
CMSC 22100/32100: Programming Languages

Homework 6

M. Blume

Due: November 11, 2008

Recall the definition of MinML from the textbook and recall the notation we used in Homework Set 5 (which was due on November 4, 2008).

1. Consider removing the expression form **fun** $f(x : \tau_1) : \tau_2$ **is** e from the language. In its place we add two other constructs:
 - ordinary (non-recursive) λ -forms $\lambda x : \tau. e$ whose static and dynamic semantics are identical to those given in the lecture notes when discussing the simply typed λ -calculus, and
 - a special binding form for recursive functions called **letrec**:

letrec $f : \tau_1 \rightarrow \tau_2 = \lambda x. e$ **in** e_2

The idea is to evaluate e_2 in the scope of a binding for the recursive function f whose type is $\tau_1 \rightarrow \tau_2$, whose formal argument is x , and whose body is e . Within e there may be free references x (for referring to the argument) and to f (for referring to function f recursively).

Your task is to:

- (a) Show the typing rule for this **letrec** form.
 - (b) Show the cases of the definition of substitution that deal with **letrec**.
 - (c) Give a rule for evaluating a **letrec** expression in a substitution-based big-step semantics. Discuss your design—especially why and how it is trickier than the rule for the **fun** expression of the original MinML.
Hint: How can you encode (or “macro-expand”) the original **fun** form in terms of **letrec** and vice versa?
 - (d) Show the corresponding small-step rule for **letrec**.
2. Since the E machine is not substitution-based, your solution to questions 1c and 1d does not work there. A “straightforward” approach to handling the problem seems to call for a “recursive” closure value $V = \langle \lambda x : \tau_1. e_1, E \rangle$ where E binds f to V . However, this does not work because the construction of such a value is not well-founded. We can handle this problem by not trying to “tie the recursive knot” directly at the time when we establish the binding for f . Instead, we postpone this step until f is actually looked up. This approach distinguishes between two different kinds of variable bindings, namely the following:

normal bindings bind variables to machine values (which are defined as shown in the lecture notes), while

recursive bindings bind variables to information about a recursive function binding. The information consists of: (1) the name of the formal parameter (x), (2) the body of the recursive function (e_1), and (3) the environment E that was in effect when the machine started to process the **letrec** expression.

Normal bindings are established at the time a function is applied. Recursive bindings are established at the time a recursive function goes into scope. Looking up a normal binding simply finds a machine value and proceeds directly. Looking up a recursive binding has to *create* a machine value *at the time of lookup*.

Your task is to flesh out this design:

- Define a syntactic category of *bindings* (call it B) such that $E = \text{Var} \mapsto^{\text{fin}} B$. There should be two cases for B corresponding to what has been said above.
 - Show all state transitions of the new machine involving variable binding (*i.e.*, function application and **letrec**).
 - Show all state transitions of the new machine involving variable lookup.
3. Consider MinML with the extension of simple exceptions (as shown in the textbook). The added expression forms are **fail** and **try** e_1 **ow** e_2 .

For this language, design a substitution-based *big-step* semantics. *Hint:* To do so, start by defining a category of “runtime values” where a runtime value can be either an ordinary value or **fail**. A result of **fail** indicates that an exception has been raised and no ordinary value is available.

- (a) Show the rule(s) for **try** e_1 **ow** e_2 .
- (b) Show the rule(s) for function application e_1 e_2 .