2013W1-lecture23

November 2, 2013

Contents

1 Question of the Day

Continuing our big sequences of questions about state!

2 Logistics

2.1 Midterm Exam: Tue, 5 Nov, 1900–2130, in CHBE 101

Remember: up to 3 textbooks and a 3" 3-ring binder of notes (or equivalent).
Also remember: individual exam followed by group exam!

2.2 Final Project

Information posted soon (hopefully by class time Monday!).

3 Pass-by-Value and Pass-by-Reference

3.1 Pass-by-Value Review

"Pass-by-Value" parameter-passing semantics sounds exciting but really just means that when we initialize a formal parameter, we use roughly the same semantics we use for assignment.

Assignment:

1. Evalute the right-hand-side
2. Get the store location of the identifier on the left-hand-side
3. Update the store location with a copy of the value from the right

Pass-by-value parameter passing:

1. Evaluate the actual parameter expression
2. Set up a store location for the formal parameter identifier
3. Update the store location with a copy of the value from the actual parameter expression

When we call a function with pass-by-value semantics, we might use any expression as the actual parameter expression: a literal number, an identifier, an arithmetic expression, a function application, and so on.

If we do use an identifier, we can be confident that the function we call cannot change the value of the identifier we pass by assigning to the formal parameter.

When we write a function that uses pass-by-value semantics, that means that mutating the formal parameter is fine but not generally very meaningful! For example:

```
// This is a silly function that doesn't accomplish anything!
// Why?  Because of pass-by-value semantics.
void swap(char x, char y) {
  char temp = x;
  x = y;
```

```
  y = temp;
}

int main(int argc, char * argv[]) {
  char a = 'a';
  char b = 'b';
  swap(a, b);
  std::cout << "a: " << a << ", b: " << b << std::endl;
}
```

However, we did note that if our value is a store location—a "memory address", "pointer", or "reference" in C, C++, or Java terms—or contains a store location, copying that value introduces "aliasing". Changes to whatever is **at** that copied store location **do** affect the calling function.

So, this actually works:

```
// Aliasing allows this function to actually accomplish something.
// In order to accomplish it, however, we have to be careful to
// use *x and *y rather than x and y.
void swap(char * x, char * y) {
  char temp = *x;
  *x = *y;
  *y = temp;
}

int main(int argc, char * argv[]) {
  char a = 'a';
  char b = 'b';

  swap(&a, &b); // "&a" is "the store location bound to a"

  std::cout << "a: " << a << ", b: " << b << std::endl;
}
```

There's nothing magical about store locations, however. In particular, as the comment above `swap` indicates, this **still** doesn't work:

```
// This is a silly function that doesn't accomplish anything!
// Why?  Because of pass-by-value semantics.
void swap(char * x, char * y) {
```

```
  char * temp = x;
  x = y;
  y = temp;
}

int main(int argc, char * argv[]) {
  char a = 'a';
  char b = 'b';

  swap(&a, &b); // "&a" is "the store location bound to a"

  std::cout << "a: " << a << ", b: " << b << std::endl;
}
```

3.2 Pass-by-Reference Semantics

Pass-by-Reference semantics is a syntactic sugar for our second version of `swap` above. In particular, with pass-by-reference semantics:

1. Instead of evaluating the actual parameter expression to a value, we "evaluate" it as if it were on the **left** side of an assignment. We get its store location.

2. Instead of creating a new store location for the formal parameter, we bind it to the same store location as the actual parameter.

Notice how in our second version of `swap`, we use the `&` operator to accomplish the first item and then use the `*` operator everywhere we use the formal parameter to accomplish the second item.

3.3 Implementing Pass-by-Reference

Let's implement pass-by-reference in a language where we (if not our programmers!) have direct access to store locations. We're going to start with a language in which surface assignment is desugared in terms of boxes.

So `(set! x 3)` becomes `(set-box! x 3)`.

That means `x` must really refer to a box. When we say `(+ x 1)`, we really mean `(+ (unbox x) 1)`.

How do we declare a variable, then? Consider:

```
(let (x 3)
  ...)
```

We need to create a box (a store location) for `x` in our desugaring. Let's desugar it to:

```
(let (x (box 3))
  ...)
```

The `box` expression **already** does just the right thing: creates a fresh store location!

Finally, pass by **value** semantics would have us desugar something like `(f a)` to `(f (box a))`. Notice that we use `box` to create a new store location for the formal parameter, just like `let` does.

Should pass by **reference** semantics also create a new store location?

Let's go do the implementation..

4 What have we learned today?

- Pass-by-Reference
  - Explain what pass-by-reference semantics means in a language.
  - Implement pass-by-reference semantics in a language (by desugaring **or** direct implementation).
  - Explain what restrictions there are on the actual parameter expression when an actual parameter is passed by reference.
  - Compare and contrast pass-by-value and pass-by-reference, including specifically identifying the one major difference between them.
    - That would be: In pass by reference, directly mutating the formal parameter has an impact on the actual parameter. "Indirect" mutation (i.e., mutation on a store location referred to in the value of the formal parameter) behaves the same for both.

5 Some Practice Exercises

- Implement pass-by-reference semantics directly in the interpreter instead of in the desugarer.
- Create a new `ref-let` construct that creates a "reference variable". The "binding expression" for the `ref-let` must be a syntactic identifier, and

the `ref-let` aliases the new reference variable with this other variable (i.e., it does not create a new store location but reuses the one used by the other variable).

- In C++, pass-by-reference semantics are actually accomplished through a type modifier that creates reference types. Reference types are initialized as with `ref-let`. Yet, it hardly seems worth bothering to have `ref-let`, since it just lets us refer to the same variable by more than one name.
  - Much more than just variables can appear on the left-hand-side of an assignment operator. Expressions like `*x`, `array[3]`, or `object.field` can appear on the left of an assignment. Justify the existence of reference variables (i.e., `ref-let`) given this fact.
  - Second, C++ even allows a function call to appear on the left side of an assignment operator as in `call() = 5;`. What do you think the type of the return value of `call` must be to make this work?