CS3110 Fall 2013 Lecture 21: A Bit of Type Theory – Unsolvable Specifications

Robert Constable

1 Lecture Plan

1. Brief remarks on Lecture 20, asymptotic complexity
2. Fixed point operators in OCaml
3. Unsolvable (unimplementable) specifications
4. Additional types for specification

2 Review, Overview, and Comments

We have seen that the `efix` operator used in Lecture 19 to define recursive functions gives us a clean way to define the semantics of recursive functions and count the steps of evaluation of recursive functions.

The fixed point operators, `fix` and `efix`, also allow us to show that there are unimplementable specifications. These are also called *unsolvable problems* in the theory of computing. Turing's Halting Problem is one of the most famous unsolvable problems, and we can prove his result in an almost trivial way, one that captures the very essence of his idea in a few lines of proof that you will be able to show your friends and relatives all the rest of your life.

Moreover it is an important result of type theory that if we have fix operators, induction on the natural numbers, `nat` and bar induction (which we will only briefly mention in this course), then from the lazy recursive types, called *co-inductive types*, we can define the recursive types. Later in the course we will briefly examine *co-inductive types*. They are also briefly mentioned by Kozen as lazy types in his CS3110 2009 fall lectures.

3 Reading and Sources

The idea for the proof given here is from results with Scott Smith [3, 2]. Similar ideas can be found in [1]. Many students will have seen the basic idea for the halting problem in CS4810 and in books such as Hofstadter [4].

4 Review of Lecture 20

Mike George lectured about *asymptotic complexity* and defined the Big-Oh notation precisely in terms of one function *asymptotically dominating* another one. This is a key definition that you should know and is restated here.

Definition: Let f and g be functions from $\mathbb{N}$ to $\mathbb{N}$. We say that g asymptotically dominates f if and only if

$$\exists c, k : \mathbb{N}. \forall x : \mathbb{N}. (k \; < \; x \; \Rightarrow f(x) \leq c \times g(x)).$$

Mike used step counting functions for f and g. He called them T_f, T_g.

He showed that if you make the wrong induction hypothesis, using $\exists c, k : \mathbb{N}$, you can prove that a certain function has constant time, $O(1)$, asymptotic complexity when it does not, it's $O(n)$.

5 The Fix Operators

We introduced the `efix` operator when we began our study of computational complexity, to give us a *clean evaluation semantics for OCaml recursive functions.* This is a key idea from the course and will be on Prelim II for sure.

For this lecture we want to examine an even simpler operator, `fix` which is used to define recursive functions in languages that use *lazy evaluation.* In lazy evaluation of a function application `f a`, the argument `a` is substituted into the body of the function `f` and only evaluated when it is needed. The OCaml `if bexp then expt else expf` uses lazy evaluation in that if the Boolean expression is true, then `expf` is not evaluated, and if it is false, then `expt` is not evaluated.

Prof. Kozen discusses lazy evaluation in 2009fa lecture 24, where he also discusses streams (co-lists). You should read those notes. Some functional programming languages such as Haskell, named for Haskell Curry whom we have met before in lecture, use only lazy evaluation. Kozen discusses how they make that more or less efficient. OCaml has a Lazy mode of evaluation, but we will not discuss it in this course. Instead we will examine using `fix` how lazy evaluation works.

Here are the two fix operators defined in OCaml.

```
(* The fix operator is used for lazy evaluation. *)

# let rec fix f = f (fix f);;
val fix : ('a -> 'a) -> 'a = <fun>

(* The efix operator is used for eager evaluation. *)

# let rec efix f x = f (efix f) x ;;
val efix :  (('a -> 'b) -> ('a -> 'b)) -> 'a -> 'b = <fun>
```

Compare this account to *Kozen 2009fa Lecture 6* on evaluating recursion.

```
(* We will use the type unit -> bool *)


# let uid (x:unit) :unit = x;;
val uid : unit -> unit = <fun>

# let unit_div = fix uid;;
Stack overflow during evaluation (looping recursion?).
```

Consider the summation function from the first week of the course.

```
let rec sum (n : int) : int =
  if n <= 0 then 0
  else n + sum (n - 1)
```

Consider this as a function of `sum`, say

`λsum.λn.if n ≤ 0 then 0 else n + sum (n-1)`

The type is `(sum : (int -> int)) (n : int) : int`
The same as
    `(int -> int) -> (int -> int)`

What is the type of

`fix (λsum.λn.if n ≤ 0 then 0 else n + sum (n-1))`?

`((int -> int) -> (int -> int)) -> int -> int`

How does `fix (λsum.λn.body(sum, n))` reduce?

Recall `fix f = f (fix f)`

In one substitution of `(fix f)` for `sum` we get

    `λn.body(fix f, n)`

which is

    `λn.if n ≤ 0 then 0 else n + fix (λsum.λn.__ )`

In a lazy evaluation, the `fix` in `body` would not be expanded until it was applied.

In some languages such as Haskell and Nuprl function evaluation is lazy.

What happens in OCaml?

> Consider just `fix f = f (fix f)`.
>
> What is evaluated first in the application `f (fix f)`?

We can invoke *lazy evaluation* in OCaml, but it can be quite expensive since we reevaluate the `(fix f)` every time it is called.

> Haskell optimizes for this.
>
> Nuprl also allows call by value (eager eval).

However, call by value and lazy evaluation each have advantages. Another advantage of call by name (lazy eval) is that we can define co-recursive data types such as *streams* ... unbounded lists such as $[1, 1/2, 1/3, 1/4, \ldots]$.

How might streams be useful?

6 An Unimplementable Specification, e.g. An Unsolvable Computing Problem

Lazy evaluation (call by name) has an interesting theoretical value. It allows what I think is the simplest proof of the unsolvability of the halting problem that I know– discovered with one of my students, Prof. Scott Smith of JHU. Here's the proof.

We consider only functions `unit -> bool`.

Suppose we try to define a function with the following intuitive specification:

intuitive halting function on unit Find an OCaml computable function *halt* which will take as input any expression t which the type checker infers has type *unit* and will return *true* if and only if t computes to `()`, the only canonical value of type *unit*. Given any purported OCaml function *halt*, what would we expect to know about it to be sure that it decides halting?

This seems simple, we expect to know for all t in *unit*, if $(halt\ t) = true$ then t converges to `()`, and if t converges to `()` then $(halt\ t) = true$. This is actually enough to do most of the work we want. Can we say this in OCaml SL? What about just saying this?

`(halt:unit -> bool * (t:unit -> ((halt t = true)` $\Leftrightarrow$ `(t` $\downarrow$ `() = true) )))` where the double arrow $\Leftrightarrow$ is the Boolean equivalence relation [1] and where $t \downarrow$ `()` means that t evaluates to `()` in OCaml? But $t \downarrow$ `()` is not a boolean value, it is a proposition that expands to the refinement type $(n : nat * (eval\ n\ t) = $ `()`$)$ which makes an existence claim, that we can find the natural number n for which $(eval\ n\ t) = $ `()`.

This simple version of the claim is enough for our initial study. It might help to note that we could define an OCaml function *eval n exp* that evaluates the syntactically correct expression *exp* for n steps. OCaml could provide this function for us. So $(t \downarrow$ `()`$)$ means just that $(n : nat\ * eval\ n\ t = $ `()`$)$.

An OCaml SL specification of halting Let `terms` be the expressions of OCaml that are syntactically correct as determined by the parser. To know that a syntactically correct term has type `unit`, we require that the type inference algorithm assigns type *unit*. Let `unit_term` be this dependent type (`t` : `term where type(t) = unit`).

Next, let `eval` be the OCaml computable function that evaluates terms for `n` steps and returns the value; that is, `eval n t` $=$ `t`$'$. This function could be provided by OCaml, but it is not supplied in the current implementation.

[1] Note that $exp_1 \Leftrightarrow exp_2$ for Boolean expressions means both are true or both are false, e.g. $\neg\, exp_1$ && $\neg\, exp_2$ || exp_1 && exp_2.

Given the above background, here is a specification for a halting function on `unit`. (We won't hold you responsible for understanding the details of this dependent type, just its intuitive meaning.)

Definition: Given two terms t_1 and t_2, we say that $t_1 \sim t_2$ if and only if either both diverge or both converge to the same canonical value.

Definition: An OCaml computable function *halt* that has type `unit -> bool` decides halting on `unit` if it has the property that given any expression t which type checks to be in `unit`, *halt* decides whether t converges to the canonical value `()`. Thus *halt* t has value *true* if and only if term t converges to `()`. If *halt* t returns $false$, then `t` $\sim$ `unit_div`, i.e. t diverges.

```
(halt:(u:unit -> bool) where (halt u = true) iff u ∼ ()
    & (halt u = false) iff (u ∼ unit_div).
```

Suppose we have found *halt* to be any function of this type. We could also say that *halt* is *in* this type.

We now prove that the Halting Specification cannot be implemented in OCaml, e.g. there is no OCaml computable function *halt*.

6.1 The argument using fix

Define this function

```
    diag = fun x -> if halt x then unit_div else ().
```

Next define `let d = fix diag`. We know that fix does not give the behavior we want unless we use lazy computation, but let's suppose that we use the Lazy module of OCaml or that we are in Nuprl or Haskell with dependent types. Then we have this very simple argument.

1. Note that d belongs to *unit* by the fix typing because *diag* $\in$ *unit* `->` *unit* so fix *diag* $\in$ *unit*, i.e. $d \in$ *unit*.

2. Now we examine the specification for *halt*. By the fact that the types in the specification are *total*, we know that *halt* must give a value on *d*.

3. The value *halt d* is either *true* or $false$. We consider both possibilities and see that neither can occur.

(a) Suppose that *halt d* $=$ *true*, then we see that *d* will be a diverging element, which contradicts the definition of *halt*.

(b) So we must have *halt d* $=$ $false$ since *halt* is a total function and must produce $false$.

But in this case, *diag d* $=$ `()`, which again contradicts the definition of *halt*. So there can't be any OCaml function that implements the specifications.

Now let's see how this simple argument works with *efix* a function that gives us a behavior closer to the one we want in OCaml.

6.2 The argument using efix

```
# let rec efix f x = f (efix f) x ;;
val efix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

Recall that the type of *efix* is

> *efix* $\in$ ((α -> β) -> (α -> β)) -> (α -> β)
> *fix* $\in$ (α -> α) -> α
>
> This shows that *efix* is a special case where α is a function type (α -> β).

So instead of *diag* and *halt* applied to `unit`, we can use `(unit -> unit)`.

> We ask for *halt* to decide whether $f \in$ `unit -> unit` halts on `()`, e.g. *halt* $f = true$ *iff* f `()` halts.

Now define

```
diag = fun f -> if halt (f ()) then λx.unit_div
                 else λx.()

diag ∈ (unit -> unit) -> (unit -> unit)

efix diag ∈ unit -> unit

let d = efix diag, d ∈ unit -> unit.
```

If $halt\ d = true$ then d `()` diverges. So $halt\ d$ is incorrect.

If $halt\ d = false$ then d `()` converges to `()`, so $halt\ d$ is incorrect.

This argument applies to OCaml SL and OCaml as defined.

How "realistic" is this version of the Halting Problem compared to Alan Turing's? Turing allowed us access to the complete syntax of his machines as data for decide.

What if we "reflected" the syntax of OCaml inside OCaml using variants to represent the BNF grammar?

Now at least we might have a "fighting chance."

But the above result shows that all such efforts are futile because undecidability does not depend on them.

> I think that Church already knew this in 1935, but he used a more complex argument, based on a logic not a programming language.

References

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Symantics*, volume 103 of *Studies in Logic*. North-Holland, Amsterdam, 1981.

[2] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 360–371, Edinburgh, UK, 1988. IEEE Computer Society Press. (Cornell TR 88-904).

[3] 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.

[4] D.R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid.* Basic Books, New York, 1979.