1 References in C

Last time we considered references and their semantics via translation and SOS. Let's take a closer look at what happens in a language like C.

1.1 Syntax

$$e \quad ::= \quad \cdots \mathsf{UML} \cdots \quad | \quad e_1 = e_2 \quad | \quad *e \quad | \quad \&e \quad | \quad e_1; e_2$$

where we think of the constructs respectively as:

- $e_1 = e_2$ is assignment where e_1 is an assignable variable name, for instance, an array reference.
- $*e$ is used to dereference a pointer e (acts sort of like ! in UML$_!$).
- $\&e$ gives a pointer to the location of e.

1.2 Semantics via Translation

The target language is uML. The translation we define will take into account the fact that C has two kinds of values

- L-values : values which can appear in the LHS of assignment; and
- R-values: ordinary values (i.e., those that can appear in RHS of assignment).

To capture this in the target language, for each expression e of C, we define $\mathcal{L}[\![e]\!]\rho\sigma$ (the left value of e) *and* $\mathcal{R}[\![e]\!]\rho\sigma$ (the right value of e). As before, we need to keep track of the state of the computation so the results of the translations are pairs. To summarize:

- $\mathcal{L}[\![e]\!]\rho\sigma =$ (location of e, σ')
- $\mathcal{R}[\![e]\!]\rho\sigma =$ (value of e, σ').

The translation is defined inductively on the structure of C programs. Note that (unlike FORTRAN), some left value translations are not defined. For instance $\mathcal{L}[\![n]\!]\rho\sigma$ is not defined. This prevents reassigning of an integer 5 to have the value 3.

$$\begin{array}{lcll} \mathcal{R}[\![n]\!]\rho\sigma & = & (n, \sigma) & \\ \mathcal{L}[\![x]\!]\rho\sigma & = & (\rho\text{“}x\text{”}, \sigma) & \\ \mathcal{R}[\![e]\!]\rho\sigma & = & \mathsf{let}\,(\ell, \sigma') = \mathcal{L}[\![e]\!]\rho\sigma\ \mathsf{in}\ (\mathsf{LOOKUP}\ \sigma'\ell, \sigma') & \text{if } \mathcal{L}[\![e]\!]\rho \text{ is defined} \end{array}$$

For assignment:

$$\begin{array}{lcl} \mathcal{R}[\![e_1 = e_2]\!]\rho\sigma & = & \mathsf{let}(\ell, \sigma') = \mathcal{L}[\![e_1]\!]\rho\sigma\ \mathsf{in} \\ & & \mathsf{let}(v, \sigma'') = \mathcal{R}[\![e_2]\!]\rho\sigma'\ \mathsf{in}\ (v, \mathsf{UPDATE}\ \sigma''\ell v) \end{array}$$

Comments:

- Force left to right evaluation.
- Allowed to "assign down the chain." For example, $e_1 = e_2 = e_3 = e_4 = 5$ will assign 5 to each e_i in our translation, just as the piece of code e1 = e2 = e3 = e4 = 5; will in C.
- An assignment statement is not an lvalue in C. Hence $(x = 1) = 3$ is not allowed in our language, and (x = 1) = 3; is not in C.

Addressing:

$$\begin{aligned} \mathcal{L}[\![*e]\!]\rho\sigma &= \mathcal{R}[\![e]\!]\rho\sigma \\ \mathcal{R}[\![\&e]\!]\rho\sigma &= \mathcal{L}[\![e]\!]\rho\sigma \end{aligned}$$

So the two operations, $*, \&$, are inverses of each other. The semantics defined above let us write, for example $*\&*\&*\&x = 2$. This will have the same effect as $x = 2$. Another example (written in C):

```
int y = 0;
int x = &y;
*x = 1;
```

This piece of code assigns 1 to y. Note that it would be illegal to include a line &y = 0x1002; because we can't change the address of a variable.

Functions (ignoring the fact that λ is not actually part of C):

$$\begin{aligned} \mathcal{R}[\![\lambda x.e]\!]\rho\sigma &= (\lambda y \sigma_{\text{dyn}}.\, \mathsf{let}\,(\ell, \sigma') = \mathsf{MALLOC}\,\sigma_{\text{dyn}}\, y\ \mathsf{in}\ \mathcal{R}[\![e]\!](\mathsf{EXTEND}\ \rho\ \text{“}x\text{”}\ \ell)\sigma', \sigma) \\ \mathcal{R}[\![e_1\ e_2]\!]\rho\sigma &= \mathsf{let}\,(f, \sigma') = \mathcal{R}[\![e_1]\!]\rho\sigma\ \mathsf{in} \\ &\quad\ \mathsf{let}\,(v, \sigma'') = \mathcal{R}[\![e_2]\!]\rho\sigma'\ \mathsf{in}\ f v \sigma'' \end{aligned}$$

Discussion:

- Choices we made: function applications don't affect state, in function application use left to right evaluation.
- In defining function application, the allocation is required because variables must be bound to locations, whereas the argument of the function is a value (c **is call by value**).
- The value that a function returns is an rvalue rather than an lvalue. Why? Returning an lvalue whould mean we could assign new values to locations allocated during computation, e.g. fibo 5 = 1;. We choose a semantics where we get back the value of a computation rather than its location.
- Note that we do not need to explicitly define the store in the translation of function application because $fv\sigma''$ returns a pair of the location and the store.

2 Pass (Call) by Reference

It is conceivable to define a language where arguments to functions are lvalues. In this case, the body of functions can update the argument and therefore change the value stored in the location passed in. Examples of such a language are PASCAL, MODULA, Ada, Matlab (sort of), FORTRAN.
Consider a simplified version:

$$e \quad ::= \quad \cdots \ \mid\ \lambda_r x.\, e \ \mid\ \mathsf{rcall}\ e_1\ e_2$$

Note that we are explicitly saying when we pass by reference since we do not have a type system which would take care of distinguishing between the two cases.

Example program:

$$\begin{array}{ll} \mathsf{let} & x = 1 \ \mathsf{in} \\ \mathsf{let} & f = \lambda_r y.\, y = y + 1 \ \mathsf{in} \\ & \mathsf{rcall}\ f x \end{array}$$

In this program y is an alias for x, so the value at the location of x is updated to 2.

To incorporate call by reference into the semantics, we just need to change the rules for functions.

$$\begin{array}{rcl} \mathcal{R}[\![\lambda_r x.e]\!]\rho\sigma & = & (\lambda y \sigma_{\mathrm{dyn}}.\mathcal{R}[\![e]\!](\mathsf{EXTEND}\ \rho\ \text{“}x\text{”}\ y)\sigma_{\mathrm{dyn}}, \sigma) \\ \mathcal{R}[\![\mathsf{rcall}\ e_1\ e_2]\!]\rho\sigma & = & \mathsf{let}\,(f, \sigma') = \mathcal{R}[\![e_1]\!]\rho\sigma\ \mathsf{in} \\ & & \mathsf{let}\,(\ell, \sigma'') = \mathcal{L}[\![e_2]\!]\rho\sigma'\ \mathsf{in}\ f\ell\sigma'' \end{array}$$

Since we are given a location, we don't need to allocate new memory.

3 Objects

3.1 Definition

Objects combine naming and the notion of state. We can think of objects as naming environments. For instance, we might have

$$\begin{array}{ll} \mathsf{class} & C\{ \\ & \mathsf{int}\ x; \\ & \mathsf{int}\ y; \\ & \mathsf{int}\ m\,(\mathsf{int}\ z)\{\cdots x \cdots y \cdots \mathsf{this} \cdots z\}; \\ \} & \end{array}$$

where x, y are fields (which are **mutable**) and m is method (**immutable**). Each of the fields and methods may be **public** or **private**, and the class itself might be mentioned using the keyword **this**. This is reminiscent of modules (recall: modules are non-hierarchical scope which introduces names that may or may not be accessible outside the scope).
Let's try to encode objects using module constructs:

$$e \quad ::= \quad \cdots \mathsf{UML} \cdots \quad | \quad (\mathsf{fields}\, x_1 = e_1, \ldots, x_n = e_n;\ \mathsf{methods}\, f_1 = \lambda y_1.e'_1, \ldots, f_m = \lambda y_m.e'_m) \quad | \quad e.x$$

where the first new clause defines the object and the second clause selects components of the object.

3.2 Translation

We will translate objects to a naming environment.

$$\begin{array}{l} [\![(\mathsf{fields}\ \overrightarrow{x = e},\ \mathsf{methods}\ \overrightarrow{f = \lambda y.e'})]\!]\rho\sigma = \\ \quad \mathsf{let}\ \overrightarrow{z = [\![e]\!]\rho\sigma}\ \mathsf{in} \\ \quad\quad \mathsf{Y}(\lambda\rho'.\, \lambda x'. \\ \quad\quad\quad \mathsf{if}\ x' = x_1\ \mathsf{then}\ z_1\ \mathsf{else} \\ \quad\quad\quad \mathsf{if}\ x' = x_2\ \mathsf{then}\ z_2\ \mathsf{else} \ldots \\ \quad\quad\quad \mathsf{if}\ x' = f_1\ \mathsf{then}\ \lambda z'_1.[\![e'_1]\!](\mathsf{EXTEND}\ (\mathsf{EXTEND}\ \rho\ \text{“}y\text{”}\ z'_1)\ \text{“this”}\ \rho') \ldots) \end{array}$$

Note that we used the Y combinator to allow method bodies to talk about the object itself, i.e. the naming environment defined by the object.

3.3 Problem

Objects as recursive records breaks when we try to implement inheritance, because the name "this" is bound in the methods of the super-object to the wrong fixed point.