Functions - Lecture 7

Josef Svenningsson

A Nested Function

Suppose we extended Javalette with nested functions.

```
double hypSq(double a, double b) {
  double square(double d) {
    return d * d;
  }
  return square(a) + square(b);
}
```

Nested functions

Another example

To make nested functions useful we would like to have *lexical scoping*.

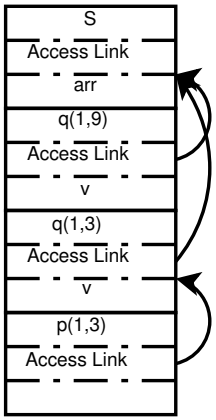
This means that we can use variables in the inner function, defined in the outer function.

```
double sqrt(double s) {
  double newton(double y) {
    return (y + s / y) / 2;
  }
  double x = 0.0;
  int i = 0;
  while (i < 10) {
    x = newton(x);
    i++;
  }
  return x;
}
```

Access Links

- *Access Links* is a mechanism to access variables defined in an enclosing procedure
- An *access link* is an extra field in a stack frame which points to the closes stack frame of the enclosing procedure

Example stack



When accessing e.g. the variable `arr` in `p` we need to go through the access link to `q` and then to `s`.

Access Links

Outline of a quicksort implementation

```
void sort(int[] arr) {
  void quicksort(int m,int n) {
    v = ..
    void partition(int y,int z) {
      .. arr .. v ..
    }
    .. a .. v .. partition .. quicksort
  }
  .. quicksort ..
}
```

Manipulating Access Links

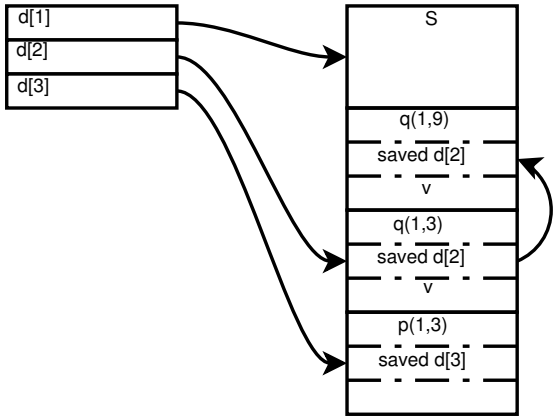
When procedure `q` calls procedure `p` there are three cases to consider:

- `p` has higher nesting depth than `q`.
  Then the depth of `p` must be exactly one larger than `q` and `p`'s access link must point to `q`.
- `p` and `q` has the same nesting depth
  The access link for `p` is the same as for `q`.
- `p` has a lower nesting depth than `q`.
  Let n_p be the nesting depth of `p` and n_q be the nesting depth of `q`. Furthermore, suppose that `p` is defined immediately within procedure `r`. The top activation record for `r` can be found by following `n_q - n_p + 1` access links down the stack.

Displays

- If the nesting depth is very large, then the link chains may be very long. Traversing these links can be costly.
- *Displays* were developed to speed up access.
- A *Display* is an stack, separate from the call stack, which maintains pointers to the most recent activation record of the different nesting depths.
  The display grows and shrinks with the maximum nesting depth of the functions on the call stack.

Displays



Lambda Lifting

- Another way of implementing nested functions is by lifting them to the top level.
- Free variables are handled by adding them as parameters to the lifted function.

Lambda Lifting - example

Original sqrt

```
double sqrt(double s) {
  double newton(double y) {
    return (y + s / y) / 2;
  }
  double x = 0.0;
  int i = 0;
  while (i < 10) {
    x = newton(x);
  }
  return x;
}
```

Lambda Lifting - example

Lambda lifted sqrt

```
double newton(double y, double s) {
  return (y + s / y) / 2;
}

double sqrt(double s) {
  double x = 0.0;
  int i = 0;
  while (i < 10) {
    x = newton(x,s);
  }
  return x;
}
```

Call-by-reference

Consider lambda lifting the function below.

The local function `incc` modifies its free variable. In order to lift `incc` we have to pass the parameter `c` *by reference*.

```
void incc(int *c) {
  (*c)++;
}

void foo() {
  int c = 0;

  incc(&c);
  incc(&c);

  printInt(c);
}
```

Call-by-reference

Consider lambda lifting the function below.

The local function `incc` modifies its free variable. In order to lift `incc` we have to pass the parameter `c` *by reference*.

```
void foo() {
  int c = 0;

  void incc() {
    c++;
  }

  incc();
  incc();

  printInt(c);
}
```

Higher Order Functions

Higher Order Fuctions in Javalette

Adding higher order functions to Javalette we need a new form of types:

Type(Type,..,Type)

Examples:

- `bool(int,int)`
  A function which takes two `int` arguments and returns a `bool`
- `void()`
  A function which takes no arguments and doesn't return anything

Higher Order Functions in Javalette

```
int main() {
  int(int) add(int n) {
    int h(int m) {
      return n + m;
    }
    return h;
  }

  int(int) addFive = add(5);
  printInt(addFive(15));
}
```

Higher Order Functions in Javalette

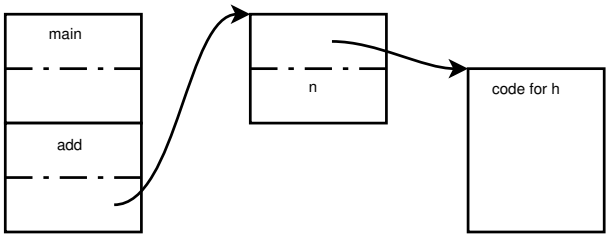
```
int main() {
  int(int) add(int n) { .. }
  int(int) addFive = add(5);

  int(int) twice(int(int) f) {
    int g(int x) {
      return f(f(x));
    }
    return g;
  }

  int(int) addTen = twice(addFive);
  printInt(twice(twice(addTen))(6));
}
```

Implementing Higher Order Functions

There are several ways implementing Higher Order Functions

- *Access Links* can be adapted to also deal with higher order functions
- *Defunctionalization* is a method to convert higher order functions to data structures. Requires whole program compilation.
- *Closures* are used to represent functions by a heap allocated record containing a code pointer and the free variables of the function.
  Closures is by far the most common implementation method

Closures



- The closure for `h` inside add contains a pointer to the code for `h` and the value for the variable `n`.
- The closure is heap allocated

Closures and mutable variables

Functional languages like Haskell and ML deal with the problem of closures and mutability as follows:

- Everything is immutable by default.
- Mutation is introduced by *references* which always live on the heap.

```
makeCounter = do
  r <- newIORef 0
  let inc = do
    n <- readIORef r
    writeIORef r (n+1)
    return n
  return inc
```

Closures and mutable variables

What happens with the stack allocated variable `counter` once we exit the function `makeCounter`?

- Heap allocate part of the stack frame
- Forbid such programs (example: Java)

```
int() makeCounter(int start) {
  int counter = start;
  int inc() {
    counter++;
    return counter;
  }
  return inc;
}
```

Anonymous nested functions - lambda expressions

- An increasingly popular language feature is to have anonymous nested functions, so called *lambda expressions*.
- Compiling lambda expressions works the same way as nested functions with names.

A note on terminology

One can often hear the phrase that a language *"has closures"*.

This is a somewhat unfortunate use of the word.

Closures is an *implementation technique* for the *language feature* higher order functions.

Lazy evaluation

Thunks

- Is it possible to implement `if` as a function?
- We can fake it by using functions which take no arguments

```
void if(bool c, void() th) {
  if (c)
    th();
}
```

Thunks

- *Call-by-name* is a calling convention where the arguments are not evaluated until needed.
- *Thunks* are used to implement call-by-name. Thunks are essentially functions which take no arguments. They are typically implemented as closures.

Lazy evaluation

- The difference between call-by-name and lazy evaluation is that once an argument is evaluated, it is not reevaluated if it is used twice.
- In order to achieve laziness, once the value is computed we need to remember it. This can be done in two ways:
  - Overwrite the thunk with an indirection pointing to the value.
  - Overwrite the thunk with the value directly, if the space allocated for the thunk is big enough to hold the value.

A Note

- Call-by-name and lazy evaluation is very handy as they allow the programmer to create new control structures.
- Be careful with combining them with side-effects. It can yield very surprising results. An impure language with lazy evaluation as default is a bad idea.