CS3110 Spring 2016 Lecture 21: Elements of Type Theory

Robert Constable

Topics

1. The logic problem – small modification suggested by the class to show a type with an unbounded number of programs. OCaml solution is given.

2. Comments on the "world's simplest" halting problem and a closer look at types.

   item The idea of "set types" such as $\{x : \text{int} | x \geq 0\}$ and other *dependent types*. The "propositions as types" idea.

3. Introduction to asynchronous computing and protocols.

1 Logic problem

This example reveals why any "programmable type" is logically true. The program is providing *computational evidence* that the proposition coded by the type is "true."

```
# fun x -> fun g -> fun f -> g(g(f)) x ;;
- : 'a -> (('a -> 'b) -> 'a -> 'b) -> ('a -> 'b) -> 'b = <fun>

# fun x -> fun g -> fun f -> g(g(g (f))) x ;;
- : 'a -> (('a -> 'b) -> 'a -> 'b) -> ('a -> 'b) -> 'b = <fun>

# fun x -> fun g -> fun f -> g(g(g(g (f)))) x ;;
- : 'a -> (('a -> 'b) -> 'a -> 'b) -> ('a -> 'b) -> 'b = <fun>
```

Comments on the halting problem example

When we think of the type `unit` as a set of elements, we see it as $\{(\,)\}$, and it is hard to imagine how there could be more than one element in the type. But all OCaml types are *partial types*. This means that the diverging computation $\perp$ belongs to every type, including `unit`. So at a minimum, we need to see the type as having $\perp$, the diverging element. This code makes that clear:

```
# let rec count n = if n = 0 then ( ) else count (n-1) ;;
val count : int -> unit = <fun>
```

If we conceive of these types in a constructive or computational way, we cannot say "For all x in the type `unit`, $x = (\,)$ or $x = \perp$."

We do not say this, because in a computational way of thinking about "or," as in "for all natural numbers n, n is either prime *or* not prime," we intend that we can tell. This means we cannot think of the partial type for `unit`, written as $\overline{\texttt{unit}}$ in Nuprl, as having only $\perp$ and $(\,)$ as members.

A better way to see $\overline{\texttt{unit}}$ is perhaps as consisting of many computations given by expressions, say $e_1, e_2, e_3, ..$ For some of them we know $e_i = (\,)$; for some we know $e_j = \perp$; for some we know $e_i = e_k$.

One new idea for enriching our analysis is related to PS5. We can measure the computational complexity of reduction of expression e to $(\,)$.

2 Unsolvability for halting in OCaml

To make the ideas clearer, in this section we use $\overline{\texttt{int}}$ for the OCaml type and `int` for the subtype of converging elements of $\overline{\texttt{int}}$.

`int`	$\overline{\texttt{int}}$
0, 1, -1, 2, -2,...	$\perp$, 0, 1, -1, 2, -2, ... As well as expressions, e_i that reduce to these values.

Fact 1. There is no OCaml function $h : \overline{\texttt{int}} \rightarrow$`bool` such that $h(n) =$ `true` iff $n\downarrow$. (p.97)

This proof follows the example from Lecture 20 that we cannot decide whether an element of $\overline{\texttt{unit}}$ converges. This proof method applies to any type T with some $t_0 \in$ T.

Computability in the Object Theory

Here is how Computational Type Theory (CTT) defines recursive functions. Consider the 3x+1 function with natural number inputs.

```
f(x) = if x=0 then 1
       else if even(x) then f(x/2)
            else f(3x+1)
            fi
       fi
```

Using Lambda Notation

```
f = λ(x. if x=0 then 1
         else if even(x) then f(x/2)
              else f(3x+1))
```

Here is a related term with function input f

```
λ(f. λ(x. if x=0 then 1
          else if even(x) then f(x/2)
               else f(3x+1)))
```

The recursive function is computed using this term.

Defining Recursive Functions in CTT

```
fix(λ(f. λ(x. if x=0 then 1
             else if even(x) then f(x/2)
                  else f(3x+1)
                  fi
             fi)))
```

Recursion in General

f(x) = F(f,x) is a recursive definition, also

f = λ(x.F(f,x)) is another expression of it, and the CTT definition is:

fix(λ(f. λ(x. F(f,x)))

which reduces in one step to:

λ(x.F(fix(λ(f. λ(x. F(f,x)))),x))

by substituting the fix term for f in λ(x.F(f,x)) .

Non-terminating Computations

CTT defines all general recursive functions, hence non-terminating ones such as this

fix(λ(x.x))

which in one reduction step reduces to itself!

This system of computation in the object language is a simple functional programming language.

Partial Functions

The concept of a partial function is an example of how challenging it is to include all computation in the object theory. It is also key to including unsolvability results with a minimum effort; the halting problem and related concepts are fundamentally about whether computations converge, and in type theory this is the essence of partiality. For example, we do not know that the 3x+1 function belongs to the type N -> N.

Partial Functions

We do however know that the 3x+1 function, call it f in this slide, is a partial function from numbers to numbers, thus for any n, f(n) is a number if it converges (halts).

In CTT we say that a value a belongs to the bar type $\bar{A}$ provided that it belongs to A if it converges. So f belongs to $A \rightarrow \bar{A}$ for $\bar{A} = N$.

Unsolvable Problems

It is remarkable that we can prove that there is no function in CTT that can solve the convergence problem for elements of basic bar types.

We will show this for non empty type $\bar{A}$ with element $\bar{a}$ that converges in A for basic types such as Z, N, list(A), etc. We rely on the typing that if F maps $\bar{A}$ to $\bar{A}$, then fix(F) is in $\bar{A}$.

Unsolvable Problems

Suppose there is a function h that decides halting. Define the following element of Ā:

d = fix(λ(x. if h(x) then ↑ else ā fi))

where ↑ is a diverging element, say fix(λ(x.x)).

Now we ask for the value of h(d) and find a contradiction as follows:

(In the lecture notes, we use ⊥ for ↑.)

Generalized Halting Problem

Suppose that h(d) = *t,* then d converges, but according to its definition, the result is the diverging computation ↑ because by computing the fix term for one step, we reduce

d = fix(λ(x. if h(x) then ↑ else ā fi))

to d = if h(d) then ↑ else ā fi .

If hd(d) = *f,* then we see that d converges to ā.

New idea

What recursive types make sense? OCaml allows this type:

```
# type int_maps = |Base of int | Fun of (int -> int_maps) ;;
Fun(fun x → (Fun(fun y → x))) ∈ int_maps
```

Can there be a type like this that makes "logical sense"?

In set theory, we can't have a set

$$(\text{int} \to \text{int}) = \text{int} \cup (\text{int} \to \text{int}) \to (\text{int} \to \text{int}).$$

Propositional Evidence

Suppose that we have evidence types [A] for the atomic propositions A. Here is how to construct evidence for compound formulas in a model $\mathcal{M}$.

[A & B] == [A] x [B]
[A v B] == [A] + [B]
[A => B] == [A] $\rightarrow$[B]
[false] == ϕ (the empty set)
[$\neg$A] == [A] => ϕ

Evidence for Quantified Propositions

The evidence types for quantified formulas use the dependent types over the universe $U_{\mathcal{M}}$ of the model $\mathcal{M}$.

[All x. B(x)] == x: $U_{\mathcal{M}} \rightarrow$ [B(x)]

[Exists y. B(y)] == y: $U_{\mathcal{M}}$ x [B(y)]

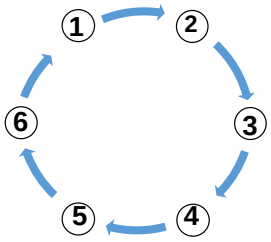
Specification for Leader Election in a Ring

Leader Election

In a Ring R of Processes with Unique Identifiers (uid's)

Specification

Let R be a non-empty list of locations linked in a ring



Let $n(i) = dst(out(i))$, the next location

Let $p(i) = n^{-1}(i)$, the predecessor location

Let $d(i,j) = \mu k \geq 1.\ n^k(i) = j$, the distance from i to j

Note $i \neq p(j) \Rightarrow d(i,p(j)) = d(i,j)-1$.

Specification, continued

$$Leader(R,es) == \exists\, ldr: R.\ (\exists e@ldr.\ kind(e)=leader)\ \&\ (\forall i:R.\ \forall e@i.\ kind(e)=leader \Rightarrow i=ldr)$$

Theorem $\forall\, R:List(Loc).\ Ring(R)$

$\exists\, D:Dsys(R).\ Feasible(D)\ \&$

$\forall es: ES.\ Consistent(D,es).\ Leader(R,es)$