CS3110 Spring 2017 Lecture 4: More OCaml Types

Robert Constable

1 Lecture Plan

1. Schedule of problem sets (6 of them) and prelim.
2. Polymorphic types continued.
3. Recursive types and definition of OCaml Lists.

2 Schedule of problem sets and in-class prelim

	Date for	Due Date
PS1	Out on Tue. Feb 7	Feb 14
PS2	Out on Feb 23	March 2
Prelim	Tue. March 14, in class	
PS3	Out on Thur. March 2	March 16
PS4	Out on March 16	March 30
PS5	Out on April 10	April 24
PS6	Out on April 24	May 8 (day of last lecture)

3 Overview – Types in Mathematics and Programming

The lectures will cover many concepts that are not directly related to using OCaml but are nevertheless enduring ideas in the development and understanding of programming languages, programming methodology, and the mathematical foundations of computer science. The practical need to program well has driven the intellectual development of computer science

right from the very beginning with John McCarthy's sketch of a mathematical basis for computing [2]. Cornell has a reputation for discovering some of these mathematical foundations and producing several of the leading researchers and CS professors in this area. Several are mentioned in the collection of references that come with the lectures. The textbook by Simon Thompson, *Type Theory and Functional Programming* [5], is now available in a free pdf. It provides an excellent account of the role of types in programming languages and cites many of the leading researchers. Simon was working on a revision of the book recently when he decided instead to make the first edition freely available. Here is the url for a download.

`https://www.cs.kent.ac.uk/people/staff/sjt/TTFP/ttfp.pdf`

We have already made superficial comparisons between OCaml and other languages such as Lisp, SML, Haskell and the programming language of the *Coq proof assistant*, called CoqPL in these lectures. These comparisons show you the wider context of functional programming and how important languages such as Lisp, SML, Haskell, and CoqPL differ from OCaml. There are other languages like this "in the pipeline" from industry and other universities. We have mentioned the *proof assistant Coq* which is used to write *specifications* of programming tasks and to show that CoqPL programs satisfy these specifications. CoqPL is similar to OCaml, and it is possible to program with it. The CoqPL types are all *total types*. The Coq research group and the OCaml group work together at INRIA, the French government research facility. They were collected into a single group by a senior INRIA manager, Gilles Kahn, who was also an excellent computer scientist who developed what is called "big step" structured operational semantics [1]. Here is a url for obtaining Coq.

`https://coq.inria.fr/`

In due course we will talk more generally about what the nature of types and the development of a rigorous mathematical type theory. 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 they terminate. Moreover, OCaml has a vast number of types that we will not study, several related to the idea of

objects, the O part of OCaml. For example all of these concepts are implemented among the many OCaml types: objects, records, modules, first-class modules, functors, and classes. All of them are ways of grouping types and operations together. We will only study a couple of them in this course. We will study records and modules but not objects or classes.

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 program 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 gave a different but related example in Lecture 3.

As already mentioned, we 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?

3.1 Polymorphic Types

We have already seen that polymorphic types allow us to express simple programming tasks. We noticed that the specifications look close to logical expressions. We saw examples like these. Recall that we mentioned that there is an OCaml type that is empty. We abbreviate it as *void*. It can play the role of $False$ in logic just as the unit type can play the role of $True$. Can we make sense of this type as an *implication*? $False \rightarrow True$?

unit, int, bool, char, string, exn
$\alpha \rightarrow \alpha$.
$void \rightarrow void$.
$(\alpha * \beta) \rightarrow \alpha$.
$(\alpha * \beta) \rightarrow L\alpha | R\beta$. [1]

[1] Sometimes we will abbreviate this verbose expression as $(\alpha \mid \beta)$. To suggest the rela-

$(\alpha * \beta) \to \beta * \alpha$.
$((L\alpha|R\beta) \to \gamma) \to L(\alpha \to \gamma)|R(\beta \to \gamma)$.
$((\alpha \to \beta) * (\beta \to \gamma)) \to (\alpha \to \gamma)$.

What about the type

$$(R(\alpha \to \gamma)|(L(\beta \to \gamma)) \to (R\alpha|L\beta) \to \gamma).$$

Is there a program in this type? When seen as a logical expression, is it true?

3.2 Currying and Uncurrying

Here is an interesting functional programming construct that goes all the way back to Haskell B. Curry. For Curry this was a fact of logic as well. Let's see how we can view it both ways.

The following type describes an operation on programs called Currying. You can see the type in the response of the OCaml evaluator.

```
# let curry = (fun h ->(fun x -> (fun y -> h (x,y)) ) ) ;;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# let uncurry = (fun f -> (fun p -> (f (fst p) (snd p) ))) ;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>

# (curry (fun x -> ( fst(x) + snd(x) )) )  ;;
- : int -> int -> int = <fun>

# (curry (fun x -> ( fst(x) + snd(x) )) ) 5 7 ;;
- : int = 12

# (uncurry (curry (fun x -> ( fst(x) + snd(x) )) )) ;;
- : int * int -> int = <fun>
# (curry (uncurry (fun x -> (fun y -> x + y))) ) ;;
- : int -> int -> int = <fun>

# (curry (uncurry (fun x -> (fun y -> x + y))) ) 5 7 ;;
```

tionship to the Boolean "or" it would also make sense to write $\alpha||\beta$ when it is clear that we are talking about types.

```
- : int = 12

# curry (fun h -> 0) ;;
- : '_a -> '_b -> int = <fun>
# curry (fun h -> 0) 5 7 ;;
- : int = 0

# uncurry (fun x -> fun y -> 0) ;;
- : '_a * '_b -> int = <fun>
# uncurry (fun x -> fun y -> 0) (5,7) ;;
- : int = 0
```

3.3 Functions that form a basis for all Church/Turing computable functions – optional reading

In some of the literature on programming language semantics, various functions are given special attention. For example, the identity function, $fun\ x \rightarrow x$ is written as a bold capital i, **I**. The function $fun\ x \rightarrow (fun\ y \rightarrow x)$ is called **K** for "constant function" in German. The **S** combinator satisfies **S** x y z = (xz)(yz). We see the OCaml definition below. These symbols **I**, **K**, and **S** are called *combinators*.

The logician Haskell B. Curry used these instead of Church's lambda calculus to write functions. Another famous logician, who was once in the Cornell Mathematics Department, J. Barkley Rosser, found a very interesting combinator that he called **X**. It is this particular OCaml function,

$$fun\ f \rightarrow f\ S(fun\ x \rightarrow fun\ y \rightarrow fun\ z \rightarrow x).$$

where S is $fun\ f \rightarrow (fun\ g \rightarrow (fun\ x \rightarrow f\ x\ (g\ x)))$.

Here is a *cultural enrichment* nugget. You are NOT responsible for this, but you can use it to amaze your friends in computing theory courses. Most of them don't know it, never even heard of it. The amazing thing about **X** is that *it is the only computable function we need.* All other computable functions can be built up from **X**. So all programs can be written as a sequence of this **X** combinator and parentheses, e.g. **XX**(**X**)(**XXX**)(**X**)(**XX**)(**XX**(**XX**)). Here we clearly see the difference between theoretical possibility and practical reality. We will not study this kind of fact about functions in this course. It is a topic that is sometimes

covered in CS4110, the undergraduate programming language course. The graduate course, CS6110 often covers the idea of combinators.

4 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 a small core OCaml type theory:

unit, int, bool, char, string, exn
$\alpha * \beta \quad \alpha \to \beta \quad L\alpha \mid R\beta$
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. This approach is due to

another Edinburgh University professor, Gordon Plotkin [4]. Gordon was one of the computer scientists who saw that the Logic of Computable Functions (LCF) was also a nice programming language [3].

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.

References

[1] G. Kahn. Natural semantics. In G. Vidal-Naquet F. Brandenburg and M. Wirsing, editors, *STACS '87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, 1987.

[2] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[3] Gordon D. Plotkin. LCF considered as a programming language. *Journal of Theoretical Computer Science*, 5:223–255, 1977.

[4] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Department, Denmark, 1981.

[5] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.