The most important topics and details of the day.

March 27, 2006

**Topic 1. Deduction Rules.** Deduction rules specify a means of logically deducing facts from other facts. We will write them in the following form, where $P$ and $Q$ range over arbitrary assertions:

$$
\begin{array}{c}
P_1 \\
\vdots \\
P_n \\
\hline \\
Q
\end{array}
$$

Rules in this form should be read “if $P_1$ through $P_n$ are deducible, then so is $Q$. Each of $P_1$ through $P_n$ are called *precedents* of the rule, and $Q$ is called the *consequent*. If $n = 0$, i.e. the rule has no precedent, then it is called an *axiom*.

**Topic 2. Operational Semantics of BOOL.** The operational semantics of BOOL is defined as a relation $e \Rightarrow v$ on expressions and values. The relation is defined in terms of deduction rules; we say that a particular relation $e \Rightarrow v$ holds iff it can be derived as the consequent of one of the following deduction rules, where each of the precedents are valid:

<table>
<thead>
<tr>
<th>$e$</th>
<th>$\Rightarrow$</th>
<th>$v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{VAL}$</td>
<td>$e \Rightarrow \text{True}$</td>
<td>$\text{NOT}$</td>
</tr>
<tr>
<td>$\text{NOT}$</td>
<td>$\text{Not } e \Rightarrow \text{False}$</td>
<td>$\text{AND}$</td>
</tr>
<tr>
<td>$\text{AND}$</td>
<td>$e_1 \Rightarrow \text{False}$</td>
<td>$e_2 \Rightarrow \text{False}$</td>
</tr>
<tr>
<td>$\text{AND}$</td>
<td>$e_1 \Rightarrow \text{True}$</td>
<td>$e_2 \Rightarrow \text{False}$</td>
</tr>
<tr>
<td>$\text{AND}$</td>
<td>$e_1 \Rightarrow \text{True}$</td>
<td>$e_2 \Rightarrow \text{True}$</td>
</tr>
<tr>
<td>$\text{AND}$</td>
<td>$e_1 \Rightarrow \text{False}$</td>
<td>$e_2 \Rightarrow \text{False}$</td>
</tr>
<tr>
<td>$\text{AND}$</td>
<td>$e_1 \Rightarrow \text{True}$</td>
<td>$e_2 \Rightarrow \text{True}$</td>
</tr>
</tbody>
</table>

(And so on for the rules relevant to $\text{Or}$). Note that this set of rules includes one axiom, the $\text{VAL}$ rule, specifying that values evaluate to themselves. Also, note that these are rule *schemas*, with the syntactic categories $e$ and $v$ ranging over any term instance of the category. Which is to say, any valid *instance* of one of these derivation rules will instantiate syntactic categories with terms, to derive a particular evaluation relation. Since any rule that is not an axiom must be substantiated via other evaluation relations as precedents, the deduction of a particular evaluation relation will have a tree-like structure, with instances of $\text{VAL}$ at the leaves. Thus, derivations have a recursive structure, and we can *induct* on them. Examples:

<table>
<thead>
<tr>
<th>$e$</th>
<th>$\Rightarrow$</th>
<th>$v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{True}$</td>
<td>$\Rightarrow$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\text{True}$</td>
<td>$\Rightarrow$</td>
<td>$\text{False}$</td>
</tr>
<tr>
<td>$\text{True And True}$</td>
<td>$\Rightarrow$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\text{False}$</td>
<td>$\Rightarrow$</td>
<td>$\text{False}$</td>
</tr>
<tr>
<td>$\text{True}$</td>
<td>$\Rightarrow$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\text{Not False}$</td>
<td>$\Rightarrow$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\text{True And Not False}$</td>
<td>$\Rightarrow$</td>
<td>$\text{True}$</td>
</tr>
<tr>
<td>$\text{Not (True And Not False)}$</td>
<td>$\Rightarrow$</td>
<td>$\text{False}$</td>
</tr>
</tbody>
</table>

**Topic 3. Properties of evaluation.** Programming languages should be “well-behaved”. What this means can often be mathematically characterized, and, if one is careful about definitions, can be proven. One desirable characteristic of a programming language is that it be *deterministic*—that is, it should have
predictable evaluation behavior, and not give a different answer for different runs of the same program. We can state and prove this property as follows; note that the proof follows by induction on the structure of derivations.

**Proposition 1.1** BOOL is deterministic; if \( e \Rightarrow v \) and \( e \Rightarrow v' \) then \( v = v' \).

*Proof.* By structural induction on the derivation of \( e \Rightarrow v \).

Another property of interest is normalization. That is, it would be a nice to know that programs in the language always evaluate to a value. Now, we’ve observed that this is not the case for OCaml, due to non-termination, and type errors can also interfere with normalization. However, we can prove a strong normalization result for BOOL, since it is a pretty simple language. Note that since expressions are terms in a recursively defined language, we can use structural induction on expressions to reason about our language:

**Proposition 1.2** BOOL is strongly normalizing; i.e., for all boolean expressions \( e \), there exists \( v \) such that \( e \Rightarrow v \).

*Proof.* By structural induction on \( e \).

**Topic 4. Language implementations.** As we have seen, BNF grammars are our tool for specifying the syntax of languages (hereafter referred to as the concrete syntax), while operational semantics are our tool for specifying the semantics of languages. However, these specifications are conceptual. To implement languages, the syntax and semantics of expressions must have computational counterparts, expressed in an existing computer language. The implementation of syntax is accomplished via abstract syntax, while the implementation of semantics will be accomplished via an interpreter.

**Topic 5. Abstract syntax.** Abstract syntax is the machine representation of language terms. The abstract syntax in an OCaml implementation is very effectively encoded as a variant type. Indeed, functional languages are extremely suited for writing interpreters (and compilers). In particular, our abstract syntax is implemented as follows:

```ocaml
type boolexp = Bool of bool | And of boolexp * boolexp | Or of boolexp * boolexp | Not of boolexp
```

It is the job of the lexer and the parser to transform concrete into abstract syntax for an actual implementation; in this course we will not be concerned with the details of those, but they will be provided when you begin to construct interpreters for the hws. However, we can define a theoretical transformation from concrete to abstract syntax, to formalize the correspondence of the two:

- \([\text{True}]\) = \(\text{Bool(true)}\)
- \([\text{False}]\) = \(\text{Bool(false)}\)
- \([\text{Not } e]\) = \(\text{Not}(\llbracket e \rrbracket)\)
- \([e_1 \text{ And } e_2]\) = \(\text{And}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket)\)
- \([e_1 \text{ Or } e_2]\) = \(\text{Or}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket)\)

example:

- \([\text{True And False}]\) = \(\text{And(Bool(true), Bool(false))}\)

**Topic 6. Interpreters.** Given the abstract syntax, we can now implement the semantics as an interpreter, which in OCaml is a function \(\text{eval}\) that has type \(\text{boolexp} \to \text{boolexp}\). Note that since our semantics is well-defined, as are both the concrete and abstract syntax and the transformation from the former to the latter, we can precisely specify, as a theorem, the desired behavior of the interpreter:
Theorem 1.1 (Correctness of implementation) $e \Rightarrow v \iff \text{eval } ([e]) \Downarrow [v]$.

We may implement a function with the desired properties as follows:

```ocaml
let rec eval exp =
    match exp with
    | Bool(b) -> Bool(b)
    | Not(exp0) ->
        (match eval exp0 with
          | Bool(true) -> Bool(false)
          | Bool(false) -> Bool(true))
    | And(exp0,exp1) ->
        (match (eval exp0, eval exp1) with
          | (Bool(true), Bool(true)) -> Bool(true)
          | _ -> Bool(false)
    | Or(exp0,exp1) ->
        (match (eval exp0, eval exp1) with
          | (Bool(false), Bool(false)) -> Bool(false)
          | _ -> Bool(true)
```