The most important topics and details of the day.

February 23, 2006

**Topic 1. Records.** In addition to variants, *records* are another sort of user-defined datatype in OCaml. Intuitively, records are collections of named values, very much like a `struct` in C/C++, except that basic record fields are immutable. Record types must be declared before they are constructed, by specifying field names and their associated types. Field values are accessed (i.e., records are deconstructed), via familiar “dot” notation; i.e., if \( r \) is a record, and \( l \) is a field name in \( r \), then \( r.l \) accesses the associated field value.

Examples:

```ocaml
type student = { name : string; email : string; gpa : float}
let bob = { name = "bob"; email = "bob@uvm.edu"; gpa = 3.2}
bob.email ⇒ "bob@uvm.edu"

let rv = { a = 1 + 2; b = true}
rv : rt
rv.a ⇒ int
rv.a = 2 ⇒ false
```

**Topic 2. Mutation and state.** So far, we have considered the purely functional subset of OCaml. Now, we turn our consideration to mutable features in OCaml. Mutation is a form of side effect, where something happens “behind the scenes”, rather than as a direct result of evaluation. With mutable variables, what happens “behind the scenes” is that the store, where mutable values are saved, is changed, affecting the state of the computer.

Reference cells are the most basic sort of mutable variables. All reference cells have a `ref` type, and may be dereferenced and updated. Reference cells are in fact memory locations, analogous to a pointer. There are three basic operations on reference cells:

- creation: \( \text{ref } v \downarrow c \), where \( c \) is a fresh reference cell; stores \( v \) at store location referenced by \( c \)
- dereferencing: \( !e \downarrow v \) returns the contents of \( c \) iff \( e \downarrow c \)
- assignment: \( e := e' \downarrow \text{unit } \) iff \( e \downarrow c \), and stores \( v \) in \( c \) where \( e' \downarrow v \).

**NB:** Assignment does not affect variable bindings. Also, now we see the usefulness of type `unit`, as the “dummy” result of a primarily side-effecting computation.

Example:

```ocaml
let a = ref 5
a : int ref
!a \downarrow 5
a := 4
!a \downarrow 4
a = 3 not well typed
```
**Topic 3. Sequencing and evaluation order.** Side effects make evaluation order really matter. For example, suppose \(x\) is a value of type \(\text{int ref}\) and suppose we perform the operations \(x := 1\) and \(x := 2\); the order in which we perform these operations has an effect on the evaluation of subsequent expressions such as \(!x + 2\).

*Sequencing* allows you to explicitly evaluate one expression before the other; an expression of the form \((e_1; e_2)\) will result in the evaluation of \(e_1\) before \(e_2\). For example:

\[
\begin{align*}
\text{let } x &= \text{ref } 0 \text{ in } (x := 1; x := 2; !x + 2) \Downarrow 4 \\
\text{let } x &= \text{ref } 0 \\
\text{let } \text{store_and_return } y &= (x := y; y) \\
\text{store_and_return } 1 \Downarrow 1 \\
!x \Downarrow 1
\end{align*}
\]

Note that the order of evaluation for compound expressions becomes very important. For example, the order in which operands of an addition expression are evaluated will affect the outcome of the following expression:

\[
\begin{align*}
\text{let } x &= \text{ref } 0 \text{ in } (x := 1; 1) + (x := 2; 1) + !x
\end{align*}
\]

does it evaluate to 2, or to 4?

**Topic 4. Using mutation.** Some languages are purely functional and contain no mutability. Some functional purists maintain that state is unnecessary! Indeed, it is easier to reason, via induction, about purely functional programs, and so correctness is easier to prove. However, the presence of state is arguably very natural and useful in many situations, e.g. designing GUIs, matrices, etc. For the latter, arrays are clearly a useful language feature:

arrays:

\[
\begin{align*}
\text{Array.make } 10 \text{ “hello” creates array with } 10 \text{ “hello” elements} \\
\text{let } ar &= \left[|2;4;6|\right] \\
ar : \text{int array} \\
ar.(2) &< 10 \\
ar \Downarrow \left[|2; 4; 10|\right] \text{ indexing begins at 0} \\
ar.(2) \Downarrow 10
\end{align*}
\]

In discussions about Homework 1, we have also discussed possible efficiency improvements In homework 1 we have also seen the example of improved efficiency via *memoization*, which can be implemented using arrays:

\[
\begin{align*}
\text{let rec fibnum_mem } n = \\
\quad \text{let tab } = \text{Array.make } (n+1) \text{ 0 in} \\
\quad \text{let rec fm } n = \text{match } n \text{ with} \\
\quad \quad 1 \rightarrow 0 \\
\quad \quad 2 \rightarrow 1 \\
\quad \quad n \rightarrow \text{if } \text{tab}.(n) > 0 \text{ then } \text{tab}.(n) \text{ else} \\
\quad \quad \quad \text{let fn } = \text{fm } (n-1) + \text{fm } (n-2) \text{ in} \\
\quad \quad \quad (\text{tab}.(n) <\text{- fn}; \text{fn}) \\
\quad \text{in } \text{fm } n
\end{align*}
\]