15-411: First-Class Functions

Jan Hoffmann
C1 is a conservative extension of C0

- (A limited form of) function pointers
- Break and continue statements
- Generic pointers (void*)
- More details in the C0 language specification
Function Pointers
Function Pointers in C

• In C we can use the address of operator & to get the address of a function

• However, we cannot modify the content of a function’s address

• Function types are defined using typedef

Example:

```c
typedef int optype(int, int);

typedef int (*optype_pt)(int, int);
```
Function Pointers in C

• In C we can use the address of operator & to get the address of a function.

• However, we cannot modify the content of a function’s address.

• Function types are defined using typedef.

Example:

```c
typedef int otype(int,int);

typedef int (*optype_pt)(int,int);
```

Not in C1!
Function Pointers in C: Examples

```c
int f (int x, int y) {
    return x+y;
}

int (*g)(int x, int y) = &f;

int main () {
    (*g)(1,2);
}
```

```c
int f (int x, int y) {
    int g (int y) {return 0};
    return x+y;
}
```
Function Pointers in C: Examples

```c
int f (int x, int y) {
    return x+y;
}
int g (int y) {return 0};
int main () {
    int (*g)(int x, int y) = &f;
    (*g)(1,2);
}
```

```c
int f (int x, int y) {
    int g (int y) {return 0};
    return x+y;
}
```
Function Pointers in C: Examples

int f (int x, int y) {
    return x+y;
}

int (*g)(int x, int y) = &f;

int main () {
    (*g)(1,2);
}

Cannot define local functions:

int f (int x, int y) {
    int g (int y) {return 0};
    return x+y;
}
Function Pointers in C: Examples

typedef int optype(int,int);

int add (int x, int y) {return x+y;}

int mul (int x, int y) {return x*y;}

optype* f1 (int x) {
    optype* g;
    if (x)
        {g = &add;}
    else
        {g = &mul;}
    return g;
}

int g1 (optype* f, int x, int y) {
    return (*f)(x,y);
}
Function Pointers in C: Examples

typedef int otype(int, int);

int h () {
    otype f2;
    int x = f2(1, 2);
    return 0;
}

Function Pointers in C: Examples

typedef int optype(int,int);

int h () {
    optype f2;
    int x = f2(1,2);
    return 0;
}

In C, ‘variables’ can have a function type.
typedef int optype(int,int);

int h () {
    optype f2;
    int x = f2(1,2);
    return 0;
}

In C, ‘variables’ can have a function type.

What happens if you compile the program?
Function Pointers in C: Examples

```c
typedef int optype(int,int);
int h () {
    optype f2;
    int x = f2(1,2);
    return 0;
}
```

Local function declaration.

In C, ‘variables’ can have a function type.

What happens if you compile the program?
typedef int optype(int,int);

int h () {
    optype f2;
    int x = f2(1,2);
    return 0;
}

int f2 (int x, int y) {return x+y;}

Function Pointers in C: Examples
Function Pointers in C: Examples

typedef int o responseType(int,int);

int h () {
    responseType f2;
    int x = f2(1,2);
    return 0;
}

int f2 (int x, int y) {return x+y;}

What happens if you compile the program?
Function Pointers in C1

gdef ::= ...
    | typedef type ftp (type vid, … , type vid)

type ::= ... | ftp
Function Pointers in C1

\[
gdef ::= \ldots \quad | \text{typedef type ftp (type vid, \ldots, type vid)}
\]

\[
type ::= \ldots \quad | \text{ftp}
\]

Functional types with different names are treated as different types.
Function Pointers in C1

\[
gdef ::= \ldots \mid \text{typedef type ftp (type vid, \ldots , type vid)}
\]

\[
type ::= \ldots \mid \text{ftp}
\]

\[
\text{Functional types with different names are treated as different types.}
\]

\[
unop ::= \ldots \mid \&
\]

\[
exp ::= \ldots \mid (\ast \ exp) (\ exp, \ldots ,exp )
\]
Function Pointers in C1

\[ gdef ::= \ldots \quad \mid \textit{typedef} \ \textit{type} \ ftp \ (\textit{type} \ vid, \ \ldots, \ \textit{type} \ vid) \]

\[ \textit{type} ::= \ldots \mid ftp \]

\[ \textit{unop} ::= \ldots \mid \& \quad \text{Functional types with different names are treated as different types.} \]

\[ \textit{exp} ::= \ldots \mid (\ast \ \textit{exp}) \ (\textit{exp}, \ \ldots, \textit{exp}) \quad \text{Can only be applied to functions.} \]
Function Pointers in C1

\[
gdef ::= \ldots \mid \text{typedef type ftp (type vid, \ldots , type vid)}
\]

\[
type ::= \ldots \mid \text{ftp}
\]

\[
\text{Functional types with different names are treated as different types.}
\]

\[
\text{Can only be applied to functions.}
\]

\[
\text{Dereference only in function application.}
\]

\[
\text{unop ::= \ldots \mid \&}
\]

\[
\text{exp ::= \ldots \mid (\ast \text{exp}) (\text{exp, \ldots ,exp})}
\]
Function Pointers in C1

```
gdef ::= ... |
       typedef type ftp (type vid, ..., type vid)
type ::= ... | ftp
       Functional types with different names are treated as different types.
unop ::= ... | &
       Can only be applied to functions.
exp ::= ... | (* exp) ( exp, ..., exp )
       Dereference only in function application.
```

**Small types:**
- int, bool, t*, t[]

**Large types:**
- struct s, ftp
Function Pointers in C1

```
unop ::= … | &

exp ::= … | (* exp) ( exp, … ,exp )
```

Small types:
- int, bool, t*, t[]

Large types:
- struct s, ftp

Dereference only in function application.

Functional types with different names are treated as different types.

Can only be applied to functions.

No variables, arguments, and return values of large type.
Static Semantics

\[
ft = (\tau_1, \ldots, \tau_n) \rightarrow \tau \quad \Gamma(f) = ft
\]
\[
\Gamma \vdash \& f : ft\ast
\]

\[
ft = (\tau_1, \ldots, \tau_n) \rightarrow \tau \quad \Gamma \vdash e : ft\ast \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n
\]
\[
\Gamma \vdash *e(e_1, \ldots, e_n) : \tau
\]
Dynamic Semantics
Expressions  
\[ e ::= c \mid e_1 \circ e_2 \mid \text{true} \mid \text{false} \mid e_1 \&\& e_2 \mid x \mid f(e_1, e_2) \mid f() \]

Statements  
\[ s ::= \text{nop} \mid \text{seq}(s_1, s_2) \mid \text{assign}(x, e) \mid \text{decl}(x, \tau, s) \mid \text{if}(e, s_1, s_2) \mid \text{while}(e, s) \mid \text{return}(e) \mid \text{assert}(e) \]

Values  
\[ v ::= c \mid \text{true} \mid \text{false} \mid \text{nothing} \]

Environments  
\[ \eta ::= \cdot \mid \eta, x \mapsto c \]

Stacks  
\[ S ::= \cdot \mid S, \langle \eta, K \rangle \]

Cont. frames  
\[ \phi ::= \_ \circ e \mid c \circ \_ \mid \_ \&\& e \mid f(\_, e) \mid f(c, \_) \mid s \mid \text{assign}(x, \_) \mid \text{if}(\_, s_1, s_2) \mid \text{return}(\_) \mid \text{assert}(\_) \]

Continuations  
\[ K ::= \cdot \mid \phi, K \]

Exceptions  
\[ E ::= \text{arith} \mid \text{abort} \mid \text{mem} \]

Reminder
Expressions

\[ e ::= c | e_1 \odot e_2 | \text{true} | \text{false} | e_1 \&\& e_2 | x | f(e_1, e_2) | f() | \& f | (*e)(e_1, e_2) | (*e)() \]

Statements

\[ s ::= \text{nop} | \text{seq}(s_1, s_2) | \text{assign}(x, e) | \text{decl}(x, \tau, s) | \text{if}(e, s_1, s_2) | \text{while}(e, s) | \text{return}(e) | \text{assert}(e) \]

Values

\[ v ::= c | \text{true} | \text{false} | \text{nothing} \]

Environments

\[ \eta ::= \cdot | \eta, x \mapsto c \]

Stacks

\[ S ::= \cdot | S, \langle \eta, K \rangle \]

Cont. frames

\[ \phi ::= _\odot e | c \odot _ | _ \&\& e | f(_, e) | f(c, _) | s | \text{assign}(x, _) | \text{if}(_, s_1, s_2) | \text{return}(_) | \text{assert}(_) \]

Continuations

\[ K ::= \cdot | \phi, K \]

Exceptions

\[ E ::= \text{arith} | \text{abort} | \text{mem} \]

Reminder
Expressions

\[ e ::= c | e_1 \odot e_2 | \text{true} | \text{false} | e_1 \&\& e_2 | x | f(e_1, e_2) | f() | \& f | (\ast e)(e_1, e_2) | (\ast e)() \]

Statements

\[ s ::= \text{nop} | \text{seq}(s_1, s_2) | \text{assign}(x, e) | \text{decl}(x, \tau, s) | \text{if}(e, s_1, s_2) | \text{while}(e, s) | \text{return}(e) | \text{assert}(e) \]

Values

\[ v ::= c | \text{true} | \text{false} | \text{nothing} | \& f \]

Environments

\[ \eta ::= \cdot | \eta, x \mapsto c \]

Stacks

\[ S ::= \cdot | S, \langle \eta, K \rangle \]

Continuation frames

\[ \phi ::= _\odot e | c \odot _ | _ \&\& e | f(_, e) | f(c, _) | s | \text{assign}(x, _) | \text{if}(_, s_1, s_2) | \text{return}(_) | \text{assert}(_) \]

Continuations

\[ K ::= \cdot | \phi , K \]

Exceptions

\[ E ::= \text{arith} | \text{abort} | \text{mem} \]

Reminder
We use state according to the dynamic semantics, then we know that we need to change either or static or dynamic semantics.

In a course like 15-312, we would learn how to prove these sorts of theorems, but just stating the theorem is still useful as a specification. If we can find a counterexample, a program that passes the static semantics and yet gets stuck in a non-final state or else a potentially effectful operation.

Example

Theorem 1 (No undefined behavior)

If a program passes all the static semantics, and is not-stuck because there exists a state $s$, that is a final state or else!

Counterexample

Sequence:

<table>
<thead>
<tr>
<th>$0$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
<th>$5$</th>
<th>$6$</th>
<th>$7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
<td>$d$</td>
<td>$e$</td>
<td>$f$</td>
<td>$g$</td>
<td>$h$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
</tr>
</tbody>
</table>

$ST$ stands for either a pure operation $e$ or a potentially effectful operation $e$.

$E$ ::= arith | abort | mem

Reminder
\[ S ; \eta \vdash e_1 \odot e_2 \triangleright K \quad \rightarrow \quad S ; \eta \vdash e_1 \triangleright (\_ \odot e_2 , K) \]
\[ S ; \eta \vdash c_1 \triangleright (\_ \odot e_2 , K) \quad \rightarrow \quad S ; \eta \vdash e_2 \triangleright (c_1 \odot \_, K) \]
\[ S ; \eta \vdash c_2 \triangleright (c_1 \odot \_, K) \quad \rightarrow \quad S ; \eta \vdash c \triangleright K \quad (c = c_1 \odot c_2) \]
\[ S ; \eta \vdash c_2 \triangleright (c_1 \odot \_, K) \quad \rightarrow \quad \text{exception(arith)} \quad (c_1 \odot c_2 \text{ undefined}) \]
\[ S ; \eta \vdash e_1 \&\& e_2 \triangleright K \quad \rightarrow \quad S ; \eta \vdash e_1 \triangleright (\_ \&\& e_2 , K) \]
\[ S ; \eta \vdash \text{false} \triangleright (\_ \&\& e_2 , K) \quad \rightarrow \quad S ; \eta \vdash \text{false} \triangleright K \]
\[ S ; \eta \vdash \text{true} \triangleright (\_ \&\& e_2 , K) \quad \rightarrow \quad S ; \eta \vdash e_2 \triangleright K \]
\[ S ; \eta \vdash x \triangleright K \quad \rightarrow \quad S ; \eta \vdash \eta(x) \triangleright K \]
\[ S ; \eta \vdash \text{nop} \rightarrow (s, K) \quad \rightarrow \quad S ; \eta \vdash s \rightarrow K \]
\[ S ; \eta \vdash \text{assign}(x, e) \rightarrow K \quad \rightarrow \quad S ; \eta \vdash e \triangleright (\text{assign}(x, \_), K) \]
\[ S ; \eta \vdash c \triangleright (\text{assign}(x, \_), K) \quad \rightarrow \quad S ; \eta[x \mapsto c] \vdash \text{nop} \triangleright K \]
\[ S ; \eta \vdash \text{decl}(x, \tau, s) \rightarrow K \quad \rightarrow \quad S ; \eta[x \mapsto \text{nothing}] \vdash s \rightarrow K \]
\[ S ; \eta \vdash \text{assert}(e) \rightarrow K \quad \rightarrow \quad S ; \eta \vdash e \triangleright (\text{assert}(\_), K) \]
\[ S ; \eta \vdash \text{true} \triangleright (\text{assert}(\_), K) \quad \rightarrow \quad S ; \eta \vdash \text{nop} \rightarrow K \]
\[ S ; \eta \vdash \text{false} \triangleright (\text{assert}(\_), K) \quad \rightarrow \quad \text{exception}(\text{abort}) \]
\[ S ; \eta \vdash \text{if}(e, s_1, s_2) \rightarrow K \quad \rightarrow \quad S ; \eta \vdash e \triangleright (\text{if}(\_, s_1, s_2), K) \]
\[ S ; \eta \vdash \text{true} \triangleright (\text{if}(\_, s_1, s_2), K) \quad \rightarrow \quad S ; \eta \vdash s_1 \rightarrow K \]
\[ S ; \eta \vdash \text{false} \triangleright (\text{if}(\_, s_1, s_2), K) \quad \rightarrow \quad S ; \eta \vdash s_2 \rightarrow K \]
\[ S ; \eta \vdash \text{while}(e, s) \rightarrow K \quad \rightarrow \quad S ; \eta \vdash \text{if}(e, \text{seq}(s, \text{while}(e, s)), \text{nop}) \rightarrow K \]

Summary: Statements
\[
S ; \eta \vdash f(e_1, e_2) \triangleright K \\
S ; \eta \vdash c_1 \triangleright (f(\_ , e_2) , K) \\
S ; \eta \vdash c_2 \triangleright (f(c_1, \_), K) \\
\]
\[
S ; \eta \vdash f() \triangleright K \\
(S , \langle \eta, K \rangle) ; \eta \vdash v \triangleright (\text{return}(\_), K) \\
\cdot ; \eta \vdash c \triangleright (\text{return}(\_), K) \\
\]
\[
\rightarrow \\
S ; \eta \vdash e_1 \triangleright (f(\_ , e_2) , K) \\
\rightarrow \\
S ; \eta \vdash e_2 \triangleright (f(c_1, \_), K) \\
\rightarrow (S , \langle \eta, K \rangle) ; [x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \triangleright \cdot \\
\text{(given that } f \text{ is defined as } f(x_1, x_2)\{s\}) \\
\rightarrow \\
(S , \langle \eta, K \rangle) ; \cdot \vdash s \triangleright \cdot \\
\text{(given that } f \text{ is defined as } f()\{s\}) \\
\rightarrow \\
S ; \eta \vdash e \triangleright (\text{return}(\_), K) \\
\rightarrow \\
S ; \eta' \vdash v \triangleright K' \\
\rightarrow \\
\text{value}(c) \\
\]
Dynamic Semantics: Function Pointers

\[
S; \eta \vdash (\ast e)(e_1, e_2) \triangleright K \quad \rightarrow \quad S; \eta \vdash e \triangleright ((\ast _\_)(e_1, e_2) , K)
\]

\[
S; \eta \vdash \& f \triangleright ((\ast _\_)(e_1, e_2), K) \quad \rightarrow \quad S; \eta \vdash e_1 \triangleright (f(\_ , e_2) , K)
\]
Nominal Types

C1 treats function types nominally

```c
typedef int optype1(int,int);
typedef int optype2(int,int);

optype1 and optype2 are different types and pointers of optype1 and optype2 cannot be compared.
```

```c
int add (int x, int y) {return x+y;}

int main {
    optype1* f = &add;
    optype2* f = &add;
    return 0;
}
```
Nominal Types

C1 treats function types nominally

typedef int optype1(int,int);
typedef int optype2(int,int);

optype1 and optype2 are different types and pointers of optype1 and optype2 cannot be compared.

int add (int x, int y) {return x+y;}

int main {
    optype1* f = &add;
    optype2* f = &add;
    return 0;
}

Like null, add can have both types.
C1 treats function types nominally

```c
typedef int optype1(int, int);
typedef int optype2(int, int);
```

optype1 and optype2 are different types and pointers of optype1 and optype2 cannot be compared.

```c
int add (int x, int y) {return x+y;}

int main {
    optype1* f = &add;
    optype2* f = &add;
    return 0;
}
```

Like null, add can have both types.

```c
(*&add)(x, y)
```
Nominal Types

C1 treats function types nominally

```c
typedef int optype1(int, int);
typedef int optype2(int, int);
```

optype1 and optype2 are different types and pointers of optype1 and optype2 cannot be compared.

```c
int add (int x, int y) {return x+y;}
int main {
    optype1* f = &add;
    optype2* f = &add;
    return 0;
}
```

Like null, add can have both types.

Not allowed in C1.

`(*&add)(x, y)`
Nominal Type and Contracts

typedef int binop_fn(int x, int y);
   //@requires x &ge; y; ensures \result &gt; 0;
typedef int binop_fn_2(int x, int y);
   //@requires x != y;

• binop_fn and binop_fn_2 are treated as different types

• The call *f(3,3) can cause a precondition violation

• The call *f2(3,3) might be fine even if f and f2 point to the same function
First-Class Functions
Currying and Partial Application

In ML we can have functions that return functions

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

Currying and Partial Application

In ML we can have functions that return functions

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

In C (C0, C1, ...) we can support this by adding a new syntactic form for anonymous functions

```c
fn (int i) { stm }
```
Currying and Partial Application

In ML we can have functions that return functions:

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

In C (C0, C1, ...) we can support this by adding a new syntactic form for anonymous functions:

```plaintext
fn (int i) { stm }
```

The type of this expression is `( t (int) )*` where `t` is the synthesized return type.
Example

```c
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);
}
```
Dynamic Semantics of Anonymous Functions

Dynamic semantics is not immediately clear

In a functional language we could define the semantics using substitution

```
addn(7)  would lead to

    return fn (int y) { return 7 + 8 + y; }
```
Dynamic Semantics of Anonymous Functions

Dynamic semantics is not immediately clear

In a functional language we could define the semantics using substitution

\[ \text{addn}(7) \] would lead to

\[
\text{return fn (int y) \{ return 7 + 8 + y; \}}
\]

But in an imperative language that does not work

The variable \( x \) might be incremented inside a loop

What would the effect of the substitution be?
First-class functions allow 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`, 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 seven to its argument, and `h2` should be a pointer to a function that adds 6 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=3`, given that `7=3` is not a valid assignment! 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,
First-class functions L19.3

arguments, like

\[ \text{int foo(int x, int y) \{ return x + y; \}} \]

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:

\[
\begin{align*}
\text{let f = fn (x, y) => x + y} \\
\text{let g = fn x => fn y => f (x, y)} \\
\text{let h = g 7}
\end{align*}
\]

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, \(\text{fn (int i) \{ stm \}}\), which evaluates to a function pointer.

With this syntactic form, we can create an analogue to the function \(g\) above:

\[
\begin{align*}
\text{unop_fn* addn(int x) \{ } \\
\text{\quad int z = x + 1; \} \\
\text{\quad return fn (int y) \{ return x + z + y; \}; } \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{int main() \{ } \\
\text{\quad unop_fn* h1 = addn(7); } \\
\text{\quad unop_fn* h2 = addn(6); } \\
\text{\quad return (*h1)(3) + (*h1)(5) + (*h2)(3); } \\
\}
\end{align*}
\]

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 \(\text{addn}(7)\) is that we evaluate the body of \(\text{addn}\) with 7 substituted for \(x\), that is:

\[
\text{return fn (int y) \{ return 7 + 8 + y; \}}
\]

The substitution semantics makes it clear, in the example above, that \(h1\) should be a pointer to a function that adds seven to its argument, and \(h2\) should be a pointer to a function that adds 6 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 \(x=3\), given that \(7=3\) is not a valid assignment!

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 \(\text{main}\) function where we call the function \(\text{addn}\) 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 \(\text{addn}\) 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,
C1 Example: Dynamic Semantics

```c
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);
}
```

Of course, function arguments are not available statically.

When we call addn the values of x and z are available.
First-class functions L19.3

arguments, like

\[
\text{int foo(int x, int y) \{ return x + y; \}}
\]

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:

\[
\begin{align*}
\text{let } & \quad f = \text{fn } (x, y) \Rightarrow x + y \\
\text{let } & \quad g = \text{fn } x \Rightarrow \text{fn } y \Rightarrow f (x, y) \\
\text{let } & \quad h = g 7
\end{align*}
\]

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, \(\text{fn } (\text{int } i) \{ \text{stm} \}\), which evaluates to a function pointer.

With this syntactic form, we can create an analogue to the function \(g\) above:

\[
\text{unop_fn* addn(int x) \{} \\
\text{int } z = x + 1; \\
\text{return } \text{fn } (\text{int } y) \{ \text{return } x + z + y; \}; \\
\text{\} }
\]

\[
\text{int main()} \{ \\
\text{unop_fn* h1 = addn(7);} \\
\text{unop_fn* h2 = addn(6);} \\
\text{return } (*\text{h1})(3) + (*\text{h1})(5) + (*\text{h2})(3); \\
\text{\} }
\]

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 \(\text{addn}(7)\) is that we evaluate the body of \(\text{addn}\) with 7 substituted for \(x\), that is:

\[
\text{return } \text{fn } (\text{int } y) \{ \text{return } 7 + 8 + y; \}
\]

The substitution semantics makes it clear, in the example above, that \(h1\) should be a pointer to a function that adds seven to its argument, and \(h2\) should be a pointer to a function that adds 6 to its argument. Substitution semantics, however, are ill-suited for languages like C. It's not meaningful to substitute 7 for \(x\) in a function that includes the assignment \(x=3\), given that \(7=3\) is not a valid assignment!

This is not just a problem for C 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 \(\text{main}\) function where we call the function \(\text{addn}\) 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 \(\text{addn}\) 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, LECTURE NOTES NOV 17, 2015

Of course, function arguments are not available statically.

Idea: Store variable environment with function code

\(\Rightarrow\) function closure
Function Closures: Dynamic Semantics

For functions with two arguments (other functions are similar)

\[
S; \eta \vdash \text{fn}(x, y)\{s\} \triangleright K \quad \rightarrow \quad S; \eta \vdash \langle \text{fn}(x, y)\{s\}, \eta \rangle \triangleright K
\]

\[
S; \eta \vdash \langle \text{fn}(x, y)\{s\}, \eta' \rangle \triangleright ((\_)(e_1, e_2), K) \quad \rightarrow \quad S; \eta \vdash e_1 \triangleright ((\langle \text{fn}(x, y)\{s\}, \eta' \rangle)(\_, e_2), K)
\]

\[
S; \eta \vdash v_1 \triangleright ((\langle \text{fn}(x, y)\{s\}, \eta' \rangle)(\_, e_2), K) \quad \rightarrow \quad S; \eta \vdash e_2 \triangleright ((\langle \text{fn}(x, y)\{s\}, \eta' \rangle)(v_1, \_), K)
\]

\[
S; \eta \vdash v_2 \triangleright ((\langle \text{fn}(x, y)\{s\}, \eta' \rangle)(v_1, \_), K) \quad \rightarrow \quad S; \langle \eta, K \rangle; [\eta', x \mapsto v_1, y \mapsto v_2] \triangleright s \triangleright .
\]
Function Closures: Dynamic Semantics

For functions with two arguments (other functions are similar)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S; \eta \vdash \text{fn}(x, y){s} \triangleright K$</td>
<td>$S; \eta \vdash \langle \text{fn}(x, y){s}, \eta \rangle \triangleright K$</td>
</tr>
<tr>
<td>$S; \eta \vdash \langle \text{fn}(x, y){s}, \eta' \rangle \triangleright ((_)(e_1, e_2), K)$</td>
<td>$S; \eta \vdash e_1 \triangleright ((\langle \text{fn}(x, y){s}, \eta' \rangle)(_), e_2, K)$</td>
</tr>
<tr>
<td>$S; \eta \vdash v_1 \triangleright ((\langle \text{fn}(x, y){s}, \eta' \rangle)(_), e_2, K)$</td>
<td>$S; \eta \vdash e_2 \triangleright ((\langle \text{fn}(x, y){s}, \eta' \rangle)(v_1, _), K)$</td>
</tr>
<tr>
<td>$S; \eta \vdash v_2 \triangleright ((\langle \text{fn}(x, y){s}, \eta' \rangle)(v_1, _), K)$</td>
<td>$S; \langle \eta, K \rangle; [\eta', x \mapsto v_1, y \mapsto v_2] \vdash s \triangleright$</td>
</tr>
</tbody>
</table>
Function Closures: Dynamic Semantics

For functions with two arguments (other functions are similar)

\[
\begin{align*}
S; \eta \vdash \text{fn}(x, y)\{s\} \triangleright K & \quad \rightarrow \quad S; \eta \vdash \langle\text{fn}(x, y)\{s\}, \eta\rangle \triangleright K \\
S; \eta \vdash \langle\text{fn}(x, y)\{s\}, \eta'\rangle \triangleright (\ast\langle e_1, e_2\rangle, K) & \quad \rightarrow \quad S; \eta \vdash e_1 \triangleright (\ast\langle\text{fn}(x, y)\{s\}, \eta'\rangle)(\_\_\, e_2\,, K) \\
S; \eta \vdash v_1 \triangleright (\ast\langle\text{fn}(x, y)\{s\}, \eta'\rangle)(\_\_\,, e_2\,, K) & \quad \rightarrow \quad S; \eta \vdash e_2 \triangleright (\ast\langle\text{fn}(x, y)\{s\}, \eta'\rangle)(v_1\,, \_\_\,, K) \\
S; \eta \vdash v_2 \triangleright (\ast\langle\text{fn}(x, y)\{s\}, \eta'\rangle)(v_1\,, \_\_\,, K) & \quad \rightarrow \quad S; \langle\eta, K\rangle; [\eta', x \mapsto v_1, y \mapsto v_2] \vdash s \triangleright .
\end{align*}
\]
Another Example

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); 
}
Function Closures in Python

```python
def makeInc(x):
    def inc(y):
        # x = x + 1
        return y + x
    x = x + 1
    return inc

inc5 = makeInc(5)
inc10 = makeInc(10)

inc5(4)
```
def makeInc(x):
    def inc(y):
        # x = x + 1
        return y + x
    x = x + 1
    return inc

inc5 = makeInc(5)
inc10 = makeInc(10)

inc5(4)
Function Closures in Python

```python
def makeInc(x):
    def inc(y):
        # x = x + 1
        return y + x
    x = x + 1
    return inc

inc5 = makeInc(5)
inc10 = makeInc(10)

inc5(4)
```

What’s the return value?

What happens when we add this line?
Implementing Function Closures

Is it be possible to translate programs with function closures to C0?

• Idea: turn local funs. into top-level funs. with additional closure argument

• But: the closure argument is different for each instance

• A closure for 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.

Lecture Notes November 27, 2018
Implementing Function Closures

Is it possible to translate programs with function closures to C0?

• Idea: turn local funs. into top-level funs. with additional closure argument

• But: the closure argument is different for each instance

• A closure for 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); }

Need union types.
Implementing Function Closures

typedef int unop(int y);

union unop_data {
    struct {} clo1;
    struct { int x; } clo2;
    struct { struct binop_closure* f; int x; int z; } clo3;
};

typedef int unop_cl_fn(union unop_data* data, int y);

struct unop_closure {
    unop_cl_fn* f;
    union unop_data* data;
};

typedef int unop_fn(struct unop_closure* clo, int y);
Implementing Function Closures

typedef int unop(int y);

union unop_data {
    struct {} clo1;
    struct { int x; } clo2;
    struct { struct binop_closure* f; int x; int z; } clo3;
};

typedef int unop_cl_fn(union unop_data* data, int y);

struct unop_closure {
    unop_cl_fn* f;
    union unop_data* data;
};

typedef int unop_fn(struct unop_closure* clo, int y);

A closure is a pair of a function pointer and the environment variables.
Implementing Function Closures

typedef int unop(int y);

union unop_data {
    struct {} clo1;
    struct { int x; } clo2;
    struct { struct binop_closure* f; int x; int z; } clo3;
};

typedef int unop_cl_fn(union unop_data* data, int y);

struct unop_closure {
    unop_cl_fn* f;
    union unop_data* data;
};

typedef int unop_fn(struct unop_closure* clo, int y);

There are three possibilities in our example.

A closure is a pair of a function pointer and the environment variables.
Implementing Function Closures

```c
int run_unop_closure (struct unop_closure* clo, int y) {
    unop_cl_fn* f = clo->f;
    return (*f) (clo->data, y);
}

int fn1 (union unop_data* data, int y) {
    return y + 3;
}

int fn2 (union unop_data* data, int y) {
    int x = data->clo2.x;
    return x + y;
}
```

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 of the expression `fn (int y) { return x + y; }` in the program with an allocation.
Implementing Function Closures

```c
int main () {
    int x = 10;

    /* unop* g = fn (int y) { return y + 3; }; */
    struct unop_closure* g = malloc(sizeof(struct unop_closure));
    g->f = &fn1;
    g->data = malloc(sizeof(struct {}));

    /* unop* h = fn (int y) { return x + y; }; */
    struct unop_closure* h = malloc(sizeof(struct unop_closure));
    h->f = &fn2;
    h->data = malloc(sizeof(struct {int x;}));
    h->data->clo2.x = x;

    /* result = g(4) */
    int result = run_unop_closure (g, 4);
    printf ("%i\n", result);

    /* result = h(1) */
    result = run_unop_closure (h, 1);
    printf ("%i\n", result);

    return 0;
}
```

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 of the expression `fn (int y) { return x + y; }` in the program with an alloca-
Implementing Functions Closures

• Need to store variable environment and function body

• Difficulty: We cannot determine statically what the shape of the environment is

• Similar to adding a struct to the function body

• Store all variables that are captured by the function closure on the heap

• Every function needs to be treated like a closure