vdash exp_2 \in bool \end{aligned}$$

$\vdash \{item : {'a}; time : float\}.item \in {'a}$

$\vdash \{item : {'a}; time : float\}.time \in float$

$\ell \in \alpha\, list,\ \ell \neq [\,] \vdash hd\ \ell \in \alpha$

$\ell \in \alpha\, list,\ \ell \neq [\,] \vdash tl\ \ell \in \alpha$

Expressing Logic with Types

Logic is a preferred way to specify tasks.

$$\begin{array}{rcl}
\alpha * \beta & \text{as} & \alpha \,\&\, \beta \\
\alpha \rightarrow \beta & \text{as} & \alpha \Rightarrow \beta \\
L\alpha || R\beta & \text{as} & \alpha \vee \beta \\
\{none : \alpha.\alpha\} & \text{as} & False \text{ (or } Void) \\
\alpha \rightarrow void & \text{as} & \sim \alpha \\
x : type \rightarrow P(x) & \text{as} & \forall x : type.P(x)
\end{array}$$

The harder bit is $\exists x : ty.P(x)$. We will discuss whether this can be done using a first class module in the next lecture. We would like to apply this idea to Euclid's GCD Theorem.

GCD Theorem in Type Theory

$\forall n, m : \mathbb{N}.\exists g : \mathbb{N}.GCD(m; n; g)$

```
GCD(m;n;g) = (g|m) ∧ (g|n) ∧ (∀z:ℤ.(((z|m) ∧ (z|n)) ⇒ (z|g)))
```

For a detailed account of this theorem and the algorithm and implementation in the Nuprl proof solver see `www.nuprl.org/MathLibrary/gcd/`.

These are examples of how to define the void type and the logical operators using modules.

```
# module type Prop = sig type t end;;

module type Prop = sig type t end

# module IMP = functor(A : Prop) -> functor(B : Prop) -> struct
   type t = A.t -> B.t end;;

module IMP:
  functor(A : Prop) -> functor(B : Prop) -> sig type t = A.t -> B.t
end

# type void = {none : 'a.'a};;

type void = {none : 'a.'a;}

# module Void = struct type t = void end;;

module Void : sig type t = void end

# type record_void = {field : record_void};;

type record_void = {field : record_void;}

# module NOT (A:Prop) = IMP (A) (Void);;

module NOT : functor(A : Prop) -> sig type t = A.t -> Void.t end

# module OR = functor(A:Prop) -> functor(B: Prop) -> struct
 type t = L of A.t | R of B.t end;;

module OR:
  functor(A : Prop) ->
     functor(B : Prop) -> sig type t = L of A.t | R of B.t
end
```