Compiler Construction
Lent Term 2013
Lecture 11 (of 16)

1. **Slang.2 as BAD SYNTAX for a subset of L3 from SPL**
2. **Short-circuit boolean operations**
3. **Improving the "middle-end" using Appel's Intermediate Representation (Tree)**

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

Slang.2 (AST_expr.sml)

```
datatype type_expr =
         TEint
       | TEunit
       | TEbool
       | TEfunc of type_expr * type_expr
       | TEref of type_expr


type var = string

datatype oper = Plus
              | Mult
              | Subt
              | GTEQ
              | EQ   (* New, not in Slang.1 *)

datatype unary_oper = Neg | Not
```

Slang.2 is BAD SYNTAX for a subset of L3 from SPL output by the parser. This subset excludes tuples, records, inl, inr, case, and objects. Implementing these features are exercises for the ambitious student!

Slang.2 (AST_expr.sml)

```
datatype expr =
          Skip
        | Integer of int
        | Boolean of bool
        | UnaryOp of unary_oper * expr
        | Op of oper * expr * expr
        | Assign of expr * (type_expr option) * expr
        | Deref of expr
        | Seq of expr * expr
        | If of expr * expr * expr
        | While of expr * expr
        | Print of (type_expr option) * expr

        (* new constructors, not in Slang.1 *)
        | Ref of expr
        | Var of var
        | Loc of var   (* will not get this from parser *)
        | Fn of var * type_expr * expr
        | App of expr * expr
        | Let of var * type_expr * expr * expr
        | Letrecfn of var * type_expr * var * type_expr * expr * expr
```

Subset of Slang.2 concrete syntax (functions/types in next Lecture)

```
program := expr EOF

expr := simple
| set expr := expr
| while expr do expr
| if expr then expr else expr
| begin expr expr_list
| let identifier : type_expr
  = expr in expr end

expr_list := ; expr expr_list
           | end

simple ::= term srest

term ::= factor trest
```

```
srest ::=  + term srest
        |  - term srest
        |  >= term srest
        |  = term srest
        |  && term srest
        |  || term srest
        |

trest ::=  *  factor trest
           |

factor := identifier
| integer
| - expr
| true | false | skip
| ( expr )
| print expr
| ref expr
| ! expr
```

NEW

squares.slang

In Slang.1

```
begin
  set n := 10;
  set x := 1;
  while n >= x do
  begin
    print (x * x);
    set x := x + 1
  end
end
```

In Slang.2

```
let n : int = 10
in
   let x : ref int = ref 1
   in
      while n >= !x do
      begin
        print (!x * !x);
        set x := !x + 1
      end
   end
end
```

All identifiers in Slang.2 must be declared (let-bound or formal parameter).

Note: “short-circuit” boolean operations

`e1 || e2` — Expand in parser → `if e1 then true else e2`

`e1 && e2` — Expand in parser → `if e1 then e2 else false`

- Pros : easy!
- Cons : harder to link type errors to program text

Let's solve a problem with the Slang.1 compiler

VRM

```
      normalize        vrm_code_gen
expr   ->   normal_expr   ->   vrm_assembler
```

VSM

```
      vsm_code_gen
expr   ->   vsm_assembler
```

Problem : there is nothing shared after the front end. This will lead to more and more duplication of effort as Slang gets more complex and optimizations become more involved.

Shopping for an Intermediate Representation (IR)

VRM

```
        trans      vrm_code_gen
expr   ->   NEW_IR   ->   vrm_assembler
```

VSM

```
        trans      vsm_code_gen
expr   ->   NEW_IR   -> vsm_assembler
```

The IR can then be used for target-independent optimizations (such as dataflow, OC, Part II).

Since the "stack-oriented" path does not need to "name" intermediate values, the IR should at least eliminate structured control constructs.

Appel's Intermediate Representation, Tree.sml

```
datatype stm     = SEQ of stm * stm
                 | LABEL of Temp.label
                 | JUMP of exp * Temp.label list
                 | CJUMP of relop * exp * exp * Temp.label * Temp.label
                 | MOVE of exp * exp
                 | EXP of exp

      and exp    = BINOP of binop * exp * exp
                 | MEM of exp
                 | TEMP of Temp.temp
                 | ESEQ of stm * exp
                 | NAME of Temp.label
                 | CONST of int
                 | CALL of exp * (exp list)

and binop = PLUS | MINUS | MUL | DIV
                 | AND | OR | LSHIFT | RSHIFT | ARSHIFT | XOR

      and relop = EQ | NE | LT | GT | LE | GE
                 | ULT | ULE | UGT | UGE
```

This stm/exp IR is good enough for the Slang.1 langauge, but we will need more for the functions of Slang.2.

See Appel, Chapter 7 (ML version). Fragments of the code can be found at http://www.cs.princeton.edu/~appel/modern/

Appel's Intermediate Representation, Tree.sml

- `SEQ(s1, s2)`
  - s1 followed by s2
- `LABEL l`
  - The current machine code address
- `JUMP(e, labs)`
  - Evaluate e to a location, jump there
- `CJUMP(rop, e1, e2, l_true, l_false)`
  - Evaluate the boolean "e1 rop e2", jump to the appropriate label
- `MOVE(e1,e2)`
  - Evaluate e1 to an address, move the results of e2 there
- `EXP e`
  - Evaluate e and discard the result

stm type is for side effects

- `BINOP(bop, e1, e2)`
  - Evaluates to "e1 bop e2"
- `MEM e`
  - Contents of address at the value of e
- `TEMP t`
  - A temporary location
- `ESEQ(s, e)`
  - Perform s, then return value of e
- `NAME l`
  - A label
- `CONST n`
  - An integer constant
- `CALL(e, el)`
  - Evaluate e to a function, call it on the values of el

exp type is for values

Houston, we have a problem

- `trans e` : could map into either `stm` or `exp`
- We need an a datatype that is the "union" of the two types

Appel's solution:

```
datatype tree_rep  = Ex of exp
                   | Nx of stm
                   | Cx of Temp.label * Temp.label -> stm
```

Note: Appel calls this datatype `exp`, but I've changed the name to `tree_rep` to avoid confusion.

`Ex e` : evaluate to a value
`Nx s` : No value, just side effect
`Cx f` : Conditional, `(f (tree_l, false_l))`, with no "fall through"

Houston, we still have a problem

- Think about something like `trans(If(e1,e2,e3))`
- We call `(trans e1)` and get a `(Cx f)`, OK
- But calls to `(trans e2)` and `(trans e3)` can each return one of three constructions. So nine cases to consider!

Appel suggests helper functions :

```
val unEx : tree_rep -> exp
val unNx : tree_rep -> stm
val unCx : tree_rep -> (Temp.label * Temp.label -> stm)
```

unEx

```
val zero = CONST 0
val one = CONST 1

fun seq([]) = Library.internal_error "seq : given empty list!"
  | seq([s]) = s
  | seq(h::t) = SEQ(h, seq(t))

fun unEx(Ex e) = e
  | unEx(Cx mk_stm) =
      let
        val result      = TEMP (Temp.newtemp())
        val true_label  = Temp.newlabel()
        val false_label = Temp.newlabel()
      in
          ESEQ(seq[MOVE(result, one),
                     mk_stm(true_label ,false_label),
                     LABEL false_label,
                     MOVE(result, zero),
                     LABEL true_label],
                 result)
      end
  | unEx(Nx s) = ESEQ(s, zero)
```

unNx

```
fun unNx(Ex e) = EXP(e)
  | unNx(Cx mk_stm) =
    let val join = Temp.newlabel()
    in
        SEQ(mk_stm(join, join), LABEL join)
    end
  | unNx(Nx s) = s
```

unCx

```
fun  unCx(Ex(CONST 0)) = (fn (t, f) => JUMP(NAME f, [f]))
   | unCx(Ex(CONST _)) = (fn (t, f) => JUMP(NAME t, [t]))
   | unCx(Ex e) = (fn (t, f) => CJUMP(NE, e, zero, t, f))
   | unCx(Cx f) = f
   | unCx(Nx _) = Library.internal_error "unCx: given an Nx!"
```

```
fun tr_if e1 then_e else_e  =
      let
        val cond = unCx(e1)
        val then_label = Temp.newlabel()
        val else_label = Temp.newlabel()
      in
        case (then_e, else_e) of
          (Ex _, Ex _) =>
          let val r = Temp.newtemp()
              val joinLabel = Temp.newlabel() in
          Ex (ESEQ (seq [cond (then_label, else_label),
                         LABEL then_label,
                         MOVE (TEMP r, unEx then_e),
                         JUMP (NAME joinLabel, []),
                         LABEL else_label,
                         MOVE (TEMP r, unEx else_e),
                         LABEL joinLabel],
                    TEMP r))
          end
        | (Cx _, Cx _) => Cx (fn (t, f) =>
                let val then_exp = (unCx then_e) (t, f)
                    val else_exp = (unCx else_e) (t, f)
                in
                   seq [cond (then_label, else_label),
                        LABEL then_label, then_exp,
                        LABEL else_label, else_exp]
                end)
        | (Nx _, Nx _) =>
          let val joinLabel = Temp.newlabel() in
            Nx (seq [(cond) (then_label, else_label),
                   LABEL then_label, unNx then_e,
                   JUMP (NAME joinLabel, []),
                   LABEL else_label, unNx else_e,
                   LABEL joinLabel])
          end
        | (Ex _, Cx _) => tr_if e1 then_e (Ex (unEx else_e))
        | (Ex _, Nx _) => tr_if e1 (Nx (unNx then_e)) else_e
        | (Cx _, Ex _) => tr_if e1 (Ex (unEx then_e)) else_e
        | (Cx _, Nx _) => tr_if e1 (Nx (unNx then_e)) else_e
        | (Nx _, Ex _) => tr_if e1 then_e (Nx (unNx else_e))
        | (Nx _, Cx _) => tr_if e1 then_e (Nx (unNx else_e))
    end
```

Conditional

```
tr_if  : tree_rep
          -> tree_rep
          -> tree_rep
          -> tree_rep
```

```
trans (If(e1, e2, e3)) =
   tr_if (trans e1)
         (trans e2)
         (trans e3)
```

See also discussion on pages 161-162 of Appel's book (ML version).

This code could be improved!!!

squares.slang, again

```
let n : int = 10
in
   let x : ref int = ref 1
   in
      while n >= !x do
      begin
        print (!x * !x);
        set x := !x + 1
      end
   end
end
```

Front end + `trans`

```
EXP(ESEQ(MOVE(TEMP n, CONST 10),
        ESEQ(MOVE(TEMP x, CONST 1),
        ESEQ(LABEL _L0;
             CJUMP(GE, TEMP n, TEMP x, _L1, _L2);
             LABEL _L1;
             EXP(ESEQ(EXP(CALL(NAME print, [BINOP(MUL,TEMP x, TEMP x)])),
                 ESEQ(MOVE(TEMP x, BINOP(PLUS,TEMP x, CONST 1)),  CONST 0)));
                      JUMP(NAME _L0);
                      LABEL _L2,  CONST 0))))
```

Perhaps we can simplify this on another pass . . .