CS3110 Fall 2013 Lecture 15: Specification Languages (10/22)

Robert Constable

1 Lecture Plan

1. The Role of Proof Assistants in Programming
2. Mathematical types and specifications
3. Dependent types
4. Knowing that programs meet specifications

2 Review, Overview, and Comments

In Dr. Vincent Rahli's Lecture 14 demonstration of Nuprl we learned that four modern *proof assistants*, Agda, Coq, Nuprl, and MetaPRL, *specify* programming tasks using a *mathematical type theory*. In this lecture we will see that this theory significantly extends the computation theory of OCaml. The mathematical theory of these proof assistants is based on total types with equality and includes dependent types, such as *subset types* of the form $\{z : \mathbb{Z} \mid z > 0\}$. This particular *dependent type* was mentioned when we defined the rational numbers.

The type theory implemented by Nuprl and MetaPRL includes partial types

as well as *total types*.[1] Coq and Agda use only total types. Total types are necessary for mathematics and for specifying computational problems. **Expressions which have total types are guaranteed to compute to canonical forms using the evaluation rules of the theory. By contrast, OCaml's partial types offer no such guarantee.** Even the OCaml unit type, with `()` as its only canonical value, does not guarantee that every expression with type unit will converge to `()`. Here is an example of an expression of type unit that does not converge to `()`.

```
# let rec unt (x : unit) : unit = if x = () then unt x else () ;;
val unt : unit -> unit = <fun>
```

Because OCaml has only partial types, and partial types include diverging elements, this expression type checks in OCaml as `unit -> unit = <fun>`. If OCaml had only total types, then this expression would not type check.

We can create expressions of any OCaml type which do not converge to the canonical values of that type.

In mathematical type theory, the total types come first. In the Nuprl type theory for example, the partial type of integers is denoted $\overline{\mathbb{Z}}$, and its definition presumes the total type. In OCaml, there are only partial types, and although we normally focus on the canonical elements and explain them first, there is no way to express that we require the canonical elements.

Since OCaml has only partial types, it is not adequate for using types to exactly capture mathematical ideas. This limits the role of OCaml types for specification and disconnects its types from standard mathematics. OCaml is only a "partial specification language." Computer science, mathematics, and industry need "more complete specification languages," and that is one reason that proof assistants were invented.[2]

[1] The terminology for these types is not standard. They might also be called *value types* or *mathematical types* or something else.

[2] Computer algebra systems such as *Mathematica* and *Maple* are more like proof assistants, and there are research projects that combine them into one system [5, 2].

In the next two lectures we will examine mathematical type theory as a specification language for computing tasks. We need to both extend OCaml types and restrict them. We extend to include dependent types, and we restrict to identify total types. We also need to be clear about equality relations on the types. Let us summarize the additions and restrictions on OCaml types required to have a specification language.

1. Assume all OCaml types are total.
2. Define *equality* on each type.
3. Add *dependent* functions, products, and records.
4. Relate the new types to logical language used in mathematics.

In the rest of the course we will use the name *OCaml Specification Language* or *Extended OCaml* for the extended type theory of OCaml that we define in the next two lectures. This will be our specification language. It will share features with Agda,Coq, Nuprl, and MetaPRL.

3 Reading and Sources

Logic and verification were covered in lectures 14,15,16, and 28 of CS3110 in 2013sp. The material for this lecture is related to those lectures. On the other hand, here we present logic and verification as it is implemented and used in modern proof assistants. More material can be found in on-line booklets from http://www.nuprl.org/MathLibrary/ , the Mathematics Library of the Nuprl proof assistant.

The on-line book *Software Foundations* [6] presents material formalized using the Coq proof assistant which is closely related to Nuprl. Chlipala [3] provides more information about Coq.

4 Specification languages

Starting with Euclid's *Elements* the language of mathematics has focused on how to *specify problems* or tasks and demonstrate that a task can be solved. In geometry the task is often to build a geometric object such as an equilateral triangle. In algebra the task might be to solve an equation for certain variables.

Since computers can be used to perform an incredibly vast number of tasks and solve a vast number of mathematical and nonmathematical problems, the challenge of finding a language for precisely stating computational tasks or problems is one of the most daunting intellectual challenges. For some of these problems there is a two thousand year history and for others there is no history.

Consider the induction problem on Prelim one, it is a problem we are all familiar with. We could have posed it like this.

Given a list lst of type $\alpha\ list$ and an element x of type α, *find and collect* all of the positions (or locations) i such that $nth\ lst\ i\ =\ x$. In this informal language we might ask, how do we collect these numbers? In a list? In a set? In a tree? In a binary search tree? In a red/black tree? Will all the elements of the list be of the same type? This last question is answered by knowing that all OCaml lists are homogeneous and we gave the type as $\alpha\ list$. We can make the specification more precise using symbolic language as we did on the Prelim.

You were required to understand the following quantified symbolic statement.[3]

[3] On the exam we gave a weaker specification, $\forall lst : \alpha\ list.\ \forall\ x\ : \alpha.\exists\ indxs\ : int\ list.\forall i : int.(i\ on\ indxs \Rightarrow\ nth\ lst\ i\ =\ x)$ that seemed to allow for a trivial program that produced the empty list as $indxs$. By the logic taught in some logic books we might argue that if we produce the empty list, [], as the answer, then to assume that i is on [], we are making a false assumption, therefore by the "rules of logic" anything follows. We disallowed that trivial solution as a side condition, but some proof assistants such as Minlog [1] do not allow this conclusion because it uses a logic called *minimal logic*. Here we give a specification that does not admit a trivial solution.

$$\forall lst : \alpha\ list.\ \forall\ x\ : \alpha.\exists\ indxs\ : int\ list.\forall i : int.(i\ on\ indxs \Leftrightarrow\ nth\ lst\ i\ =\ x).$$

In English this reads: For all *lst* an α list and for all x of type α, we can *construct* an integer list, *indxs*, such that i is on *indxs* if and only if the i-th element of *lst* is x.

One way to prove our claim about such lists is to write a program that builds the list of positions of *lst* at which the element x of type α occurs. A program to do this will need to check for equality on the type α. This means that we should in fact also assume that the type α has such a test, i.e. that equality on α is *decidable*.[4]

So with this understanding, we can imagine a clearer symbolic expression such as this one:

$$\exists\ positions\ :\ \alpha\ list \to \alpha\ \to int\ list.\ \forall\ lst : \alpha\ list, x : \alpha.$$
$$\forall i : int.(i\ on\ (positions\ lst\ x) \Leftrightarrow\ nth\ lst\ i\ =_\alpha\ x).$$

This specification begins to look like a programming problem, and it would look even more this way if we just wrote:

$$Create\ positions\ :\ \alpha\ list \to \alpha\ \to int\ list.$$
$$\forall\ lst : \alpha\ list, x : \alpha.\ \forall i : int.(i\ on\ (positions\ lst\ x) \Leftrightarrow\ nth\ lst\ i\ =_\alpha\ x).$$

Exercise: We can take another step closer to a purely computational explanation of the problem if we replaced the $\Leftrightarrow$ by something more intuitive from a computational point of view. What is the meaning of $\Leftrightarrow$, also read as *if and only if* or just *iff*? Can we give a computational explanation of that meaning? Might that explanation tell us why Minlog does not accept the trivial solution to this specification?

[4]We did not ask you to comment on the equality relation, but we would have been impressed if you had. For example, if α is the type $int \to int$ or the type of reals, $\mathbb{R}$, then there will be no equality test, so our idea for the algorithm and for the proof would fail. So this proposition is in fact true only for the special class of types α in which equality is decidable. As Dr. Rahli mentioned, in Standard ML (SML) these types are denoted *''a*.

Related to this discussion is another interesting result of logic that we will examine computationally. In many accounts of logic, if we have proved $\forall x : \alpha . \exists y : \beta . R(x, y)$, then we can also prove $\exists\ f : \alpha\ \rightarrow\ \beta . \forall\ x : \alpha . R(x, f(x))$.

The prevailing view on specification languages in the academic programming language community is that *some version of type theory is the richest specification language available.* We have seen OCaml's type theory in action now for eight weeks. We have used OCaml types as specifications and have learned to provide types for inputs as well as types of solutions. We have tried to code into the types the assumptions on inputs to a problem, the invariants that we expect a data structure to maintain, the requirements on the output and so forth.

We will follow the steps in the overview section to define our specification language. The first step is to define the total types, then we discuss equality, and finally dependent types.

4.1 Total types

Many times we have mentioned mathematical types such as the Booleans, $\mathbb{B}$, the natural numbers, $\mathbb{N}$, the integers $\mathbb{Z}$, the rationals $\mathbb{Q}$, and the reals $\mathbb{R}$. We will now have these as OCaml specification types, under their OCaml names, bool, int, rationals, reals.

4.2 Equality on a type

All mathematical types come with a notion of equality. In a sense it is the notion of equality that characterizes mathematical objects. Given a mathematical type T, the relation of equality on T is written either as $x =_T y$ or as $x = y\ in\ T$. When we defined the rational numbers $\mathbb{Q}$ and the reals, $\mathbb{R}$, the definition of equality was interesting. For example, if we had defined equality of rational numbers (a, b) and (c, d) as $a =_{\mathbb{Z}} c$ and $b =_{\mathbb{Z}} d$, this would not match mathematical practice and would signal that we were

actually talking about *fractions*.

OCaml seems to provide equality on any type. For example, we can define this equality operator on 'a.

```
# let eq (x : 'a * 'a) : bool = let (p,q) = x in p = q;;
val eq : 'a * 'a -> bool = <fun>
# eq (3,4) ;;
- : bool = false
# eq ([],[]) ;;
- : bool = true
```

But we see this behavior as well.

```
# eq ( (fun x -> x),(fun y -> y));;
Exception: Invalid_argument "equal: functional value".
```

The equality relation on a type, $x =_T y$ is called *decidable* if and only if we can construct a function $eq_T : T \to T \to bool$ such that
$(eq_T x, y = true) \Leftrightarrow (x =_T y)$.

There are two standard notions of equality on total function types such as int $\to$ int. One of them is what you see in mathematics text books, called extensional equality defined this way.

Extensional function equality: Given f,g of type A $\Rightarrow$ B, we say f $=_{A \Rightarrow B}$ g if we know f(x) $=_B$ g(x) for all x of type A.

4.3 Dependent types

The expressive power of type theory used in mathematics and computing theory comes from *dependent types*. The simplest dependent type is a generalization of the product type, $\alpha \star \beta$, in OCaml. In mathematics this is the *Cartesian product* of types α and β, and it consists of the ordered pairs (a, b) for $a \epsilon \alpha$ and $b \epsilon \beta$. The dependent product is written $x : \alpha * \beta(x)$. It consists of the ordered pairs (a, b) such that $a \epsilon \alpha$ and $b \epsilon \beta(a)$.

The expression $\beta(x)$ is a type parameterized by the type α. We express this precisely by saying $\beta : \alpha \rightarrow$ Type. This requires that we consider Type itself as something we call a *large type*. It will be the only large type we need for now. We say more about Type later.

A closely related type is the *subset type* or *refinement type* which in OCaml might be written as (x:α where β(x)) where β is a bool valued total function on α. The elements of this type are all the elements of type α for which the function value β(x) is true. The mathematical type is $\{x : A|B(x)\}$ which consists of the elements a of type A for which there is an element b in type $B(a)$. We do not include the element b in the type, but we must have created such a value before we know that a is in the type.

The *dependent function* type is the other critical type for specifications. It generalizes the OCaml type $\alpha \rightarrow \beta$. In OCaml it is written as x:$\alpha \rightarrow \beta$(x). It consists of the computable total functions f such that for any a in α, we know that $f(a)$ is of type $\beta(a)$.

Here is a very simple and clear use of these dependent types to define the date for a non leap year. Let the type $Month$ be the variant type Jan — Feb — March —...— Dec and let the type Num_days be a function from $Month$ to Int defined as follows.

```
let Num_days (m:Month) : Int = match m with |Jan -> 31 | Feb -> 28
|March -> 31 | April -> 30 | May -> 31 | June -> 30 |July -> 31 |
Aug -> 31 | Sept -> 30 | Oct -> 31 | Nov -> 30 | Dec -> 31
```

Now we can define the date for a typical year as this dependent product using subset types.

$Date = m : Month \times (n : int\ where\ 1 \leq n \leq Num\_days(m))$

Exercise: Define this type with y:Year as the first component and is correct for the century (or for the millennium if you can).

4.4 Large types

For dependent types we need the idea of a type whose elements are types. We call these large types. Right now we only need one of them, Type. In general we will have Type_1, Type_2, etc.

We note that the atomic types unit, bool, int, the variants like Month belong to Type. Also, if $\alpha, \beta \epsilon$ Type, then so are $\alpha * \beta, \alpha \to \beta$, the recursive types, and the new dependent types all belong to Type.

References

[1] H. Benl, U. Berger, H. Schwichtenberg, et al. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. G. Schmitt, editors, *Automated Deduction*, volume II. Kluwer, 1998.

[2] Bruno Buchberger. Theory exploration versus theorem proving. In Alessandro Armando and Tudor Jebelean, editors, *Proceedings of Calculemus 99, a satellite workshop of CADE 16 at FLoC'99*, volume 23/3 of *Electronic Notes in Theoretical Computer Science*, Trento, Italy, July 1999. Elsevier Science Publishers.

[3] Adam Chlipala. An introduction to programming and proving with dependent types in Coq. *Journal of Formalized Reasoning (JFR)*, 3(2):1–93, 2010.

[4] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In D. Gries, editor, *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 183–193. IEEE Computer Society Press, June 1987.

[5] Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, Ithaca, NY, January 1995.

[6] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjberg, and Brent Yorgey. *Software Foundations.* Electronic, 2013.

[7] D. Scott. Outline of a mathematical theory of computation. In *Proceedings 4th Annual Princeton Conference on Information Sciences & Systems*, pages 169–176, Princeton, NJ, 1970.

[8] D. Scott. Data types as lattices. *SIAM J. Comput.*, 5:522–87, 1976.

[9] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings Symposium on Computers and Automata*, pages 19–46. Polytechnic Inst. of Brooklyn Press, New York, 1971.