Overview

Compiling nested methods

Inline method expansion

Constants

Examples
Optimizing Functional SSA

Recap: we are optimizing Functional SSA programs... what does this mean?
Compiling nested methods

Four ways to deal with nested methods:

1. Static links.
2. Closure conversion.
3. Lambda lifting.
4. Limit inner method calls to tail-calls.

Look at code generated by (4)...
Compiling nested methods

An example:

class Fact implements Object {
    method fact (n:Integer) : Integer {
        method loop (result:Integer, i:Integer) : Integer {
            if (i < n) {
                let i:Integer = i + 1;
                let result = result * i;
                return this.loop (result, i);
            } else {
                return result;
            }
        }
        return inner.loop (1, 1);
    }
}
Inline method expansion

What is inline method expansion?

Why do inline method expansion?

How can we inline expand:

```java
final class C {
    method foo (Params) : Result {
        B1
    }
}
class D {
    method bar (c : C) {
        let r:Result = c.foo(Args); B2
    }
}
```
Inline method expansion

An example:

```java
final class C {
    field s : String;
    method prefix $ () : String { return this.s; }
}

class D {
    method foo (c : C) : String {
        let tmp : String = $c;
        return "c = " + tmp;
    }
}
```
**Inline method expansion**

Another example:

```java
final class C {
    field b : Boolean;
    method prefix $ () : String {
        if (this.b) { return "hello"; }
        else { return "world"; }
    }
}

class D {
    method foo (c : C) : String {
        let tmp : String = $c;
        return "c = " + tmp;
    }
}
```
Inline method expansion

What about:

```java
final class C {
    field b : Boolean;
    method prefix $ () : String {
        if (this.b) { return "hello";
        } else { return "world";
        }
    }
}

class D {
    method foo (c : C) : String {
        let tmp : String = $c;
        ...
        a very large block ...
    }
}
```
Inline method expansion

What about:

```java
final class C {
    field b : Boolean;
    method prefix $ () : String {
        let x : Boolean = this.b;
        return $x;
    }
}

class D {
    method foo (c : C) : String {
        let x : Integer = 37;
        let tmp : String = $c;
        return tmp + $x;
    }
}
```

We have to be careful about variable scope!
Inline method expansion

What about:

```java
method bar (x : Integer) {
    method foo (y : Integer) {
        stdout.print("result is ");
        stdout.println($y);
    }
    var a : Integer; a := 0; // can a be put in a register?
    var b : Integer; b := 1; // can b be put in a register?
    while (a < x) {
        a := a + 1; b := b*a;
    }
    inner.foo (b); // should we inline this?
}
```

Method inlining can sometimes slow code down!
Inline method expansion

What about:

    class B { method bar (C c) { c.foo (); } }  
    class C { method foo () { print("hello"); } }  

Can we inline c.foo()?
Inline method expansion

What about:

```plaintext
class B { method bar (C c) { c.foo (); } }
class C { method foo () { print ("hello"); } }
class D extends C { method foo () { print ("world"); } }
```

Now can we inline c.foo()?

We need to know the exact type of c. How can we get this?
Inline method expansion

What about:

```java
method fact (x : Integer) {
    if (x <= 0) { return 1; }
    else {
        let tmp1 : Integer = x-1;
        let tmp2 : Integer = this.fact (tmp1);
        return tmp2 * x;
    }
}
```

When do we stop inlining?
Inline method expansion

Heuristics for inlining:

a) Inline small methods.

b) Inline methods which are called very frequently (how can we detect this?)

c) Inline methods which are called once.

Hobbes does (a) because it’s easy!
Constants

What can we do with:

// constant propagation
let x = 37; B

// constant folding
let x = 1 + 2; B
let b = 1 == 2; B

// constant condition
if (True) { B1 } else { B2 }
Constants on the heap

What can we do with:

class Foo { immutable field a : Integer; }
method foo () : String {
    let x = new Foo { a = 37; }
    return "a = " + $x.a;
}

What about:

class Foo { immutable field a : Integer; }
method foo () : String {
    let x = new Foo { a = 37; }
    bar (x);
    return "a = " + $x.a;
}
Constants on the heap

What can we do with:

```java
class Foo { mutable field a : Integer; }
method foo () : String {
    let x = new Foo { a = 37; }
    bar (x);
    return "a = " + $x.a;
}
```

What about:

```java
class Foo { mutable field a : Integer; }
method foo (z : Bar) : String {
    let x = new Foo { a = 37; b = true; }
    z.c = x;
    return "a = " + $x.a;
}
```

This requires *escape analysis*. 
Homework

Implement inline method expansion.
Summary

For code generation for Functional SSA we need to deal with nested methods: either by static links, closure conversion or lambda-lifting.

The main optimizations for Functional SSA are inline method expansion and constant folding. Functional SSA relies on tail call optimization for efficiency.

Constant propogation for heap-allocated objects is possible, but requires escape analysis for correctness.

Next week: Garbage collection