Lecture Notes on First-Class Functions

15-411: Compiler Design
Rob Simmons and Jan Hoffmann

Lecture 25
Nov 29, 2016

1 Introduction

In this lecture, we discuss two generalizations of C0: function pointers and nested, anonymous functions (lambdas). As a language feature, nested functions are a natural extension of function pointers. However, because of the necessity of closures in the implementation of nested functions, the necessary implementation strategies are somewhat different.

2 Function pointers

The C1 language includes a concept of function pointers, which are obtained from a function with the address-of operator `&`f. The dynamic semantics can treat `&`f as a new type of constant, which represents the memory address where the function f is stored.

$$\begin{array}{lcl} S;\eta \vdash (*e)(e_1,e_2) \rhd K & \longrightarrow & S;\eta \vdash e \rhd ((*\_)(e_1,e_2)\ ,\ K) \\ S;\eta \vdash \texttt{\&}f \rhd ((*\_)(e_1,e_2),K) & \longrightarrow & S;\eta \vdash e_1 \rhd (f(\_,e_2)\ ,\ K) \end{array}$$

Again, we only show the special case of evaluation function calls with two and zero arguments. After the second instruction, we continue evaluating the arguments to the function left-to-right and then call the function as in our previous dynamics. We do not have to model function pointers using a heap as we did for arrays and pointers since we are not able to change the functions that is stored at a given address.

It is relatively straightforward to extend a language with function pointers, because they are addresses. We can obtain that address at runtime by referring to the label as a constant. Any label `lab1` in an assembly file represents an address in memory (since the program must be loaded into memory in order to run), and can

be treated as a constant by writing `$labl`. Therefore `f = &foo` in a C1 program will eventually be compiled into an instruction like `MOVQ $_c0_foo %r11`. It's possible to call a function whose address is stored in a register with the assembly instruction like `CALL *%r11`.

The static semantics of function pointers are straightforward, though slightly obfuscated because C1 inherits C's horrendous syntax for function types. The C1 language disallows the common definition-as-use C version of declaring function types, which looks like this:

```
int (*f)(int, int) = &foo;
```

Instead, function types in C1 *must* be declared as type definitions, by writing `typedef` followed by a function declaration:

```
typedef int binop_fn(int x, int y);
typedef int unop_fn(int x);
typedef int binop_fn_2(int x, int y);
```

These declarations create three function types, `binop_fn`, `unop_fn`, and `binop_fn_2`; the address-of operation gives us pointers to those functions.

```
binop_fn* f = &foo;
binop_fn_2* g = &foo;
unop_fn* h = &bar;
```

A further restriction on C's syntax is that we treat function types *nominally* – the types of `f` and `g` are different, and these pointers cannot be compared for equality. This means that `&foo` has a status similar to `NULL`: it can have type `binop_fn*` or type `binop_fn_2*`, but we don't know until we assign it. As was the case for `NULL`, both `(*NULL)(e1,e2)` and `(*&foo)(e1,e2)` are disallowed in C1.

As an aside, the reasons for treating function types nominally is a result of the use of contracts in the C1 language, a topic we've ignored so far in this class. We can attach different contracts to different type definitions by including them as part of the type definition:

```
typedef int binop_fn(int x, int y);
  //@requires x >= y; ensures \result > 0;
typedef int binop_fn_2(int x, int y);
  //@requires x != y;
```

These type definitions mean that even though both `f` and `g` refer to the same function `foo`, C1 will signal a precondition violation if we call `(*g)(3,3)` but not if we call `(*f)(3,3)`.

The type rules for the new syntactic forms are given below.

$$\frac{ft = (\tau_1, \ldots, \tau_n) \to \tau \quad \Gamma(f) = ft}{\Gamma \vdash \texttt{\&}f : ft*}$$

$$\frac{ft = (\tau_1, \ldots, \tau_n) \to \tau \quad \Gamma \vdash e : ft* \quad \Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n}{\Gamma \vdash *e(e_1, \ldots, e_n) : \tau}$$

3 First-class functions

The ability to pass functions around as values is convenient, but it is insufficient for many of the idioms usually associated with functional programming. One such idiom is currying and partial application: the ability to take a function with two arguments, like `int foo(int x, int y) { return x + y; }`, and turn it into a function with one argument by setting the first argument `x`, to be a specific value. In Standard ML, this would look like this:

```
let f = fn (x, y) => x + y
let g = fn x => fn y => f (x, y)
let h = g 7
```

Now g is a function from integers to integers that adds seven to its argument.

Syntactically, we can support these types of functions in our language by adding a new expression form, `fn (int i) { stm }`, which evaluates to a function pointer. With this syntactic form, we can create an analogue to the function g above:

```
unop_fn* addn(int x) {
   int z = x + 1;
   return fn (int y) { return x + z + y; };
}

int main() {
   unop_fn* h1 = addn(7);
   unop_fn* h2 = addn(6);
   return (*h1)(3) + (*h1)(5) + (*h2)(3);
}
```

However, the dynamic semantics of this addition are not entirely straightforward.

In functional programming languages, it is common to present the dynamic semantics in terms of *substitution*. In that case, we can say that the result of calling `addn(7)` is that we evaluate the body of `addn` with 7 substituted for `x` and 8 substituted for z, that is:

```
return fn (int y) { return 7 + 8 + y; }
```

The substitution semantics makes it clear, in the example above, that `h1` should be a pointer to a function that adds 15 to its argument, and `h2` should be a pointer to a function that adds 13 to its argument. Substitution semantics, however, are ill-suited for languages like C0. It's not meaningful to substitute 7 for `x` in a function that includes the assignment `x = x + 1` in a loop. This is not just a problem for C0 and C, but for all "Algol-like" languages, including Java and, to some extent, JavaScript and Python. If you take the Foundations of Programming Languages class at Carnegie Mellon, you can learn more about this.

Let's back up. At the point in the `main` function where we *call* the function `addn` through the function pointer `h1` or `h2`, we have access to the expected value of the argument `y`, but in order to evaluate the function, we also need to know the value of `x`. There's no way this information can be available at compile time: it is different depending on whether we are calling `addn` through the function pointer `h1` (where `x` is 7) or `h2` (where `x` is 6). Therefore, we need to be able to access, at runtime, both the function's code itself (the function pointer) and the stored information about what the appropriate value of the variables `x` and `z` is. This combination of the function pointer and the values of the captured variables is called a *closure*. When we evaluate a function expression, we capture the current environment into the closure, and when we call the function, we use the stored environment for the function call.

$$
\begin{array}{lcl}
S;\eta \vdash \texttt{fn}(x,y)\{s\} \rhd K & \longrightarrow & S;\eta \vdash \langle\!\langle \texttt{fn}(x,y)\{s\}, \eta \rangle\!\rangle \rhd K \\
S;\eta \vdash \langle\!\langle \texttt{fn}(x,y)\{s\}, \eta' \rangle\!\rangle \rhd ((*\_)(e_1,e_2), K) & \longrightarrow & S;\eta \vdash e_1 \rhd ((*\langle\!\langle \texttt{fn}(x,y)\{s\}, \eta' \rangle\!\rangle)(\_,e_2)\ , K) \\
S;\eta \vdash v_1 \rhd ((*\langle\!\langle \texttt{fn}(x,y)\{s\}, \eta' \rangle\!\rangle)(\_,e_2)\ , K) & \longrightarrow & S;\eta \vdash e_2 \rhd ((*\langle\!\langle \texttt{fn}(x,y)\{s\}, \eta' \rangle\!\rangle)(v_1,\_)\ , K) \\
S;\eta \vdash v_2 \rhd ((*\langle\!\langle \texttt{fn}(x,y)\{s\}, \eta' \rangle\!\rangle)(v_1,\_)\ , K) & \longrightarrow & S;\langle \eta, K\rangle; [\eta', x \mapsto v_1, y \mapsto v_2] \vdash s \blacktriangleright \cdot
\end{array}
$$

This is only a particular implementation of closures, however, and it differs in important ways from the way Python and JavaScript handle closures. Here, we capture the values of variables in the closures while Python stores a reference to the content of the variable. To understand the difference, consider what the semantics above says about the result value of this computation, and then compare that result to the comparable Python program:

```
unop_fn* addn(int x) {
   unop_fn* f = fn (int y) { x++; return x + y; };
   x++;
   return f;
}

int main() {
   unop_fn* h1 = addn(7);
   unop_fn* h2 = addn(6);
   return (*h1)(3) + (*h1)(5) + (*h2)(3);
}
```

One of the problems with nested functions in imperative programming languages

is that it necessitates knowing a lot about the details of closures. Even a language like ML is using closures at runtime – there's no actual substitution happening – but it's not necessary to know the details like it is when doing programming in Python or in this variant of C0. In lecture, we discussed the possibility of using the static semantics to disallow modifying temps that will be captured inside of closures, which is one way of avoiding these sorts of ambiguities.

We can implement the semantics we described above by rewriting the program. Each anonymous function can be turned into a top-level function that takes one extra argument: the struct containing the extra initialization data needed to run . We can't quite rewrite the program into a valid C0 program with function pointers. A closure for a `unop_fn*` may need:

- no extra data, as in `fn (int y) { return y + 3; }`
- only one piece of extra data, as in `fn (int y) { return x + y; }`
- multiple pieces of extra data, as in `fn (int y) { return (*f)(x,z); }`

In every case, when we evaluate the function to a value, we know at compile-time what data needs to be associated with the pointer.

Using a union type, we can describe appropriate C code for compiling the three examples above as follows:

```
typedef int unop_fn(struct unop_closure* clo, int y);
typedef int binop_fn(struct binop_closure* clo, int x, int z);

struct unop_closure {
   unop_fn* f;
   union unop_fn {
     struct {} clo1;
     struct { int x; } clo2;
     struct { struct binop_closure* f; int x; int z; } clo3;
   } data;
};

int fn1 (struct unop_closure* clo, int y) {
   return y + 3;
}

int fn2 (struct unop_closure* clo, int y) {
   int x = clo->data.clo2.x;
   return y + 3;
}

int fn3 (struct unop_closure* clo, int y) {
   struct binop_closure* f = clo->data.clo3.f;
   int x = clo->data.clo3.x;
   int z = clo->data.clo3.z;
   return (*f->f)(f, x, z);
}
```

As we can see in the last example, introducing closures means that we need to treat the runtime value of *every* function as a closure. We would additionally replace any the expression `fn (int y) { return x + y; }` in the program with an allocation of a `struct unop_closure`, storing `&fn2` in the first field `clo->f` and storing `x`, as defined in the current environment in `clo->data.clo2.x`.