

CMSC 631, Spring 2009

Homework 5

Due Wednesday, April 8, in class

Problems 1–3 must be done individually.

Problem 4 may be done in pairs.

1. Write down a sequence of reduction steps reducing each of the following terms to normal form. For this problem, reduction is allowed anywhere within a term, including under a λ .

- (a) $(\lambda x.x(xy))(\lambda u.u)$
- (b) $(\lambda xyz.zyx)aa(\lambda pq.q)$
- (c) $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)$

Note: $\lambda xy.e$ is short for $\lambda x.\lambda y.e$. Remember also that the scope of λ extends as far to the right as possible, and that application associates to the left.

2. For each type, construct a simply-typed lambda calculus term (variables, functions, and function application only) whose most general type is that type, or argue that no term has that type. (Hint: You can double-check your answers in OCaml.)

- (a) $\alpha \rightarrow \beta \rightarrow \beta$
- (b) $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \beta \rightarrow \alpha \rightarrow \gamma$
- (c) $\alpha \rightarrow \beta$
- (d) $\alpha \rightarrow \alpha \rightarrow \alpha$

3. Does the simply-typed lambda calculus with integers have a *subject expansion* property, meaning if $A \vdash e : t$ and $e' \rightarrow e$, does $A \vdash e' : t$? Here \rightarrow is reduction under call-by-value semantics. Either prove that subject expansion holds, or give a counterexample showing that it does not hold.
4. Consider the simply typed lambda calculus with references added:

$$\begin{aligned} e &::= x \mid n \mid \lambda x : t.e \mid e e \mid \text{ref } e \mid !e \mid e := e \\ t &::= \text{int} \mid t \rightarrow t \mid t \text{ ref} \end{aligned}$$

Figure 1 gives an operational semantics for this language. In these rules, instead of just having an expression reduce to another expression, we also track a *store* s , which is a (partial) mapping from *locations* ℓ to values. Locations are just our abstraction for pointers. In the S-REF rule, we allocate new memory by picking an ℓ that is not already in the domain of s and then mapping ℓ to v in the store. Then we can apply S-DEREF if we're asked to dereference a value that is in the store. Similarly, S-ASSIGN handles assignment, which in these semantics not only updates the store, but also returns the value of the right-hand side (so we don't have a unit value, unlike OCaml).

$$\begin{array}{c}
\text{S-APP} \\
\frac{}{\langle s, (\lambda x.e) v \rangle \rightarrow \langle s, e[x \mapsto v] \rangle} \\
\\
\text{C-APP-LEFT} \qquad \text{C-APP-RIGHT} \\
\frac{\langle s, e_1 \rangle \rightarrow \langle s', e'_1 \rangle}{\langle s, e_1 e_2 \rangle \rightarrow \langle s', e'_1 e_2 \rangle} \qquad \frac{\langle s, e_2 \rangle \rightarrow \langle s', e'_2 \rangle}{\langle s, v e_2 \rangle \rightarrow \langle s', v e'_2 \rangle} \\
\\
\text{S-REF} \qquad \text{C-REF} \\
\frac{\ell \notin \text{dom}(s)}{\langle s, \text{ref } v \rangle \rightarrow \langle s[\ell \mapsto v], \ell \rangle} \qquad \frac{\langle s, e \rangle \rightarrow \langle s', e' \rangle}{\langle s, \text{ref } e \rangle \rightarrow \langle s', \text{ref } e' \rangle} \\
\\
\text{S-DEREF} \qquad \text{C-DEREF} \\
\frac{\ell \in \text{dom}(s)}{\langle s, !\ell \rangle \rightarrow \langle s, s(\ell) \rangle} \qquad \frac{\langle s, e \rangle \rightarrow \langle s', e' \rangle}{\langle s, !e \rangle \rightarrow \langle s', !(e') \rangle} \\
\\
\text{S-ASSIGN} \qquad \text{C-ASSIGN-LEFT} \qquad \text{C-ASSIGN-RIGHT} \\
\frac{\ell \in \text{dom}(s)}{\langle s, \ell := v \rangle \rightarrow \langle s[\ell \mapsto v], v \rangle} \qquad \frac{\langle s, e_1 \rangle \rightarrow \langle s', e'_1 \rangle}{\langle s, e_1 := e_2 \rangle \rightarrow \langle s', e'_1 := e_2 \rangle} \qquad \frac{\langle s, e_2 \rangle \rightarrow \langle s', e'_2 \rangle}{\langle s, \ell := e_2 \rangle \rightarrow \langle s', \ell := e'_2 \rangle}
\end{array}$$

Figure 1: (Small-step) Operational semantics with stores

$$\begin{array}{c}
\text{T-VAR} \qquad \text{T-INT} \qquad \text{T-LAM} \qquad \text{T-APP} \\
\frac{x \in \text{dom}(A)}{A \vdash x : A(x)} \qquad \frac{}{A \vdash n : \text{int}} \qquad \frac{A, x : t \vdash e : t'}{A \vdash \lambda x : t.e : t \rightarrow t'} \qquad \frac{A \vdash e_1 : t \rightarrow t' \quad A \vdash e_2 : t}{A \vdash e_1 e_2 : t'} \\
\\
\text{T-REF} \qquad \text{T-DEREF} \qquad \text{T-ASSIGN} \\
\frac{A \vdash e : t}{A \vdash \text{ref } e : t \text{ ref}} \qquad \frac{A \vdash e : t \text{ ref}}{A \vdash !e : t} \qquad \frac{A \vdash e_1 : t \text{ ref} \quad A \vdash e_2 : t}{A \vdash e_1 := e_2 : t}
\end{array}$$

Figure 2: Type rules with references

Next, we need to extend our expression grammar with locations and also define values v :

$$\begin{aligned} e &::= \dots \mid \ell \\ v &::= n \mid \lambda x : t. e \mid \ell \end{aligned}$$

i.e., a value is either an integer, function, or pointer.

Figure 2 gives the type system for this language, which matches the one we discussed in class.

Your task is to prove progress and preservation for this language. If we want to do this, we are immediately faced with a question: How do we assign types to pointers? For example, S-REF produces a result containing the value ℓ , but we have no type rules that would assign ℓ a type; hence proving preservation would be difficult. To solve this problem, we're going to use a standard trick: We'll extend type environments A so that they also map locations to types. Specifically, A will have the form

$$A ::= \emptyset \mid (A, x : t) \mid (A, \ell : t)$$

Then we just add a rule

$$\frac{\text{T-Loc} \quad \ell \in \text{dom}(A)}{A \vdash \ell : A(\ell)}$$

But now that we're giving types to ℓ in A , we need to be able to talk about whether A gives a valid type to ℓ . For example, suppose we had $A = \ell : \text{int ref}$, but we had a store s such that $s(\ell) = \lambda x. \dots$. Then clearly the type of ℓ in A (a pointer to an integer) does not match what ℓ actually points to in s (a function). In a well-typed program this kind of difference should never happen, and we can make that explicit with a definition:

Definition 1 (Compatibility) $A \sim s$ (pronounced “Type environment A is compatible with store s ”) if:

- $\text{dom}(A) = \text{dom}(s)$, i.e., they talk about the same locations
- $\forall \ell \in \text{dom}(s)$ there exists a t such that $A(\ell) = t \text{ ref}$ and $A \vdash s(\ell) : t$. In other words, the type that ℓ points to in A matches the type of the value stored at location ℓ in s .

The first part of the above definition may seem overly strong, but it will work out. Next, for convenience, we can put the above definition together with standard typing:

Definition 2 (Configuration typing) $A \vdash \langle s, e \rangle : t$ if $A \vdash e : t$ and $A \sim s$.

Now, prove the following two lemmas.

Lemma 1 (Progress) If $A \vdash \langle s, e \rangle : t$ and e is a closed term (no free variables, though it may contain locations ℓ) then either e is a value or there exist s' and e' such that $\langle s, e \rangle \rightarrow \langle s', e' \rangle$.

Lemma 2 (Preservation) If $A \vdash \langle s, e \rangle : t$ and $\langle s, e \rangle \rightarrow \langle s', e' \rangle$ then there exists A' such that $A' \vdash \langle s', e' \rangle : t$ and $A'|_{\text{dom}(A)} = A$.

In the above lemma, $A'|_{\text{dom}(A)}$ means the mapping A' with its domain restricted to the domain of A .