CS3110 Spring 2016 Lecture 7: OCaml Type Theory

Summary of OCaml Types

We will list the atomic types and the type constructors for the purely functional core of the OCaml Type Theory. This includes eight atomic types and eleven type constructors plus the void type for a total of 20 types. For each type there are several rules, some about typing the canonical values and some about typing the operators on these values.

The number of rules varies from two to perhaps 20, so this is by no means a small theory. It expresses precisely many concepts from CS, mathematics, science, and engineering.

If we compare the "weight" of this theory to that of the standard theories of mathematics, it is revealing. The minimal foundation of 20th century mathematics can be given with ten logical rules and and ten set theory rules (for ZFC). The OCaml Type Theory will have over 200 rules. What is going on?

We have now used the OCaml programming language enough to appreciate that it is not only a useful and expressive programming language, but it is based on a rich *implemented theory of types*. This theory includes rules for evaluating expressions that we have studied, the *small step operational semantics*.

In this week's lectures we will present the elements of the *OCaml type theory*. The type theory is governed by rules for defining types and for typing syntactically correct meaningful expressions of the language. Explaining the type theory anticipates the direction in which a course like this one will be taught when it is supported by a *proof assistant*. This is a direction in which the US will invest, following examples explored widely in the European Union.

First we focus on examples of typing rules for List Theory, a subset of the broader type theory. We will see a *pattern to the rules.* Then we will enumerate the OCaml types and estimate the *formal content* of the theory based on the number of typing rules and their interaction with the computation rules.

In Lecture 8 we will see that OCaml can express a significant number of *logical propositions as types* and provide precise rules for logical reasoning in the type theory.

To grasp how the type theory rules work, we look at some of the rules for lists. Recall that we examined the form of the "one-step" computation rules previously. (Real World OCaml, Ch. 1 & 3.)

Consider *polymorphic* lists, $'a$ list. We also write them in lecture using Greek letters, α-list. These lists are representative of the unbounded variety of typed lists such as *int* list, *bool* list, *unit* list, *char* list, etc.

Using variants we also have hybrids such as

$$(Int \text{ of } int \mid\mid Flt \text{ of } float) \text{ list}$$

A canonical list is usually given as $[e_1; e_2; ...; e_n]$, where e_i are elements of the right type.

The canonical *empty list* is [ ].

A non-empty list is built using the "cons" operator. $e :: \ell$.

The list $e_1 :: (e_2 :: (e_3 :: [\ ]))$ is displayed $[e_1; e_2; e_3]$.

The list operators include these primitives:

cons is	::	
head is	*hd*	$hd\ a :: [\ ] \downarrow a$
tail is	*tl*	$tl\ a :: [\ ] \downarrow [\ ]$

Summary of the OCaml (Functional) Type System

Atomic Types

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