CS3110 Fall 2013 Lecture 16: Specifications and dependent types (10/24)

Robert Constable

1 Lecture Plan

1. OCaml as a specification language
2. Dependent type examples
3. Specifications with dependent types
4. Types look a lot like formulas in logic

2 Review, Overview, and Comments

In Lecture 15 we examined specifications of computational problems using mathematics, going back as far as Euclid who explicitly posed his theorems as geometric problems.

We also examined in more detail the specification given on Prelim 1 for the task of listing all the positions in a given list at which a specific element occurs. We showed how to gradually transform a natural language specification into symbolic language and then into logic and then into types. We will now examine the specification using dependent types. This will be a guide to how to express many propositions as types.

The main work done in Lecture 15 was to define *Extended OCaml* or the *OCaml Specification Language*, an implementable extension of OCaml that can be used as a specification language and logic. This work shows how close OCaml is to mathematical type theory.

We also noted that there can be no perfect type checker for Extended OCaml. That is because given the typing $a : \alpha$ the checker needs to know that the expression a converges. If that expression arises from applying a function f known to be of type $int \rightarrow \alpha$ and we apply f to 0, then $f(0)$ should have type α, but if f diverges on 0, the result will not be a canonical element of type α. The only way to know that is to solve an instance of the halting problem – and we might be able to do this by hand or in many special cases, but we cannot do it for all expressions in Extended OCaml. With a nod to Mick Jagger and Keith Richards we noted that "You can't always get what you want. But if you try, sometimes, well you just might find you get what you need."

So the proof assistant Coq can't solve the halting problem either, but it makes a good try at it, and we have a very widely used proof assistant that helps us program with dependent types and specify computing tasks with dependent types. *Experience with Coq in both mathematics and programming is that very often, we do get just what we need.*

3 Reading and Sources

As mentioned in Lecture 15, logic and verification were covered in lectures 14,15,16, and 28 of CS3110 in 2013sp. The material for this lecture is also related to those lectures as we will see in Lecture 18.

4 Specifications in Extended OCaml and Logic

We are accustomed to seeing mathematical specifications in the form of equations such as

$$e \;=\; \sum_{i=1}^{\infty} \frac{1}{i!}$$

or

$$e \;=\; lim_{n\to\infty} \left[\frac{(n+1)^{n+1}}{n^n} - \frac{n^n}{(n-1)^{n-1}} \right]$$

or

$$e = lim_{n\to\infty}(1 + \frac{1}{n})^n.$$

4.1 Symbolic language for specification

The language of equations is adequate for many mathematical specifications. Those specifications require that we precisely understand equality on types. However, many general and natural programming specifications are written as types or in symbolic language closer to natural language in which certain common expressions such as the *universal quantifier* "*for all x of type t1*" and the *existential quantifier*, "*we can construct an object y of type t2*" can be abbreviated. We also want to symbolically abbreviate the common natural language *connectives* such as *and*, *or*, *not*, *implies*, *if and only if*, say by $\&, \vee, \neg$ (*or* $\sim$), $\Rightarrow, \Leftrightarrow$. We have used these notations in previous lectures in an informal manner.

We have already been using the phrase $\forall\ x : T_1$ to abbreviate the universal statement, and $\exists\ y : T_2$ to abbreviate the *existential statement* (i.e. the *we can find* statement), and we have been writing with the logical connectives as well.

The formal mathematical type theories implemented by proof assistants all use this kind of symbolic language of connectives and quantifiers.

On the first prelim you were required to understand the following quantified symbolic statement.[1]

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

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 int list $indxs$ 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*.

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 $''a$.

For this discussion, we assume that α is one of those types with *decidable equality*. We could express this in mathematical type theory as:

$$\exists eq : \alpha \star \alpha \to bool.\forall\ x, y : \alpha.(eq\ (x, y) = true \Leftrightarrow x =_\alpha y).$$

Note, we need to know here that *bool* is the mathematical type of Boolean values, not the OCaml partial type *bool*. We know this because *this is a logical specification not an OCaml type.*

[1] 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 allowed for a trivial program that produced the empty list as $indxs$. We disallowed that trivial solution as a side condition. Here we give a specification that does not admit a trivial solution.

4.2 Dependent types for specification

The prelim problem

We can write the specification of the prelim problem using dependent types. It looks like this.

$$lst : \alpha\ list\ \rightarrow\ x : \alpha\ \rightarrow\ indxs : int\ list.$$

$$i : int\ \rightarrow (j : int\ where\ nth\ indxs\ j\ =\ i)\ \leftrightarrow\ nth\ lst\ i\ =_{\alpha}\ x.$$

We can rewrite the $\leftrightarrow$ to say that there are two functions, one from $(j : int\ where\ nth\ indxs\ j\ =\ i)$ to $nth\ lst\ i\ =_{\alpha}\ x$ and another from $nth\ lst\ i\ =_{\alpha}\ x$ to $(j : int\ where\ nth\ indxs\ j\ =\ i)$.

It would be good enough to define the recursive function $is\_on(l : int\ list)(x : int) : bool$ and then say

$$(i : nat\ where\ 0 \leq\ i\ < len(lst))\ \rightarrow (is\_on\ indxs\ i\ ) =_{bool}\ (nth\ lst\ i\ =_{\alpha}\ x).$$

This would be a finite check that Boolean values are equal. The ultimate evidence is then just checking the equality of Boolean values. Here are some almost philosophical thoughts about this specification.

Ground evidence: What is apparent from this simple example is that type checking is providing us with the most basic kind of evidence for our knowledge about this programming task and correct solution of it. In the end, all of the *atomic evidence* for this problem solution is of the form $exp_1 =_{bool} exp_2$ in the OCaml Specification Language type bool, *but not* in the OCaml Programming Language type bool.

How do we say that we know $exp_1\ =_{bool}\ exp_2$? We say that the expression evaluates to *true*. The evaluation system tells us this if we are looking at a specific concrete instance. But how do we say this in the specification?

How do we say "we know that this will evaluate to the value *true*"? Do we try to say $(exp_1 =_{bool} exp_2) =_{bool} true$? This is just another specification. How do we know it returns the value *true* without running it? Why don't we need to run $((exp_1 =_{bool} exp_2) =_{bool} true) =_{bool} true$?

We need something that the type checker can report to guarantee that when we execute any specific example, the result will be the boolean value *true*, and we will see that output. One way to do this is to consider $exp_1 =_{bool} exp_2$ as a type, and to say as an *axiom of evidence* that $true =_{bool} true$ has some kind of atomic ground evidence as a value that we put in the type to indicate that this is the irreducible evidence for knowing a primitive fact of mathematics. We don't have to run $true =_{bool} true$, we just know it by saying that this *identity* is an atomic type with atomic evidence $\star$. As humans, we know that we can check simple enough identities by inspection of the symbols. We can also trust a program that simply checks symbolic identity of expressions after reduction to symbolic canonical form. In the end, we can reduce this to checking *true* is identical to *true*.

Likewise to say that $0 =_{int} 0$ is known to us, we can supply atomic evidence of this type or reduce this to a boolean check. We could put $\star$ or *axiom* or *evatom* in these identity expressions regarded as types.

There is a conceptual advantage to using the same bit of atomic evidence for all statements of canonical value equality between identical elements of a type. Doing this provides the ultimate in computational evidence for building knowledge based on concrete symbolic evidence. To evaluate this belief further would take us into philosophy where these ideas have been discussed in one form or another since at least 1907. But the power of this idea is that you can judge it for yourself.

Existence and construction in programming.Notice that an important part of this specification is that *we can actually build the list indxs*. This is what we usually mean in computer science specification languages when we say that something "exists." So although we often use the standard existential quantifier notation, we also agree that it has the "we can construct" meaning. Such a meaning is not necessarily the one you will find in most logic books which are written to cover "non computational

mathematics" as well as computational mathematics. *But it is the meaning adopted by all four main proof assistants for programming: Agda, Coq, Nuprl, and MetaPRL.*[2]

It also turns out that if we take the computational meaning for the existential quantifier, then we can define from it the one used in many logic books, and in previous CS3110 lecture notes. Some books use $\bigvee x : type$ for the non computational existential quantifier, and they think of it as a "big or" operator. We might take up the relationship between these two quantifiers later, but for now the key point is that our meaning for "exists" is "we can construct" or "we can build."

We also provide a computational meaning to the universal quantifier. When we say $\forall x : \alpha \ .P(x)$, we mean that we can construct a computable function f that given any value a of type α, the function will compute evidence $f(a)$ that provides us one reason we know $P(a)$.[3]

Induction on the integers There is a single induction rule for proving properties of the mathematical integers. We will take the OCaml Specification type *int* to be this mathematical type. *We can type this rule exactly with dependent types and then remarkably, give an OCaml recursive definition for how to compute with induction.* This is a strong fundamental link of mathematics with computation.

Recall from Lecture 15 that we need the idea of *large types* to define the dependent types. We will take β to be a function from *int* into $Type$, that is $\beta : int \rightarrow Type$.

[2]This kind of logic is called *constructive* or *intuitionistic* in logic books. This logic is useful in practice because it can express the so called classical logics of the textbooks, but if one starts with the classical logics, it is not possible to express the constructive ones faithfully. This choice makes the proof assistants for programming more generally applicable.

[3]The proof assistants also provide the reason that $P(a)$ makes sense as a proposition. Sometimes this is just a matter of type checking, as in Coq, and sometimes it is a matter of proof.

```
let rec ind (P:int -> Type)(n:int)

(dn: (u:int where u<0) -> v:P(u+1)-> P u )

(up: (u:int where u>0) -> v:P(u-1)-> P u )

base: P 0 :P n =

 case n<0 -> dn n (ind P (n+1) dn up base)
      n=0 -> base
      0<n -> up n (ind P (n-1) dn up base)
```

In the special case of induction on nat = (z:int where z $\geq$ 0), the rule is the usual induction rule along with its computational interpretation.

```
let rec nat_ind (P:nat -> Type) (n:nat) (base: P 0)

(up: (u:nat where 0< u)-> v:P (u-1) -> P u) : P n =

 if n=0 then base else up n (nat_ind P (n-1) base up)
```

5 Types look a lot like logic

The induction rules look very close to what we see in mathematics textbooks if we think of $P : nat \rightarrow Type$ as a proposition on natural numbers that we are trying to prove. The induction computation is computing something that belongs to the type $P\ n$ for for every n. The element *base* is something in the specific type $P\ 0$.

Let's look at a particular example of this rule in action that asserts something obvious. Suppose we define the type valued functions on *nat* by

```
let even (n:nat) :Type = (k:nat where 2 * k = n)

let odd (n:nat) :Type = (k:nat where (2 * k)+1 = n)

let thm_decide (n:nat) :Type = Even of even n | Odd of odd n

(n:nat) -> Even of even n | Odd of odd n
```

Now we can define a function by induction (recursion) which is in this dependent function type. We leave this as an optional exercise for now.