CS3110 Spring 2016 Lecture 4: OCaml Type System Continued – Partial types

Robert Constable

1 Lecture Plan

1. Response to Piazza questions and comments.
2. Schedule of problem sets (6 of them) and prelim.
3. Enrichment topics ("beyond what Google knows"), e.g. *partial types* tie to computability theory and mathematics (sets versus types).
4. What is a type? (sets vs types) more enrichment.
5. Thoughts about problem set 1:
   - Defining the rational numbers (the issue is $n/0$ and using the gcd algorithm)
   - PS2 will look at using modules for rationals $\mathbb{Q}$ and extending to reals $\mathbb{R}$ in PS3.
   - The gcd algorithm allows removing common factors in numerator and denominator.
6. Bits of logic among the OCaml types, polymorphic types:

   $\alpha \,\&\, \beta \qquad \alpha \Rightarrow \beta \qquad \alpha \vee \beta \qquad False\,? \qquad True\,?$

2 Piazza questions and comments

What resources do we have for learning OCaml?

- Textbook: *Real World OCaml* (we use about 1/2)

  Please read all of Chapter 1 ('assigned' before), Chapter 2 (will be used in Lecture 5), and Chapter 3 (List basics) in Lecture 6..

- See *past CS3110 lecture* notes, Nate Foster builds on Dexter Kozen.
- Other resources listed on course site, e.g. we already used small step semantics and posted a resource.
- Canonical values under evaluation – discussed in Lectures 2 and 3.

  The key point is that they are defined by *eager evaluation*, the pair $(2*3,\ 5/5)$ is <u>not canonical</u>, reduces to (6,1), likewise $[2*3;\ 5/5;\ 0*7]$.

  What does $[fun\ x \rightarrow x;\ \ fun\ x \rightarrow 1]$ show when evaluated?

3 Schedule of problem sets and in-class prelim

	Date for	Due Date
PS2	Out on Wed. Feb. 17	March 3
PS3	Out on Fri. March 4	March 24
Prelim	Thur. March 17, in class	
PS4	Out on Thur. March 24	April 8
PS5	Out on Fri. April 8	April 28
PS6	Out on Thur. April 28	May 11 (last day of classes)

4 Enrichment topics ("Beyond what Google knows")

The lectures also discuss ideas that are not directly related to how to use OCaml. We have made comparisons between OCaml and other languages such as Lisp and Haskell. These comparisons show you the wider context of functional programming and where important languages such as Haskell and Lisp differ from OCaml. We have mentioned the proof assistant Coq which is used to write *specifications* of programming tasks and to show that OCaml programs satisfy these specifications. The Coq proof assistant uses OCaml as its programming language.

In due course we will talk more generally about what types are. We already mentioned that types in OCaml are actually *partial types* in the sense that OCaml allows some expressions that may not terminate to be members of essentially all of its types. Thus the type of integers, `int` includes not only integers, but also expressions that "would be integers" if they terminated, but we do not yet know whether or not they terminate.

The topic of partial types is important to a thorough understanding of modern programming and to the notion of correctness. We will discuss several topics in this category to give you a better idea of the directions in which programming languages are evolving and in which our understanding of computation is deepening. So it will be important for this course to know why `int` is a partial type. Suppose we have the boolean valued functions `gr` to test whether an integer is greater than 0 and `le` to test whether less than 0.

```
let rec loop n:int :int if x = 0 then 0
else if gr(x) then loop(x+1) else loop(n-1).
```

We say that this diverges on any non-zero input. So while `loop(0),loop(1)` both have type *int*, only one expression is actually an integer, namely 0. The other expression does not converge to an integer. We say it *diverges.*

We sometimes write $\perp$ for a generic expression that has no canonical value because it "diverges," i.e. its computation does not terminate. We can even find $\perp$ in *bool* and *unit*! Why?

5 What is a type?

For computer science, types are a fundamental concept, analogous to the concept of sets in mathematics, but different in very important ways that you need to understand. It is important that in this course you know the difference between sets and types. You are probably much more familiar with sets because they are used in mathematics right from the start. There are two ways that sets are discussed: intuitive and axiomatic.

Intuitive (informal): the empty set is a set and any collection of sets with no repetitions is a set, e.g

$$\{\emptyset\},\ \{\{\emptyset\},\{\emptyset,\{\emptyset\}\},....\}$$

.

Axiomatic: Zermelo-Fraenkel with Choice (ZFC)

A0. there is a set	A5. union
A1. equality	A6. replacement
A2. foundation	A7. infinity
A3. comprehension $\{x\!:\!A \mid P(x)\}$	A8. power set
A4. pairing	A9. choice

Plus the axioms of First-Order Logic (&, $\vee$, $\Rightarrow$, $\sim$, $\forall$, $\exists$), a dozen rules plus 10 axioms.

Compare to OCaml type theory:

> *unit, int, bool, char, string, exn*
> $\alpha * \beta \quad \alpha \rightarrow \beta \quad L\alpha \mid\mid R\beta$
> records, variants
> lists, recursive types
> asyn-package
> (refs)

45 or so rules plus computation rules.

Types are based on a computation system defined on untyped expressions. OCaml uses small step evaluation semantics.

Types are a collection of *canonical values* from the computation system with a notion of equality on them. As canonical values, the expressions simply stand on their own. The meaning of the expressions arises from relating the canonical and non-canonical values. For example, consider the list `[1;2;3]`. The relationship between the constructors and the destructors starts to reveal their "meaning." For example, if we say `hd [1;2;3] = 1` then we see the meaning of the *hd* operation. We start to "understand" what a list is by applying operations to build them and others to take them apart. We can see this relationship at the top level of the evaluator, but how do we express it in the language of the OCaml type theory itself? We use equations. We will see that types can be understood as *partial equivalence relations on expressions*. From this definition, we already see that before we can understand an OCaml type, we need to know the computation rules relevant to it. That is why we started by discussing evaluation and canonical values. Without those ideas, we cannot understand types in computer science. In contrast, to understand sets in mathematics, we never need to mention computation.