Lecture 11

CS 4860

September 29, 2016

1 Developing a Theory of Polymorphic Programming Logics

We are using other logical theories, such as the Propositional Calculus, to guide our thinking about a precise formal theory of polymorphic programming logics. For instance, are there analogs of the concepts of *consistency* and *completeness*? To explore these questions, we need a precise semantics for the polymorphic type "assertions" such as

$$\sim\sim (\alpha \vee \ \sim \alpha)$$

$$\alpha * \beta \Rightarrow \alpha \ \vee \ \beta.$$

We discussed the following ideas in lecture for developing such a theory. It will depend on a precise account of the syntax of programs and data and the *rules of computation*. Our programs are functional, so among the basic objects of the theory are functions. We use the Nurpl syntax for these, $\lambda(x.b(x))$. We say that functions are *canonical values* of the theory. This means that they are irreducible by the computation rules. We use functions by *applying* them. There is an operator for this, $ap(f; a)$. We read this as "the application of function f to data element a." The logic will have this *rule for evaluation*,

$$ap(\lambda(x.b(x)); a) \text{ reduces to } b(a/x).$$

$$ap(\lambda(x.b(x)); a) \ \downarrow \ b(a/x)$$

We sometimes just write $b(a)$. That is, we substitute the value a for all occurrences of x in $b(x)$.

This method of evaluation is called *lazy evaluation* because we do not attempt to evaluate the expression a first. If the result of this evaluation is another application, say $ap(\lambda(y.e(y)); b)$, then we continue evaluating.

Another canonical value of our theory is the pair. We write it as $pair(a; b)$. We might abbreviate it at times as $< a, b >$. We need a corresponding operator to decompose a pair and access its components. We use the Nuprl operator

$$spread(p; x, y.exp(x, y)).$$

Like $ap(f;a)$, $spread(p;x,y.exp(x,y))$ is a *noncanonical* term of the logic. There is an evaluation rule for it:

$$spread(pair(a;b);x,y.exp(x,y)) \text{ evaluates in one step to } exp(a,b)$$

(We can also write this with the substitution operator, $exp(a/x, b/y)$.)

The canonical values of $\alpha \ \vee \beta$ (e.g. $\alpha \mid \beta$ in variant notation) are $inl(a)$, $inr(b)$, where a is of type α and b of type β. The term "*inl*" stands for "in left," and "*inr*" stands for "in right."

The corresponding destructor is $decide(d;l.left(l);r.right(r))$. This expression is non-canonical. It evaluates as follows,

$$decide(inl(a);l.left(l);r.right(r)) \text{ evaluates to } left(a)$$

$$decide(inr(b);l.left(l);r.right(r)) \text{ evaluates to } right(b).$$

The expressions $left(a)$ and $right(b)$ might evaluate further. The expressions $left(l)$ and $right(r)$ are just mnemonic not canonical.

1.1 Consistency and Completeness

- A good analogue of *consistency* is the notion that a program in the polymorphic type is correct for any specific types substituted for the type variables, $\alpha, \beta, \gamma, \ldots$.
- An analogue of *completeness* is that we can find a program in any polymorphic type which has concrete instances, e.g. for which there are programs when we use concrete types such as *int, bool, char,* etc. for the type variables.

2 Relating Smullyan's account of First-Order Logic to Refinement Logic for FOL

First-Order Logic (FOL) is about *predicates*, also called *relations*, on an arbitrary non-empty domain D of *individual* elements. Concrete examples we will study are based on the following domains.

- $\mathbb{N}$ The *natural numbers*: 0,1,2,...
- $\mathbb{Z}$ the *integers*: 0, ± 1, ± 2, ...
- $\mathbb{Q}$ the rational numbers
- $\mathbb{R}$ the real numbers

- *Sets* the collection of all sets
- $\mathbb{U}_i$ the universes of types $\mathbb{U}_1$, $\mathbb{U}_2$, ...

Over the domain of natural numbers $\mathbb{N}$ we will use these predicates (relations):

- $eq(x, y)$ a binary relation of equality
- $add(x, y, z)$ the relation $x + y = z$
- $mult(x, y, z)$ the realtion $x * y = z$
- $zero(x)$ the predicate "x is zero"

In FOL we do not have variables for arbitrary relations. These are allowed in Higher-Order Logic, HOL. In some versions of HOL there are also variables for function on the domain D, e.g. $f : D \to D$, $f : D \times D \to D$, ...

2.1 FOL Refinement Rules

1. Exists Construction

   $H \vdash \exists x.B(x)$ by $pair(\ d\ ;\ slot_b(d)\ )$

   $\vdash d \in D$ by $obj(d)$

   $\vdash B(d)$ by $slot_b(d)$

2. Exists Decomposition

   $H,\ e:\ \exists y.B(y),\ H' \vdash G$ by $spread(e; d, r.\ slot_g(d, r)\ )$ new d, r

   $H,\ d : D, r : B(d),\ H' \vdash G$ by $slot_g(d, r)$

3. All Construction

   $H \vdash \forall x.B(x)$ by λ(x. $slot_b$ (x)) new x

   $H,\ x : D \vdash B(x)$ by $slot_b$(x)

4. All Decomposition

   $H,\ f : \forall x.B(x),\ H' \vdash G$ by $ap(f; d)$

   $\vdash d \in D$ obj(d)

   H, " ", H', v:B(d) $\vdash$ G by $slot_g$(v) note, $v = ap(f; d)$

Here is a proof of an FOL proposition from page 56 of Smullyan.

1. $\exists x.(P(x) \ \vee \ Q(x)) \ \Rightarrow \ (\exists x.\ P(x) \ \vee \ \exists x.\ Q(x))$

   $\vdash \exists x : D.\ (P(x) \ \vee \ Q(x)) \Rightarrow (\exists x : D.\ P(x) \ \vee \exists x : D.\ Q(x))$ by $\lambda(h.\_\_\_\_\_)$

   h: $\exists x : D.\ (P(x) \ \vee \ Q(x)) \ \vdash \ \exists x : D.\ P(x) \ \vee \ \exists x : D.\ Q(x) \quad spread(h; d, or.\_\_\_\_\_)$

   d: D, or:$(P(d) \vee Q(x)) \ \vdash \ \exists x : D.\ P(x) \vee \exists x : D.\ Q(x) \quad decide(or;\ l.\_\_\_\_\_\ ;\ r.\_\_\_\_\_\ )$

   l: P(d) $\vdash \ \exists x : D.\ P(x) \ \vee \ \exists x : D.\ Q(x) \quad inl(\qquad)$

   l: P(d) $\vdash \ \exists x : D.\ P(x) \qquad < d, l >$

   r: Q(d) $\vdash \ \exists x : D.\ P(x) \ \vee \ \exists x : D.\ Q(x) \quad inr(\qquad)$

   r: Q(d) $\vdash \ \exists x : D.\ Q(x) \qquad < d, r >$

   The "program" or proof term or extract is this lambda term

   $$\lambda(h.spread(h; d, or.decide(or; l.inl(< d, l >); r.inr(< d, r >)))).$$

   We can translate this proof term or extract into a program in any functional programming language.

   Exercise: Write the proof term in OCaml of Haskell of Lisp or Java, your favorite programming language with functions as data.

2. $\vdash \exists x.(P(v) \Rightarrow C) \Rightarrow (\forall x.P(x) \Rightarrow C)$ by $\lambda(h.\_\_\_\_\_)$

   h: $(\exists x.(P(x) \Rightarrow C) \vdash \ \forall x P(x) \Rightarrow C \qquad spread(h; d, i.\_\_\_\_\_)$

   $d : D,\ i : P(d) \ \Rightarrow C \ \vdash \forall x P(x) \Rightarrow C \qquad \lambda(a.ap(i; ap(i; ap(a; d))))$

   $a :\ \forall x.P(x) \ \vdash \ C \qquad ap(a; d) \ = \ a'$

   a': $P(d) \ \vdash \ C \qquad ap(i; a') \qquad v = \ ap(i; a')$

   $\vdash P(d)$ by a'

   $v \ : \ C \ \vdash \ C$

$\lambda(h.spread(h; d, i.\ \lambda(a.ap(i; ap(i; ap(a, d))))))$