Compiler Construction
Lent Term 2014
Lectures 15, 16

- **More on the Slang.3 compiler**

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

Example of L3 to IR1 translation

counter.slang

```
fun mk_counter (i : int) : unit -> int =
    let c : int ref = ref (i)
    in fun inc () : int =
          let x : int = !c
          in (c := !c + 1; x) end
       in inc end
    end
in
   let v : int  = read()
   in let g : unit -> int = mk_counter (v)
      in (print (v); print (g ()); print (g ()); print (g ()); print (g ())) end
   end
end
```

```
let _6_v = KNOWN(read) ()
 in let _7_g = KNOWN(_1_mk_counter) (_6_v)
    in (KNOWN(print) (_6_v);(KNOWN(print) (_7_g ());(KNOWN(print) (_7_g ());
      (KNOWN(print) (_7_g ()); KNOWN(print) (_7_g ())))))
    end
 end
fun _1_mk_counter (_2_i ) =
   let _3_c = ref _2_i
    in let _9_CL = CLOSURE(_4_inc, [_3_c])
       in _9_CL end
    end;
fun _4_inc (_8_ENV ) =
      let _5_x = !_8_ENV[1 in (_8_ENV[1] := (!_8_ENV[1] + 1); _5_x) end;
```

Details of L3 to IR1 translation

```
…
    fun f (X) = body in e end
…
```

If every free variable in `body` is either `f` or in `X`,
then translated this as a “known” function.
Otherwise, let the free variables be `{a1, a2, …, ak}`
and translate as

```
…
     let g = Closure(f, [a1, a2 ... ak])
     in translate(e)[f <- g] end
…

fun f(ENV, x) =
   translate(body)[ai <- ENV[i], f(exp) <- f(ENV, exp)]
```

IR2

```
type index = int       type range = int       type value_loc = index * range

datatype ir2_var_kind = IR2_ArgVar of var * value_loc
                      | IR2_EnvVar of var * value_loc  * int
                      | IR2_LocalVar of var * value_loc

datatype ir2_expr =
          IR2_Comment of string
        | IR2_Label of label
        | IR2_Skip
        | IR2_Halt
        | IR2_Var of ir2_var_kind
        | IR2_KnownFun of var
        | IR2_Integer of int
        | IR2_Boolean of bool
        | IR2_UnaryOp of unary_oper * (ir2_expr list)
        | IR2_Op of (ir2_expr list) * oper * (ir2_expr list)
        | IR2_Assign of (ir2_expr list) * (ir2_expr list)
        | IR2_StoreLocal of (ir2_expr list) * int
        | IR2_Deref of (ir2_expr list)
        | IR2_Ref of (ir2_expr list)
        | IR2_Discard of (ir2_expr list)
        | IR2_App of (ir2_expr list) * (ir2_expr list list)
        | IR2_Jump of label
        | IR2_Fjump of (ir2_expr list) * label
        | IR2_Closure of var * (ir2_var_kind list)

(* IR2_FunLabel(f, number_of_locals, number_of_args, body) *)
datatype ir2_function = IR2_FunLabel of label * int * int * (ir2_expr list)

type program = (ir2_expr list) * (ir2_function list)
```

Using (ir2_expr list) here allows us to avoid the awkward distinction between expressions and statements used in the slang1 compiler.

Example of IR1 to IR2 translation

rem2.slang

```
KNOWN(_Main) ({});
halt;
FUNLABEL(_Main, 2, 0){
   LOCAL[1] := KNOWN(read) ();
   LOCAL[2] := KNOWN(read) ();
   KNOWN(print) (KNOWN(_1_rem) LOCAL[_6_x1, 1, 2], LOCAL[_7_x2, 2, 2]))
};

FUNLABEL(_1_rem, 1, 2){
   LOCAL[1] := CLOSURE(_4_aux, [ARG[_3_n, 2, 2]];
   FJUMP (ARG[_3_n, 2, 2] >= 1) _L0;
   LOCAL[_9_CL, 1, 1] (ARG[_2_m, 1, 2]);
   JUMP _L1;
   LABEL(_L0);
   0;
   LABEL(_L1)
};

FUNLABEL(_4_aux, 0, 2){
   FJUMP (ARG[_5_m, 2, 2] >= ENV[_8_ENV, 1, 2, 1]) _L2;
   KNOWN(_4_aux) (ARG[_8_ENV, 1, 2], (ARG[_5_m, 2, 2] - ENV[_8_ENV, 1, 2, 1]));
   JUMP _L3;
   LABEL(_L2);
   ARG[_5_m, 2, 2];
  LABEL(_L3)
};
```

Example of IR1 to IR2 translation

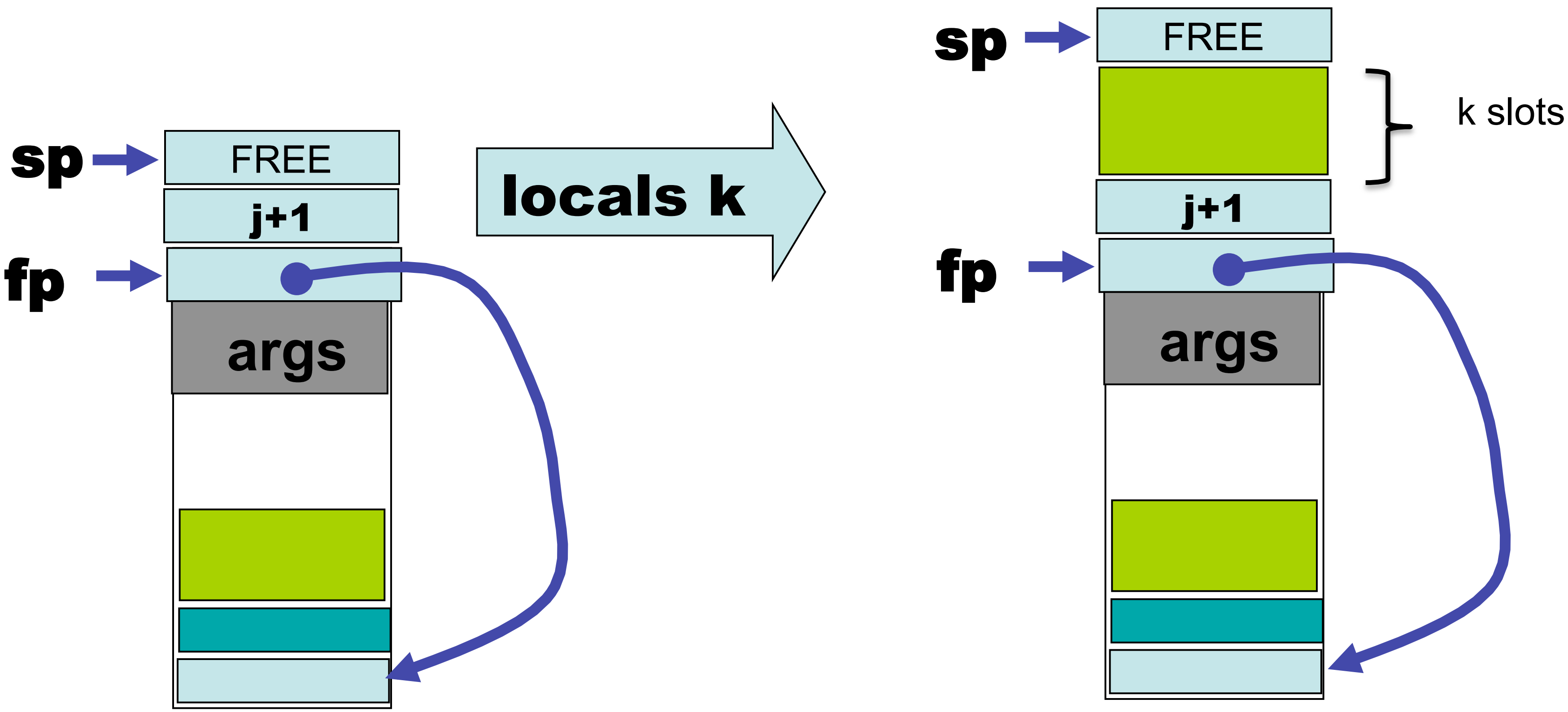
counter.slang

```
KNOWN(_Main) ({});
Halt;
FUNLABEL(_Main, 2, 0){
   LOCAL[1] := KNOWN(read) ();
   LOCAL[2] := KNOWN(_1_mk_counter) (LOCAL[_6_v, 1, 2]);
   DISCARD KNOWN(print) (LOCAL[_6_v, 1, 2]);
   DISCARD KNOWN(print) (LOCAL[_7_g, 2, 2] ());
   DISCARD KNOWN(print) (LOCAL[_7_g, 2, 2] ());
   DISCARD KNOWN(print) (LOCAL[_7_g, 2, 2] ());
   KNOWN(print) (LOCAL[_7_g, 2, 2] ())
};

FUNLABEL(_1_mk_counter, 2, 1){
   LOCAL[1] := REF ARG[_2_i, 1, 1];
   LOCAL[2] := CLOSURE(_4_inc, [LOCAL[_3_c, 1, 2]];
   LOCAL[_9_CL, 2, 2]
};

FUNLABEL(_4_inc, 1, 1){
   LOCAL[1] := DEREF ENV[_8_ENV, 1, 1, 1];
   DISCARD ENV[_8_ENV, 1, 1, 1] := (DEREF ENV[_8_ENV, 1, 1, 1] + 1);
   LOCAL[_5_x, 1, 1]
};
```

Allocation for let-bound local values



load j = push value at fp + j

store j = pop top into location at fp + j

Finally, on to IR2 to VSM translation

rem2.slang

```
         call _Main
         hlt
_Main :  locals 2
         rdi
         store 1
         rdi
         store 2
         load 1
         load 2
         call _1_rem
         pri
         push 0
         return 0
```

```
_1_rem : locals 1
         arg 1
         closure _4_aux 1
         store 1
         push 1
         arg 1
         sub
         ifp _L4
         push 0
         jmp _L5
_L4 :    push 1
_L5 :    ifz _L0
         load 1
         arg 2
         callc 2
         jmp _L1
_L0 :    push 0
_L1 :    return 2
```

```
_4_aux : arg 2
         deref 2
         arg 1
         sub
         ifp _L6
         push 0
         jmp _L7
_L6 :    push 1
_L7 :    ifz _L2
         arg 2
         arg 2
         deref 2
         arg 1
         sub
         call _4_aux
         jmp _L3
_L2 :    arg 1
_L3 :    return 22
```

IR2 to VSM translation

counter.slang

```
          call _Main
           hlt
_Main :    locals 2
           rdi
           store 1
           load 1
           call _1_mk_counter
           store 2
           load 1
           pri
           push 0
           pop
           load 2
           callc 1
           pri
           push 0
           pop
           load 2
           callc 1
           pri
           push 0
           pop
           load 2
           callc 1
           pri
           push 0
           pop
           load 2
           callc 1
           pri
           push 0
           return 0
```

```
_1_mk_counter :
        locals 2
        arg 1
        ref
        store 1
        load 1
        closure _4_inc 1
        store 2
        load 2
        return 1
```

```
_4_inc :locals 1
        arg 1
        deref 2
        deref 1
        store 1
        arg 1
        deref 2
        arg 1
        deref 2
        deref 1
        push 1
        add
        assign
        push 0
        pop
        load 1
        return 1
```

“Peep hole” optimisation

Syntax-directed compilation often concatenates sequences of code

sequence for construct 1	sequence for construct 2	sequence for construct 3	… …

This can lead to unexpected code combinations across boundaries.

Peep hole optimisation is a technique used to scan code, looking only at a narrow window of instructions, and rewriting unwanted patterns to more efficient code.

```
fun vsm_peep_hole prog =
    let fun aux carry [] = List.rev carry
          | aux carry ((VSM_Push _) :: VSM_Pop :: rest) = aux carry rest
          | aux carry (a :: rest) = aux (a :: carry) rest
       val code =
in aux [] prog  end
```

After peep-hole

counter.slang

```
        call _Main
        hlt
_Main : locals 2
        rdi
        store 1
        load 1
        call _1_mk_counter
        store 2
        load 1
        pri
        load 2
        callc 1
        pri
        load 2
        callc 1
        pri
        load 2
        callc 1
        pri
        load 2
        callc 1
        pri
        push 0
        return 0
```

```
_1_mk_counter :
        locals 2
        arg 1
        ref
        store 1
        load 1
        closure _4_inc 1
        store 2
        load 2
        return 1
```

```
_4_inc :locals 1
        arg 1
        deref 2
        deref 1
        store 1
        arg 1
        deref 2
        arg 1
        deref 2
        deref 1
        push 1
        add
        assign
        load 1
        return 1
```

Compiler, still clean and simple

```
fun back_end fout sl =
    case !Global.target of
      Global.VSM => vsm_emit.emit_vsm_bytecode fout
                      (vsm_assemble.vsm_assemble
                        (vsm_assemble.vsm_peep_hole
                          (IR2_to_VSM.ir2_to_vsm sl)))
    | Global.VRM => (print "\n\n No VRM backend yet\n")


fun compile fin fout =
    let val lexbuf = (createLexerStream (Nonstdio.open_in_bin fin))
    in
       (back_end fout
          (IR1_to_IR2.translate
                (L3_to_IR1.translate
                   (Alpha.convert
                      (Typing.check_types
                         (Parser.main Lexer.token lexbuf))))))
         handle Parsing.ParseError f => reportParseError lexbuf
    end
```

What about a register-oriented target? What is the “register allocation problem”?

At some point in the back-end, the compiler must confront the fact that the target machine does not have an infinite number of registers.

A solution will

- Assign temporaries to finite number of registers
- Attempt to assign source and target of “move” instructions to same register so that the move can be eliminated

Of course the “live” temporaries at a given point in a program may not fit in the available registers, so the associated values must be “spilled” into memory (into a stack frame, or onto the heap).

Good solutions to this problem require the kind of “dataflow analysis” that is covered in Optimising Compilers (Part II). In the meantime, if you are curious see Appel Chapters 10 and 11.

Sections of the Lecture Notes (cc_notes_2014.pdf) NOT covered in Lectures, and so NOT examinable

- Section 7 : Code generation of Target Machine
- Section 8 : Object Modules and Linkers
- Section 9.2 : Lambda calculus
- Section 9.4 : Mechanical evaluation of lambda expressions
- Section 9.6 : A more efficient implementation of the environment
- Section 10.9 : Arrays
- Section 10.11 : C++ multiple inheritance
- Section 10.16 : Data types
- Section 10.17 : Source-to-source translation
- Section 11: Compilation revisited and debugging