1 Introduction

In this lab assignment, we get to the final part of the compiler: the back-end. The task in the back-end is to generate machine code from the FIR we produced in the middle-end. The back-end has four parts:

1. The FIR is converted to *abstract-assembly language* by the `aal/fc_aal_fir.ml` module, which is provided for you. The instruction set for the AAL is defined in the file `cg_type/aal_type.mlz`.
2. The code is converted to real assembly code. In our case, we are produceing code for the Intel x86 architecture. The x86 code generator is defined in the file `arch/i386/x86_codegen.ml`.
3. Dataflow analysis is used to calculate the *interference graph* that defines which variables can't be stored in the same machine registers. The dataflow analysis is defined in `cg/dataflow.ml`.
4. The *register allocator* brings it all together. The register allocator is passed AAL, and it performs a loop:
   (a) produce assembly code,
   (b) perform dataflow analysis to get an interference graph,
   (c) try to assign the variables to registers. If some variables are spilled, rewrite the assembly code and start over.

2 Important notes from Lab 4

As you know, the escape analysis is changed for lab 5. The new analyzer guarantees that each escaping function has at most one free variable. You **must** use the new escape analysis for Lab 5.

The FIR includes a new instruction.

2.1 `LetFuns2`

The `LetsFuns2` instruction divides functions into *four* classes:

FunDeclClass the function is a function declaration (and it contains a `LetExtCall` in it's body).

FunContClass the function is a continuation generated by `Fc_ir_cont`.

FunLocalClass the function is a local function (this corresponds to a `false` flag in a `LetFun` definition).

FunGlobalClass the function is a global function (this corresponds to a `true` flag in a `LetFun` definition).

Here is what you need to do to get your code working.

- You must define each continuation in `Fc_ir_cont` with `LetFuns2`, and the `FunContClass`.

- The new escape analysis will generate functions in `LetFuns2`.
- The closure conversion doesn't really care what class functions belong to, but you need to add the match code for the `LetFuns2` definition.

Also, during closure conversion, you *must* treat all `SetVars` as *binding* occurrences. That is, for the dataflow analysis, a `SetVar` should be treated as if it were a `LetAtom`.

3 What you need to do

For this lab, you will implement three parts:

1. x86 code generation (in the `arch/i386/x86_codegen.ml` file),
2. dataflow analysis (in the `cg/dataflow.ml` file),
3. register allocation (in the `cg/register_alloc.ml` file).

The register allocator will be the most difficult part, you will benefit greatly by reading the book (especially Chapter 11).

4 Getting started

For code generation, you will need to understand the format of the AAL. The AAL expressions are defined in the file `cg_type/aal_type.mlz`. The instruction have two major parts: addresses and expressions.

4.1 Addresses

There are two kinds of addresses defined by the AAL: *safe* addresses, and *unsafe* addresses. Unsafe addresses must produce code to check the bounds of the memory reference (the addressing code is provided for you in the `x86_codegen.ml` file).

```
type addr =
   AddrSafe of atom
 | AddrSafeOff of atom * atom
 | AddrUnsafeOff of atom * atom * atom
 | AddrSafeOff2 of atom * atom * atom
 | AddrUnsafeOff2 of atom * atom * atom * atom
 | AddrSafeMul of atom * atom * int
 | AddrSafeMulOff of atom * atom * int * atom
 | AddrUnsafeMulOff of atom * atom * int * atom * atom
```

For the `AddrSafe a` address, the atom a must be a pointer. The address refers to the value stored at a.

For the `AddrSafeOff (a, i)` address, the atom a is a pointer, and the atom i is an unsigned integer. The value is stored at location $a + i$ (i is a byte offset).

For the `AddrUnsafeOff (a, i, b)`, the atom a is a pointer, the atom i is an unsigned number, and the atom b is an unsigned number that represents the maximum value of i. It is an error if $i \geq b$.

The rest are similar.

`AddrSafeOff2(a, i_1, i_2)`: value at $a + i_1 + i_2$.

`AddrUnsafeOff2 (a, i_1, a_2, b)`: value at $a + i_1 + i_2$, for $i_1 + i_2 < b$.

`AddrSafeMul (a, i, j)`: value at $a + i * j$.

`AddrSafeMulOff (a, i_1, j, i_2)`: value at $a + i_i * j + i_2$.

`AddrUnsafeMulOff (a, i_1, j, i_2, b)`: value at $a + i_1 * j + i_2$, for $(i_1 * j + i_2 < b)$.

4.2 AAL expressions

```
type exp =
   (* Arithmetic *)
   LetAtom of var * atom * exp
 | LetUnop of var * unop * operand * exp
 | LetBinop of var * binop * operand * operand * exp

   (*
    * Branches and functions.
    * Jump jumps to a label (the name of a function).
    * RJump jumps to an address stored in a register.
    * CJump jumps based on condition code.
    *)
 | Jump of label
 | RJump of operand
 | CJump of relop * operand * operand * label * label
 | Call of label

   (*
    * Reserve is use for garbage collection.
    * The var list is the list of vars that
    * contain pointers.
    *)
 | Reserve of var list

   (*
    * Memory operations.
    * LetMem/SetMem access words.
    * LetMemB/SetMemB access bytes.
    * The size in Memcpy is the number of words.
    *)
 | LetAddr of var * addr * exp
 | LetMem of var * addr * exp
 | SetMem of addr * operand * exp
 | LetMemB of var * addr * exp
 | SetMemB of addr * operand * exp

   (* External interface *)
 | LetExtCall of var * var * atoms * exp
```

The `LetAtom`, `LetUnop`, and `LetBinop` instructions correspond to the same instructions in FIR. Note that `LetUnop` and `LetBinop` take arbitrary operands as arguments.

The `Jump` instruction jumps to a label (which defines a block). The `RJump addr` instruction jumps to the address in the operand. The `CJump` instruction performs the comparison on the two operands, then jumps to the first label if the comparison holds, and the second label otherwise.

The `Reserve` instruction is used for garbage collection. The `X86_inst_type` module contains a pseudo-instruction `RES` for this instruction.

The `LetAddr` instruction is like `LetAtom`, but it saves the *address* represented by the operand. The `LetMem` and `SetMem` instructions access memory (in 32-bit words). The `LetMemB` and `SetMemB` instructions access *bytes* in memory.

`LetExtCall (v, f, args, e)` saves the value of the external C function call `f(args)` in variable v.

4.3 The x86 instruction set

For documentation on the Intel instruction set, you should refer to the Intel documentation (on the CS134b home page). The instructions follow the Intel notation: if there are two operands, the first is the destination. This differs from `gas` assembler syntax, where the operands are in reverse order (why, I don't know).

5 What to turn in

You should turn in your entire compiler in your `submit/lab5` directory on `mojave` (you should probably keep a copy in your CS account too). IF you are working in a group, only one of you should do the submission, and you should send mail to `cs134-admin` to tell us of your submission.

In addition, you should include the following.

- A `README` file explaining what you did, how it works, and whether you had any problems.
- A `DIFF` file generated using the command `cvs diff` in the `arch/i386` and `cg` directories. If you like, you can insert this into the `README` file, with brief explanations of what you changed.
- The files `test1.s`, `test2.s`, `test3.s`, and `test4.s` generated using your compiler with the `-o` option.

```
% ./fcc -o test1.s test1.c
% ./fcc -o test2.s test2.c
% ./fcc -o test3.s test3.c
% ./fcc -o test4.s test4.c
```