NOTES 11: NANOCAML SEMANTICS

Cameron Moy

Thursday July 5th

1 Semantics, onward!

We are implementing a stripped down version of OCaml called NanOCaml. Our lexer and parser will produce for us an AST. Recall the grammar from last time.

$$T \to (T\ T) \ \mid\ (\texttt{fun}\ X\ \texttt{->}\ T) \ \mid\ X$$

$$X \to a \ \mid\ b \mid\ \dots\ \mid\ z\ \mid aa\ \mid\ \dots$$

Now, what should one do with this? The semantics tell us. In this course we will use a (big-step) operational semantics. These are sets of rules that define a partial function from the terms in our language T to a set of resulting values V.

We're going to look at two different possible semantics for NanOCaml. The first corresponds to our intuitive notion of function application. The other is how OCaml actually implements functions. It turns out these semantics define the same evaluation function, but their implementations and performance are significantly different in practice.

2 The substitution model

In this model, when we apply an argument to a function we substitute the argument in for occurrences of the parameter in the function body. This substitution operation is denoted $t[x \mapsto v]$. We also must define our set of values,

$$(1)\,\frac{}{(\texttt{fun}\ x\ \texttt{->}\ t) \Downarrow_s (\texttt{fun}\ x\ \texttt{->}\ t)}$$

$$(2)\,\frac{t_1 \Downarrow_s (\texttt{fun}\ x\ \texttt{->}\ t_{12}) \qquad t_2 \Downarrow_s v_2 \qquad t_{12}[x \mapsto v_2] \Downarrow_s v}{(t_1\ t_2) \Downarrow_s v}$$

Figure 1: Substitution semantics for NanOCaml.

the co-domain of our evaluation function. Here, the values of our language are just functions.

$$V \to (\texttt{fun}\ X\ \texttt{->}\ T)$$

The semantics in Figure 1 define an evaluation function $\Downarrow_s : T \to V$. Rule (1) states that functions evaluate to themselves. Rule (2) states that applications are evaluated by

1. evaluating the first component to a function,
2. evaluating the second component (argument) to a value,
3. substituting occurrences of the function parameter in the body with the argument.

Example. Prove that ((fun x -> x) (fun y -> y)) evaluates to (fun y -> y) under $\Downarrow_s$.

Proof. We construct a proof tree.

$$\frac{\dfrac{}{(\texttt{fun}\ x\ \texttt{->}\ x) \Downarrow_s (\texttt{fun}\ x\ \texttt{->}\ x)} \qquad \dfrac{}{(\texttt{fun}\ y\ \texttt{->}\ y) \Downarrow_s (\texttt{fun}\ y\ \texttt{->}\ y)} \qquad \dfrac{}{x[x \mapsto (\texttt{fun}\ y\ \texttt{->}\ y)] \Downarrow_s (\texttt{fun}\ y\ \texttt{->}\ y)}}{((\texttt{fun}\ x\ \texttt{->}\ x)\ (\texttt{fun}\ y\ \texttt{->}\ y)) \Downarrow_s (\texttt{fun}\ y\ \texttt{->}\ y)}$$

❑

3 The environment model

While intuitively nice, the substitution model suffers from a number of drawbacks. Namely, it's inefficient. The substitution operation has to traverse the entire abstract syntax tree for every appliation. This could get expensive.

$$(1)\,\frac{}{e\ ;\ (\texttt{fun}\ x\ \texttt{->}\ t) \Downarrow_e \langle e, (\texttt{fun}\ x\ \texttt{->}\ t)\rangle}$$

$$(2)\,\frac{e(x) = v}{e\ ;\ x \Downarrow_e v}$$

$$(3)\,\frac{e\ ;\ t_1 \Downarrow_e \langle e', (\texttt{fun}\ x\ \texttt{->}\ t_{12})\rangle \qquad e\ ;\ t_2 \Downarrow_e v_2 \qquad e'[x \mapsto v_2]\ ;\ t_{12} \Downarrow_e v}{e\ ;\ (t_1\ t_2) \Downarrow_e v}$$

Figure 2: Environment semantics for NanOCaml.

An alternate approach, and the one taken by OCaml, is that of delaying substitution. Instead of a direct replacement we store the binding in an environment and only make a replacement when needed. Application is simply extending this environment with a new binding, indicated by $e[x \mapsto v]$.

Like the substitution model, we have to define our set of values V.

$$V \triangleq E \times F$$

$$E \triangleq X \times V$$

$$F \rightarrow (\texttt{fun}\ X\ \texttt{->}\ T)$$

We no longer have functions as our values, but closures. Closures are pairs of environments E and functions F.

Remark. The $e[x \mapsto v]$ notation looks similar to that of substitution, but has an entirely different meaning. Substitution works directly on the AST and makes a substitution, while extension adds a binding to an environment.

In the semantics defined in Figure 2, we are defining an evaluation function $\Downarrow_e : E \times T \rightarrow V$. Rule (1) states that functions evaluate to a closure containing the environment and body of the function. Rule (2) states that variables evaluate to their bound value according to the environment. Rule (3) states that applications are evaluated by

1. evaluating the first component to a closure,

2. evaluating the second component (argument) to a value,

3. extending the environment (from the closure) by binding the parameter to the argument and evaluating the body of the closure.

Remark. The semantics above implement lexical scope. How would you modify them to implement dynamic scope?

Example. Prove that $((\texttt{fun}\ x\ \texttt{->}\ x)\ (\texttt{fun}\ y\ \texttt{->}\ y))$ evaluates to $\langle \cdot, (\texttt{fun}\ y\ \texttt{->}\ y)\rangle$ under $\Downarrow_e$ in the empty environment $\cdot$.

Proof. We construct a proof tree.

$$\frac{\dfrac{}{\cdot\ ;\ (\texttt{fun}\ x\ \texttt{->}\ x) \Downarrow_e \langle \cdot, (\texttt{fun}\ x\ \texttt{->}\ x)\rangle} \qquad \dfrac{}{\cdot\ ;\ (\texttt{fun}\ y\ \texttt{->}\ y) \Downarrow_e \langle \cdot, (\texttt{fun}\ y\ \texttt{->}\ y)\rangle} \qquad \dfrac{(x : \langle \cdot, (\texttt{fun}\ y\ \texttt{->}\ y)\rangle)(x) = \langle \cdot, (\texttt{fun}\ y\ \texttt{->}\ y)\rangle}{x : \langle \cdot, (\texttt{fun}\ y\ \texttt{->}\ y)\rangle\ ;\ x \Downarrow_s \langle \cdot, (\texttt{fun}\ y\ \texttt{->}\ y)\rangle}}{\cdot\ ;\ ((\texttt{fun}\ x\ \texttt{->}\ x)\ (\texttt{fun}\ y\ \texttt{->}\ y)) \Downarrow_e \langle \cdot, (\texttt{fun}\ y\ \texttt{->}\ y)\rangle}$$

❑

4 Parting thoughts

We've constructed an extremely simple version of OCaml. We have no base types, records, tuples, lets. No syntax to define recursive functions, no pattern matching, no variant types. Think about the following.

1. How powerful is our NanOCaml language?

2. Could you write non-terminating programs in it?

3. What about recursive functions?

4. Would it be possible to encode numbers and then be able to do arithmetic computations?

Would it surprise you if the answer to all of these was yes?