Advanced Language Support

• “advanced” language features so far: objects
• Next four lectures: more modern language features
  – first-class functions
  – exceptions
  – parametric polymorphism
  – dynamic typing and meta-object protocols

First-class vs. Second-class

• Values are first-class if they can be used in all the usual ways
  – assigned to local variables
  – passed as arguments to functions/methods
  – returned from functions
  – created at run-time
• Iota: modules, functions are denoted by expressions but are only usable in limited ways (uses, function call)

First-class functions

• Many languages allow functions to be used in a more first-class manner than in Iota or Java: C, C++, ML, Modula-3, Pascal, Scheme, ...
  – Passed as arguments to functions/methods
  – Nested within containing functions (exc. C, C++)
  – Used as return values (exc. Modula-3, Pascal)

Function Types

• Iota-F_0: Iota with function values that can be passed as arguments (still not fully first-class)
• Need to declare type of argument; will use program notation function(T_1, T_2): T_3 to denote the function type T_1 \times T_2 \rightarrow T_3,
• Example: sorting with a user-specified ordering:
sort(a: array[int],
  order: function(int, int):bool) {
  ...
  if (order(a[i], a[j])) {
    ... }
  ...
}
Passing a Function Value

\[
\begin{align*}
\text{leq}(x, y) & : \text{bool} = x \leq y \\
\text{geq}(x, y) & : \text{bool} = x \geq y \\
\text{sort}(a, \text{leq}) \\
\text{sort}(a, \text{geq})
\end{align*}
\]

- Allows abstraction over choice of functions

Type-checking functions

- Same rules as in Iota static semantics, but function invoked in function call may be a general expression

\[
\frac{\forall f: T_1 \times \ldots \times T_n \rightarrow T_R \in A}{A \vdash e_0 : T_1 \times \ldots \times T_n \rightarrow T_R} \quad \frac{\forall e_i: T_i \in A}{A \vdash f(e_1, \ldots, e_n) : T_R}
\]

- Subtyping on function types: usual contravariant/covariant conditions

Nested Functions

- In functional languages (Scheme, ML) and Pascal, Modula-3, Iota-F\(_1\)
- Nested function can access variables of the containing lexical scope

\[
\begin{align*}
\text{plot}_\text{graph}(f: \text{function}(x: \text{float}): \text{float}) &= \\
\text{plot}_\text{quadratic}(a, b, c: \text{float}) &= \\
& (\ldots y = f(x) \ldots)
\end{align*}
\]

iteration in Iota-F\(_1\)

- Also useful for iterators, other user-defined control flow constructs

\[
\text{countAnimals}(s: \text{set}) = \\
\begin{array}{l}
\text{count : int = 0;}
\text{loop_body(o: object) =}
\begin{array}{l}
\text{if (cast(o, Animal)) count ++;}
\end{array}
\text{s.members(loop_body);}
\text{return count;}
\end{array}
\]

- Nested functions may access, update free variables from containing scopes! Must change function representation
A subtle program

```c
int f(n: int,
g1: function(): int,
g2: function(): int) = (int x = n+10;
g(): int = x;
if (n == 0) f(1, g, dummy)
ext else if (n==1) f(2, g1, g)
ext else g1() + g2() + g())
f(0,dummy,dummy) = ?
```

Lexical scope

- `g(): int = x` creates a new function value
- Free variable (x) is bound to the variable _lexically visible_ at evaluation of function expression

```c
int f(n: int,
g1: function(): int,
g2: function(): int) = (int x = n+10;
g(): int = x;
if (n == 0) f(1, g, dummy)
ext else if (n==1) f(2, g1, g)
ext else g1() + g2() + g())
f(0,dummy,dummy) = ?
f(1, g, dummy)
f(2, g1, g)
g1(), g2(), g()
```

Closures

- Problem: nested function (g) may need to access variables _arbitrarily_ high up on stack
- Before nested functions: function value was pointer to code (1 word)
- With nested functions: function value is a _closure_ of code + environment for free variables (2 words)

Supporting Closures

```
ESEQ(MOVE(t1, e0),
CALL(MEM(t1), MEM(t1+4), e0, e1, ..., eN))
```

```
S[id,a; T] = T_R = e
```

- Can optimize direct calls
- Function variable takes 2 stack locations
- What about variable accesses?

Closure

- _Closure_ -- A pointer to the code _plus a static link_ to allow access to outer scope
- Static link passed to function code as implicit argument

Static Link Chains

```
f() = (a: int;
g() = (b:int;
h() = {
c = a + b;
) ...
} ...
```
Variable access code
- Local variable access unchanged
- Free variable access: walk up n static links before indexing to variable

Progress Report
- Passed as arguments to functions/methods
- Nested within containing functions as local variables
  - Used as return values
    - If no nested functions, functions are just pointers to code; can be used as return values (C)
    - Problem: interaction with nested functions

Iota-F₂ (first-class functions)
- Augment Iota-F₁ to allow the return type of a function to be a function itself.
  - `make_counter( ) : (function( ) : int) = (`
    - `// returns a new counter function`
    - `int count = 0;`
    - `inc( ) : int = ( count++; );`
    - `return inc;`
  - `make_counter()() + make_counter()() = ?`
  - `c = make_counter(); c() + c() + c() = ?`

Heap allocation
- Solution: heap-allocate the `make_counter` activation record (at least count)
- Activation record ≠ stack frame
- Even local variable accesses indirected

Dangling static link!
- Every function call creates an object that must be garbage collected eventually — increases rate of garbage generation
- Activation records of all lexically enclosing functions are reachable from a closure via stack link chains
- Activation record makes a lot of garbage look reachable
Escape analysis

- Idea: local variable only needs to be stored on heap if it can escape and be accessed after this function returns
- Only happens if
  - variable is referenced from within some nested function and
  - the nested function is turned into a closure:
    - returned, or
    - passed to some function that might store it in a data structure (calls to nested functions not a problem)
- This determination: escape analysis

Benefits of escape analysis

- Variables that don’t escape are allocated on stack frame instead of heap: cheap to access
- If no escaping variables, no heap allocation at all (common case)
- Closures don’t pin down as much garbage when created
- One problem: precise escape analysis is a global analysis, expensive. Escape analysis must be conservative.

Example

```make_counter(start: int): function( ) : int = {
  // returns a counter function
  int count = start;
  inc( ): int = ( c: int; count++; )
  return inc;
}
```

Summary

- Looked at 3 languages progressively making functions more first-class
- No lexical nesting (F0, C)
  - Fast but limited
  - Function = pointer to code
- Lexical nesting, no upward function values or storage in data structures (F1, Pascal, Modula-1-23): 
  - function value in closure
- Fully first-class: return values (F2, Scheme, ML):
  - lots of heap-allocation, more indirection
  - Functions roughly as powerful as objects (sometimes more convenient), but as expensive as objects… without optimization