Cornell University
Computer Science 6110

Homework 3 DUE: Monday, March 17, 11:59PM

Updates

- None yet.

What to turn in

Turn in the assignment by midnight on CMSX on the due date, including both code and written problems.

Working with a partner

You may have a partner on this assignment. Both partners are expected to work on all parts of the assignment, and to understand the solution.

1. **Dangling References** [40 pts]

   In class we claimed that during evaluation, uML! programs never generate dangling references. Let's prove it. Consider the fragment of uML! consisting of the following expressions and values:

$$\begin{array}{ll} e ::= & n \mid x \mid \mathsf{ref}\ e \mid !e \mid e_1\ :=\ e_2 \mid \mathsf{null} \mid \lambda x.\, e \mid e_0\ e_1 \\ & \mid \mathsf{let}\ x\ =\ e_0\ \mathsf{in}\ e_1 \mid (e_1,\ e_2) \mid \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 \\ v ::= & n \mid (v_1,\ v_2) \mid \lambda x.e\ ^{\text{(closed)}} \mid \mathsf{null} \end{array}$$

   To define the small-step semantics of uML!, we augment the grammar of expressions and values with a set of locations $\ell \in \mathbf{Loc}$.

$$e ::= \ldots \mid \ell \qquad\qquad v ::= \ldots \mid \ell$$

   A *store* σ is a partial map from locations to values (which could be other locations). The small-step semantics of uML! programs was defined in terms of *configurations* $\langle e, \sigma\rangle$, where e is an augmented expression and σ is a store.

   We define *loc*(e) to be the set of locations that occur in the expression e. Thus, for example, $loc((!\ell_2)\ (\lambda x.\, (!\ell_1)+!(\mathsf{ref}\ 4))) = \{\ell_1, \ell_2\}$.

   A uML! *program* is a closed expression not containing any locations. Thus, if e is a program then *loc*$(e) = \emptyset$ and $FV(e) = \emptyset$. However, during evaluation of the program, locations may appear in the current term.

   (a) Consider the following uML! configuration.

$$\langle\ (\lambda x.\, (!\ell_1)\ 2)\ (\mathsf{ref}\ 1)\ \ ,\ \ \{\ell_1 \mapsto \lambda y.\, \mathsf{ref}\ y\}\ \rangle$$

   Show the evaluation of this configuration. For each configuration $\langle e', \sigma'\rangle$ in the evaluation, give *loc*(e').

   (b) Give an inductive definition of the set *loc*(e) of locations occurring in e.

   (c) Prove that if e is a uML! program and $\langle e, \emptyset\rangle \longrightarrow^* \langle e', \sigma\rangle$, then *loc*$(e') \subseteq \mathrm{dom}(\sigma)$. If you use induction, identify the relation you are using in your induction and argue that it is well-founded.

2. **Transactions** [35 pts]

Transactions are a useful mechanism for imperative programming in the presence of failure. When a system fails in the middle of some computation, it can be left in an inconsistent state that is hard to fix. Transactions solve this problem by allowing state to be cleanly rolled back to what it was at the beginning.

Transactions are becoming popular as a way to support concurrency, by cleanly handling inconsistent updates from different threads. However, they can also be used as a way to recover from other failures, such as exceptions. In the exception mechanisms we looked at in class, a handled exception could leave the store in an unexpected state.

Let's extend uML$_!$ with transactions, resulting in a language we'll call uML$_{\mathrm{T!}}$. We add a term $\mathsf{transaction}\ e$ that evaluates e to produce a value. However, if the transaction fails, the side effects of e on the store are erased. We also add a term abort that causes the innermost dynamically enclosing transaction to fail immediately, producing result null.

$$e ::= \ldots \mid \mathsf{transaction}\ e \mid \mathsf{abort}$$

For example, we expect that this program:

$$\begin{array}{l}\textbf{let}\ x = \mathsf{ref}\ 0\ \textbf{in}\ \mathsf{transaction} \\ \quad (\\ \quad\quad \textbf{let}\ z = (x := 1)\ \textbf{in}\ \mathsf{abort} \\ \quad)\end{array}$$

evaluated in the empty store results in the value null and a store in which some location ℓ_x is mapped to 0, because the update to x is erased when the transaction aborts.

Let's start by writing an operational semantics for this new language feature. During the execution of a transaction, we need to save the store so it can be restored at the end. Therefore we introduce another term $\mathsf{save}_\sigma\ e$ which evaluates e but with σ saved in case e aborts.

$$\begin{array}{rl} e ::= \ldots & \mid \mathsf{save}_\sigma\ e \\ E ::= \ldots & \mid \mathsf{save}_\sigma\ E \end{array}$$

Complete the following operational semantics, which relies on a context T to propagate transaction aborts. in rule ABORT (see 2(c)).

$$\frac{}{\langle \mathsf{transaction}\ e, \sigma\rangle \longrightarrow \langle \mathsf{save}_\sigma\ e, \sigma\rangle}\ \textsc{Begin} \qquad \frac{}{\langle \mathsf{save}_\sigma\ T[\mathsf{abort}], \sigma'\rangle \longrightarrow \boxed{2(b)}}\ \textsc{Abort}$$

$$\frac{}{\langle \mathsf{save}_\sigma\ v, \sigma'\rangle \longrightarrow \boxed{2(a)}}\ \textsc{Commit}$$

(a) What is missing in the rules above marked 2(a)?

(b) What is missing in the rule above marked 2(b)?

(c) Describe how the "abort context" T should be constructed. How does it differ from the evaluation context E? (Be clear about how T is defined, but you don't need to write out its whole definition.)

(d) Suppose that we wanted to return a diagnostic value from an abort, writing $\mathsf{abort}\ e$ rather than just abort. How would we update the rule ABORT to achieve this? Give an example program showing that this language design could create problems for strong typing, **and** explain the issue briefly. (Hint: think about a value of e that contains locations in the store.)

Now let's design a semantics by translation from uML$_{T!}$ to uML, using continuations.

The result will be similar to the semantics of exception handling we saw in class. The idea there was that $[\![e]\!]\rho k h$ was the translation of e in naming environment ρ, control context k, and exception handling environment h. For transactions, we will need to add a store as well, so we have the meaning of e as $[\![e]\!]\rho k h \sigma$.

We can construct a similar translation to explain how transactions work. Since there is only one exception (abort), we let h be just a single continuation rather than a map from exception names to continuations. In the translated code, functions expect a control context, an exception-handling context, and a store, in addition to the ordinary parameter x: something like $\lambda k h x \sigma.\, e$. We can think of the translation of a function as having type **Function** = **Cont** $\rightarrow$ **Cont** $\rightarrow$ **Cont**. Continuations k and h expect an argument and a store, something like $\lambda x \sigma.\, e$, so **Cont** = **Value** $\rightarrow$ **Store** $\rightarrow$ **Answer**.

(e) Write a continuation-passing style translation from uML$_{T!}$ to uML. It should include run-time type checking by tagging values and checking the tags using functions such as *CHECK-BOOL*, *CHECK-FUN*, *CHECK-LOC* (for locations), and so on.

(f) Show that with your translation, $[\![\text{transaction abort}]\!]\rho k h \sigma \approx_{\alpha\beta\eta} [\![\text{null}]\!]\rho k h \sigma$.

(g) In the operational semantics, we saw that the ability to roll back the store appeared explicitly in the creation of the new save term that holds a copy of the old store. How is the coexistence of two stores manifested in your translation? Explain briefly. (Hint: Look for an important way that this translation differs from the uML$_!$ translation.)

3. **Axiomatic semantics** [25 pts]

(a) In class, we have seen the Hoare logic rule for while loops:

$$\frac{\{A \wedge b\}\ c\ \{A\}}{\{A\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{A \wedge \neg b\}}$$

Prove or disprove that the following two alternative Hoare logic rules for while loops are sound:

i.
$$\frac{\{A\}\ c\ \{A\}}{\{A\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{A \wedge \neg b\}}$$

ii.
$$\frac{\{A \wedge b\}\ c\ \{A \vee b\}}{\{A\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{A \wedge \neg b\}}$$

(b) Besides while and for, there is a third common form of loops, which is the do–until statement:

$$\textbf{do}\ c\ \textbf{until}\ b$$

The condition b is checked at the end of each iteration, and when b evaluates to **true**, the loop ends. Write a useful and sound Hoare logic rule for the do-until loop.