NOTES 10: NANOCAML SEMANTICS

Cameron Moy

Tuesday July 9th

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.

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

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

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. Let's look at a possible semantics for NanOCaml!

2 The environment model

Our semantics will be centered around the idea of substitution. When we apply an argument to a function, we substitute the argument in for occurrences of the parameter in the function body.

How should one implement this substitution operation? One could traverse the entire abstract syntax tree and make the replacement. This could get expensive.

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

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

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

Figure 1: Environment semantics for lexically-scoped 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 $e[x \mapsto v]$.

We have to define our set of values V.

$$V \triangleq E \times F$$
$$E \triangleq X \times V$$
$$F \to (\texttt{fun}\ X\ \texttt{->}\ T)$$

We don't have functions as our values, but closures. Closures are pairs of environments E and functions F.

In the semantics defined in Figure 1, we are defining an partial evaluation function $\Downarrow_l : E \times T \rightharpoonup V$. 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 closure's environment by binding the parameter to the argument and evaluating the body of the closure.

$$(1)\ \frac{}{e\ ;\ (\texttt{fun}\ x\ \texttt{->}\ t) \Downarrow_d \boxed{}}$$

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

$$(3)\ \frac{e\ ;\ t_1 \Downarrow_d \boxed{} \qquad e\ ;\ t_2 \Downarrow_d v_2 \qquad \boxed{}}{e\ ;\ (t_1\ t_2) \Downarrow_d \boxed{}}$$

Figure 2: Incomplete semantics for dynamically-scoped NanOCaml.

Exercise. The semantics above implement a lexically-scoped NanOCaml. A semantics for dynamic scope would define $\Downarrow_d : E \times T \rightharpoonup V$, a partial evaluation function where instead of closures as V, we simply let $V \triangleq F$. Fill in the blanks in Figure 2 to create a semantics that implements dynamic scope.

3 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?