Lecture 4

CS 4860

Thursday, September 1, 2016

1. So far we have focused on the remarkably strong connection between functional programming and propositional logic. Key to this correspondence is the idea of "polymorphic types." This idea involves using type variables such as $\alpha, \beta, \gamma, ...$ (written in OCaml as 'a, ' b, ' c, ...). When we find a program in the type $(\alpha * \beta) \Rightarrow \alpha$, we know that it works for all concrete choices for α, β using int, bool, unit, char, list, etc. The polymorphic code is $fun\ x \rightarrow$ let$(a, b) =$ x in a. If we prove the input $(3, 5)$ in type $int * int$, the function produces 3. If the input is $(true, 17)$ the value is $true$.

2. We will see in this lecture that not all *polymorphic types* are programmable. For example, we studied in considerable detail in Lecture 3 that the type $\sim (\alpha * \beta) \Rightarrow \sim \alpha \vee \sim \beta$ is not programmable. Also, we could not express the type in this simple logical way in OCaml. We needed to define (α, β)union = L of α| R of β and write

   $$((\alpha * \beta) \rightarrow void) \rightarrow (\alpha, \beta) \text{ union },$$

   where (α, β)union is a defined type. The fact that Ocaml is a somewhat verbose language will lead us to eventually use the programming language of the Nuprl proof assistant. It is much more compact, and it is grounded in McCarthy's idea of *abstract syntax*.

   It is interesting that we can build *display forms* in Nuprl that allow the system to display types and programs in a wide variety of "surface syntax." We illustrate these later in the course.

3. We examine now some *unprogrammable polymorphic types*. Consider $\alpha \vee (\alpha \Rightarrow void)$, written as $\alpha \vee \sim \alpha$. To be legitimate OCaml, we would need to write

   $$\text{type } (\alpha, \beta)\text{union} = \text{Left of } \alpha | \text{ Right of } \beta$$

   and specialize to

   $$\text{type } (\alpha, \alpha \rightarrow void)\text{union} = \text{Left of } \alpha | \text{ Right of } (\alpha \rightarrow void).$$

   How could we program this type?

   If we had concrete types, say $(bool, bool \rightarrow void)$union, then we know that $L\ true$ is a value, but $R\ fun\ x \rightarrow ?$ can not be completed. There is no function in $(bool \rightarrow void)$.

   We can not find an element in $\alpha \vee \sim \alpha$ because any concrete value requires that we know an element of α or that $\alpha = void$.

   We can find an element in $\alpha \rightarrow \alpha$ because $fun\ x \rightarrow\ x$ belongs to *all instances* of this polymorphic type, especially $bool \rightarrow bool$, $int \rightarrow int$, $unit \rightarrow unit$, $void \rightarrow void$, etc. Once we see one *unprogrammable polymorphic type*, we can find an unbounded number of them. Here are more examples,

   - Pierce's Law $((\alpha \Rightarrow \beta) \Rightarrow \alpha) \Rightarrow \alpha$
   - "Dreaded rule 8" $\sim\sim \alpha \Rightarrow \alpha$
   - $\sim (\alpha * \beta) \rightarrow\ (\sim \alpha \vee \sim \beta)$ or $\sim (\alpha * \beta) \rightarrow (\sim \alpha, \sim \beta)union$

Smullyan adopted a purely mathematical, noncomputational approach to the proof of `Valuation Theorem`. In his text, he writes:

> Consider a single formula X and an interpretation v_0 of X – or for that matter any assignment v_0 of truth values to a set of propositional variables which includes at least all variables of X (and possibly others). It is easily verified by induction on the degree of X that there exists one and only one way of assigning truth values to all *subformulas* of X such that the *atomic* subformulas of X (which are propositional variables) are assigned the same truth values as under v_0, and such that the truth value of each *compound* subformula Y of X is determined from the truth values of the immediate subformulas of Y by the truth-table rules $B_1 - B_4$. [We might think of the situation as first constructing a formation tree for X, then assigning truth values to the end points in accordance with the interpretation v_0, and then working our way up the tree, successively assigning truth values to the junction and simple points, in terms of truth values already assigned to their successors, in accordance with the truth-table rules]. In particular, X being a subformula of itself receives a truth value under this assignment; if this value is *true* then we say that X is true *under the interpretation* v_0, otherwise *false* under v_0. Thus we have now defined what it means for a formula X to be true under an *interpretation.* [16, pp. 10–11]

That is, we wish to show that for any formula Z and interpretation v_0 of the variables of Z, there exists one and only one function f assigning truth values to the subformulas of Z such that f is an extension of v_0 and f is a partial Boolean valuation on subformulas of Z.

The existence of valuations is expressed in the standard way:

$$\forall x\colon \mathit{form}.\ \forall v_0\colon \mathit{Var}(x) \to \mathbb{B}.\ \exists f\colon \mathit{Sub}(x) \to \mathbb{B}.\ \mathit{valuation}(x, v_0, f)$$

The formal definitions are straightforward. We define a datatype for the formulas of propositional logic by:

$$\mathit{form} := \mathit{var}(\mathit{Atom}) \mid \mathbf{not}\ (\mathit{form}) \mid \mathit{form}\ \mathbf{and}\ \mathit{form} \mid \mathit{form}\ \mathbf{or}\ \mathit{form} \mid \mathit{form}\ \mathbf{implies}\ \mathit{form}$$