CSC 548: Lecture 3

Overview

Low-level optimizations

Code generation

The Hobbes code generator

Register allocation

Low-level optimizations

Most optimizations are source-to-source transforms.

Some low-level optimizations are in the code generator.

Which optimizations should be source-to-source? Which should be low-level?

What are the tradeoffs?

Register allocation

What is register allocation?

What makes for a correct register allocator?

What makes for a good register allocator?

How does register allocation affect code generation?

Peephole optimization

What is peephole optimization?

What kind of optimizations are good for a peephole optimizer?

Code generation

What is a frame (a.k.a. activation record)?

How are frames allocated for method calling?

What is the standard GCC calling convention?

Code generation

What is a callee save register? What is a caller save register?

The GCC calling convention is that %ebx, %esp, %ebp, %esi, and %edi are all callee save. What impact does this have on codegen? On regalloc?

Hobbes can use its own calling convention! We make all general purpose registers caller save. What impact does this have on codegen? On regalloc?

Hobbes also passes parameters in registers rather than on the stack. Why?

Code generation

IA32 instructions are (mostly) of the form:

```
instruction source, target
```

For example:

```
movl 0(%esp), %edi
movl %edi, 4(%esp)
```

What does this do? Why not just `movl 0(%esp), 4(%esp)`?

What code should we generate for `let z : Integer = x+y;`?

Code generation

Some optimizations we can perform during code generation:

1. Don't emit code of the form `movl foo, foo`.

2. Tail-call optimization: what is this? How do we emit code for it?

When can we perform tail-call optimization? How does the gcc calling convention effect tail-call optimization?

The Hobbes code generator

```
package hobbes.arch;
public interface Frame extends PrettyPrintable {
    public int numLocals ();
    public Access getLocal (int index);
    public Access newLocal ();
    public int numRegisters ();
    public Access getRegister (int index);
    public int numScratchRegisters ();
    public Access getScratchRegister (int index);
    public int numCalleeParams ();
    public Access getCalleeParam (int index);
    public Access getCalleeResult ();
    public Access getCallerParam (int index, int numParams);
    public Access getCallerResult ();
    ...
    public void addVar (Var var, Access location);
    public Access getVar (Var var);
    public boolean containsVar (Var var);
}
```

The Hobbes code generator

```
package hobbes.regalloc;
aspect RegAlloc {
    public void Block.regalloc (Var[] params, Frame frame, PrettyPrinter debug) { ... }
    ...
}
```

How can we implement a simple register allocator?

The Hobbes code generator

```
package hobbes.ia32;
aspect CodeGen {
    public void Program.codegen (PrettyPrinter out) { ... }
}
```

How can we implement the code generator?

The Hobbes code generator

One part of the code generator:

```
public void ExpCall.codegen (Access dest, Frame frame, PrettyPrinter out, StringPool pool) {
    String method = getMethod (); Val[] args = getArgs ();
    Type objType = args[0].getType ();
    String name = "" + objType + "$" + method;
    for (int i=0; i<args.length; i++) {
        args[i].codegen (frame.getCallerParam (i, args.length), frame, out, pool);
    }
  int paramStackSize = IA32Frame.WORD * (args.length - frame.numRegisterParams ());
    out.println ("subl $" + paramStackSize + ", %esp");
    out.println ("call " + name);
    out.println ("addl $" + paramStackSize + ", %esp");
    frame.getCallerResult ().codegen (dest, frame, out);
}
```

What does this code do?

Homework

Implement tail-call optimization.

Summary

Register allocation is important phase of a compiler, where local variables are allocated registers or stack slots.

The code generator requires register allocation to have taken place before codegen.

Your task is to implement a nïave register allocator, which puts every local variable into a new stack slot.

Next week: real register allocation using graph coloring!