Translation into Intermediate Code

Mooly Sagiv
Tel-Aviv University
Schriber 317
Phone (03)640-7606
sagiv@math.tau.ac.il

December 10, 2000

1 Subjects

- L-value vs. R-value
- Why use intermediate language?
- A Tree Intermediate language
- Examples
- Translation into Trees
  - Value computation vs. location computation
  - Simple Variables (including static links)
  - Arrays and structures
  - Run-Time Checks
  - Arithmetic
  - Conditionals
  - Strings
  - Record and array creation
  - Loops
  - Function calls
  - Variable Definition
  - Function Definition
  - Fragments

2 L-values vs. R-values

- Assignment $x := exp$ is compiled into:
  1. Compute the **address** of x
  2. Compute the **value** of exp
  3. Store the value of exp into the address of x
- Generalization

  R-value

$$\begin{aligned} \text{r-val}(x) &= \text{value of } x \\ \text{r-val}(5) &= 5 \\ \text{r-val}(x+y) &= \text{r-val}(x) + \text{r-val}(y) \end{aligned}$$

  L-value

$$\begin{aligned} \text{l-val}(x) &= \text{address of } x \\ \text{l-val}(5) &= \textbf{undefined} \\ \text{l-val}(x+y) &= \textbf{undefined} \\ \text{l-val}(\textbf{C}-\textbf{array}a) &= \textbf{undefined} \\ \text{l-val}(\textbf{C}-\textbf{pointer}a) &= \text{address of } a \\ \text{l-val}(\textbf{Pascal}-\textbf{array}a) &= \text{base address of } a \\ \text{l-val}(a[i]) &= \text{l-val}(a) + \text{r-val}(i) \end{aligned}$$

3 Why use intermediate languages

- Simplifying the compilation phase
  - ultimately leads to a more efficient code
- Portability of the compiler front-end
- Reusability of the compiler back-end

4 IR Design Goals

- Convenient to generate IR from the source
- Convenient to generate machine code from IR
- Clear operational meaning

5 Intermediate Representation Trees

```
/* tree.h */
typedef struct T_stm_ *T_stm;
typedef struct T_exp_ *T_exp;
typedef enum {T_plus, T_minus, T_mul, T_div,
              T_and, T_or, T_lshift, T_rshift, T_arshift, T_xor } T_binOp ;
typedef enum {T_eq, T_ne, T_lt, T_gt, T_le, T_ge,
              T_ult, T_ule, T_ugt, T_uge} T_relOp;
struct T_stm_ { enum { T_SEQ, T_LABEL, T_JUMP, T_CJUMP, T_MOVE,
                       T_EXP } kind;
              union { struct {T_stm left, right;} SEQ;
                      Temp_label LABEL;
                      struct {T_exp exp; Temp_labelList jumps;} JUMP;
                      struct { T_relOp op; T_exp left, right; Temp_label _true, _false;} CJUMP;
                      struct {T_exp dst, src;} MOVE;
                      T_exp EXP; } u; };
struct T_exp_ {enum {T_BINOP, T_MEM, T_TEMP, T_ESEQ, T_NAME,
                      T_CONST, T_CALL } kind;
              union { struct {T_binOp op; T_exp left, right;} BINOP;
                      T_exp MEM;
                      Temp_temp TEMP;
                      struct {T_stm stm; T_exp exp;} ESEQ;
                      Temp_label NAME;
                      int CONST;
                      struct {T_exp fun; T_expList args;} CALL; } u; };
T_stm T_Seq(T_stm left, T_stm right);
T_stm T_Label(Temp_label);
T_stm T_Jump(T_exp exp, Temp_labelList labels);
T_stm T_Cjump( T_relOp op, T_exp left, T_exp right,
               Temp_label _true, Temp_label _false );
T_stm T_Move(T_exp, T_exp);
T_stm T_Exp(T_exp);
T_exp T_Binop(T_binOp, T_exp, T_exp);
T_exp T_Mem(T_exp);
T_exp T_Temp(Temp_temp);
T_exp T_Eseq(T_stm, T_exp);
T_exp T_Name(Temp_label);
T_exp T_Const(int);
T_exp T_Call(T_exp, T_expList);
typedef struct T_expList_ *T_expList;
struct T_expList_ {T_exp head; T_expList tail;};
typedef struct T_stmList_ *T_stmList;
struct T_stmList_ {T_stm head; T_stmList tail;};
T_expList T_ExpList (T_exp head, T_expList tail);
T_stmList T_StmList (T_stm head, T_stmList tail);
```

6 Shorthands

$$
\begin{aligned}
TEMPr &\equiv TEMP(r) \\
CONST7 &\equiv CONST(7) \\
+(x,y) &\equiv BINOP(T_PLUS, x, y) \\
SEQ(a,b,c) &\equiv SEQ(a, SEQ(b,c))
\end{aligned}
$$

7 Tiger Program

```
let
   function nfactor   (n: int): int =
                            if   n = 0
                                                then 1
                                 else n * nfactor(n-1)
in
    nfactor(10)
end
```

8 Abstract Tiger Program

```
 letExp(decList(
   functionDec(fundecList(
     fundec(nfactor,
      fieldList(
       field(n, int,
        fld-escaped=FALSE),
       fieldList()),
      int,
      ifExp(
       opExp(EQUAL, varExp(simpleVar(n)), intExp(0)),
       intExp(1),
       opExp(TIMES, varExp(simpleVar(n)),
         callExp(nfactor,
         expList(opExp(MINUS, varExp(simpleVar(n)),
           intExp(1)),
          expList())))))),
     fundecList())),
   decList()),
  seqExp(expList(
```

```
    callExp(nfactor, expList(intExp(10), expList())),
    expList())))
```

9 Compiling Main

```
        .globl  t_main
        .ent    t_main
t_main_framesize=36
       .frame   $sp,t_main_framesize,$31
t_main:
       addiu    $sp,$sp,-t_main_framesize  # open frame
L4:
       sw   $2,0+t_main_framesize($sp)       # save static link
       or   $25,$0,$31
       sw   $25,-4+t_main_framesize($sp)     # save caller ra
       addiu    $25,$sp,0+t_main_framesize
       or   $2,$0,$25                        # calculate static link
       addi $25,$0,10
       or   $4,$0,$25                        # arg1=10
       jal  nfactor
       lw   $25,-4+t_main_framesize($sp)
       or   $31,$0,$25                       # restore caller ra
       b    L3
L3:
       addiu    $sp,$sp,t_main_framesize      # close frame
       j    $31
       .end t_main
```

10 Compiling `nfact`

```
        .globl  nfactor
        .ent    nfactor
        nfactor_framesize=40
        .frame  $sp,nfactor_framesize,$31
nfactor:
        addiu   $sp,$sp,-nfactor_framesize   # open frame
L6:
        sw  $2,0+nfactor_framesize($sp)    # save static link
        or  $25,$0,$4                      # save arg1
        or  $24,$0,$31
        sw  $24,-4+nfactor_framesize($sp)  # save caller ra
        sw  $30,-8+nfactor_framesize($sp)  # save prev n
        beq $25,$0,L0                     # n = 0?
L1:
```

```
        or  $30,$0,$25
        lw  $24,0+nfactor_framesize($sp)
        or  $2,$0,$24                       # calculate static link
        addi    $25,$25,-1                    # n-1
        or  $4,$0,$25                       # arg1 = n-1
        jal nfactor
        or  $25,$0,$2                       # r25= (n-1)!
        mult    $30,$25                       # r30=n(n-1)!
        mflo    $30
L2:
        or  $2,$0,$30                       # ret = r30
        lw  $30,-4+nfactor_framesize($sp)
        or  $31,$0,$30                      # restore caller ra
        lw  $30,-8+nfactor_framesize($sp)  # restore prev n
        b   L5
L0:     addi    $30,$0,1                      # ret = 1
        b   L2
L5:     addiu   $sp,$sp,nfactor_framesize     # close frame
        j   $31
        .end    nfactor
```

11 IR for Main

```
/* prologue of main */

/* body of main */
   SEQ(
    MOVE(
     TEMP t103,
     CALL(
      NAME nfactor,
       TEMP t127,
       CONST 10)),

/* epilogue of main */
```

12 IR for `nfactor`

```
/* prologue of nfunc */
```

```
   MOVE(
    TEMP t128,
    TEMP t104)),


/* body of nfactor */
   SEQ(MOVE(
     TEMP t103,
     ESEQ(CJUMP(EQ, TEMP t128, CONST 0, L0, L1),
           ESEQ(LABEL L0,
            ESEQ(MOVE(TEMP t129, CONST 1),
                    ESEQ(JUMP(NAME L2),
                          ESEQ(LABEL L1,
                             ESEQ(MOVE(TEMP t129,
                                       BINOP(TIMES,
                                             TEMP t128,
                                             CALL(NAME nfactor,
                                                  MEM(BINOP(PLUS,
                                                            TEMP t127,
                                                            CONST 0)),
                                                  BINOP(MINUS,
                                                        TEMP t128,
                                                        CONST 1)))),
                                  ESEQ(LABEL L2,
                                       TEMP t129)))))))),
/* epilogue of nfactor */
```

13 Example Tiger Program ‘‘nested.tig’’

```
let
   function fun1():int =
            let var d:=0
            function fun2():int =d+1
            in
               fun2()
            end
in
  fun1()
end
```

14 ‘‘nested.tig’’
t_main

```
⋮
SEQ(
    MOVE(
        TEMP t103,
        CALL(
            NAME fun1, TEMP t127)),
⋮
)
```

15 ‘‘nested.tig’’
fun1

```
⋮
SEQ(
    ESEQ(
        MOVE(
            MEM(
                BINOP(PLUS,
                    TEMP t127,
                    CONST -4)),
                CONST 0),
            CALL(
                NAME fun2,
                TEMP t127)),
    )
⋮
```

16 ‘‘nested.tig’’
fun2

```
⋮
SEQ(
    MOVE(
        TEMP t103,
        BINOP(
            PLUS,
            MEM(
                BINOP(PLUS,
```

```
                                MEM(
                                    BINOP(PLUS,
                                          TEMP t127,
                                          CONST 0)),
                                CONST -4)),
                        CONST 1)),
    )
⋮
```

17 Outline of the Translation `translate.c`

- Top-down traversal over the abstract syntax tree
- Generate code to allocate memory for declarations and initializations
- Generate code for function declarations:
  - Prologue
  - The body expression
  - Epilogue
- Generate code for expressions
  - Value expressions $x + y$
  - Location expressions $x < y$
  - Statement $x := y$, `while`-statement

18 Top Level Invocation `semant.c`

```
static struct expty transExp(Tr_level level,
                             S_table env,
                             S_table tenv,
                             A_exp a);
static Tr_exp   transDec(Tr_level level,
                         S_table env,
                         S_table tenv,
                         A_dec dec);

static Ty_ty    actual_ty(Ty_ty ty);


struct expty
```

```
    Tr_exp          exp;
    Ty_ty           ty;
;
```

19 Declarations

- Call `allocLocal` to allocate new places in the frame
- Generate invocations to `malloc` to allocate new places in the heap
- Generate initialization code
- For functions and type declarations use "no-ops', e.g.,

  CONST(0)

20 Example: Tiger Record Allocation

Source

```
type foo =  ty1 a, ty2 b
... =  foo a =e1, b = e2
```

Intermediate Representation

ESEQ (SEQ (MOV(TEMP r, CALL(NAME MALLOC, CONST 2*W)),
SEQ(MOV(MEM(+(0*W, TEMP r)), (TransExp(e_1))),
MOV(MEM(+(1*W, TEMP r)), (TransExp(e_2)))))),
TEMP r)

21 Example: Tiger Array Allocation

Source

```
let
  type intArray = array of int
  var a := intArray[12] of 0
  var b := intArray[13] of 7
in a := b
```

Intermediate Representation

```
SEQ(
SEQ(CONST 0,
SEQ(MOVE(TEMP ta, CALL(NAME initArray, CONST 12, CONST 0)),
SEQ(MOVE(TEMP tb, CALL(NAME initArray, CONST 13 CONST 7))
 MOVE(TEMP ta, TEMP tb)))))
```

22 Handling Functions

prologue

- pseudo-instructions to announce the beginning of the function
- a label for the beginning of the code
- An instruction to adjust the stack frame
- Save "escaping" arguments into the frame and move other arguments into registers
- Save callee-save registers

body `trasExp`

epilogue

- Move the return value into a register `RV` (update `Frame`)
- Restore callee-saved registers
- Reset stack-pointer
- return instruction
- pseudo-instructions to announce the end of the function

23 Expressions and Statements

- Recursive traversal of the abstract expression tree
- Use `frame` functions to manipulate the frame
- Combine Values and Location Computations
- Generate run-time tests if necessary

24 L-Values of simple variables

- For a local variable v

$$MEM(+CONSTk, TEMPfp)$$

- For a non-local variable v

MEM(+ (CONST k_n,
MEM(+ (CONST k_{n-1},
$\cdots$, MEM(+ CONST k_1, TEMP fp)))))

25 "Big" Values

- In some programming languages, more than one word need to be copied or stored
- Examples: C structures, Pascal arrays
- Extend the intermediate language

  T_exp T_Mem(T_Exp, int size)

- The l-value of a local variable

  MEM(+(CONST k_n, TEMP fp), S)

- Every subscript yields another expression
- l-values are converted into r-values using `MEM`
- In some cases, a more efficient code can be generated, e.g., for static arrays

26 Arrays and Structures in Tiger

- The l-value of $a[i]$

  MEM(+("a", *(CONST W, "i")))

- For a structure $s.f$,

  MEM(+("s", *(CONST W, CONST k_f)))

27 Safety

- Compiler cannot always check array references
- Solutions:

  Pascal, ML, Java, PL1
  - Generate code to check array references
  - Optimizers can reorder or remove many of these checks

  C No checks

  ADA Pragmas determine the checking level

28 Value Computation vs. Location Computation

Value-Computation

source

x * 7

Target expression

```
BINOP (MUL,
        MEM(BINOP(PLUS, TEMP fp, CONST(12))),
        CONST(7) )
```

Location-Computation

source

x > 7

Target statement

```
CJUMP(GT,
        MEM(BINOP(PLUS, TEMP fp, CONST(12))),
        CONST(7),
        t,
        f)
```

29 C Code

```
/* translate.h */
typedef struct Tr_exp_ *Tr_exp;
⋮
/* translate.c */
struct Cx {patchList trues; patchList falses; T_stm stm;};
/* expression, statement, or conditional */
typedef enum {Tr_ex, Tr_nx, Tr_cx} Tr_ex_kind;
struct Tr_exp_
        { Tr_ex_kind kind;
        union {
                T_exp ex;
                T_stm nx;
                struct Cx cx;
                } u;
        };
static Tr_exp Tr_Ex(T_exp ex);
static Tr_exp Tr_Nx(T_stm nx);
static Tr_exp Tr_Cx(patchList trues, patchList falses, T_stm stm);
```

30 C Code(semant.c)

```
/* translate.h */
Tr_Exp Tr_SimpleVar(Tr_Access, Tr_Level);
/* semant.c */
#include "translate.h"
static Tr_exp transDec(Tr_level level,S_table env, S_table tenv, A_dec dec)
{ switch (dec→kind) {
        ⋮
        case A_varDec: {
            ⋮
            Tr_access access = Tr_allocLocal(level, TRUE);
            ⋮
            S_enter(env, dec→u.var.var, E_VarEntry(access, ty));
            ⋮
            } } }
static struct expty transVar(Tr_level level,
                             S_table env, S_table tenv, A_var v)
{ switch (v→kind) {
        case A_simpleVar: {
            E_enventry x = (E_enventry) S_look(env, v→u.simple);
            if (x && x→kind == E_varEntry)
              return expTy(Tr_simpleVar(x→u.var.access, level),
                           actual_ty(x→u.var.ty));
              ⋮
```

31 C Code(translate.c)

```
static T_exp simpleVar(Tr_access access, Tr_level level)
{
  T_exp exp = staticLink(access→level, level);
  return F_Exp(access→access, exp);
}
Tr_exp Tr_simpleVar(Tr_access access, Tr_level level)
{
  return Tr_Ex(simpleVar(access, level));
}
```

32 Translating of Expressions

- Straightforward by induction on the abstract expression

- C code for Binary expressions

```
/* translate.c */
Tr_exp Tr_opExp(A_oper oper, Tr_exp left, Tr_exp right)
{
 switch (oper) {
         case A_plusOp: return Tr_opArithExp(T_plus,left,right);
         case A_minusOp: return Tr_opArithExp(T_minus,left,right);
         case A_timesOp: return Tr_opArithExp(T_mul,left,right);
         case A_divideOp: return Tr_opArithExp(T_div,left,right);
         case A_eqOp: return Tr_opCondExp(T_eq,left,right);
         case A_neqOp: return Tr_opCondExp(T_ne,left,right);
         case A_ltOp: return Tr_opCondExp(T_lt,left,right);
         case A_leOp: return Tr_opCondExp(T_le,left,right);
         case A_gtOp: return Tr_opCondExp(T_gt,left,right);
         case A_geOp: return Tr_opCondExp(T_ge,left,right);}
 assert(0); return NULL;}
static Tr_exp Tr_opArithExp(T_binOp oper, Tr_exp left, Tr_exp right)
{
  return Tr_Ex(oper, left, right);
}
```

33 Expressions with No Values

- Input

  $x := exp$

- Generated Code

  MOV("x", Code for exp)

- C-code

```
Tr_exp Tr_assignExp(Tr_exp leftExp, Tr_exp rightExp)
{
  return Tr_Nx(T_Move(leftExp, rightExp)) ;
}
```

34 Conditional Expressions

- Input

  if $a > b$ then $x := 5$

- Generated Code

  SEQ(
  CJUMP(GT, “a”, “b”, t, f),
  LABEL(t),
  Code for $x := 5$,
  LABEL(f))

- C-code

```
static Tr_exp Tr_opCondExp( T_relOp oper, Tr_exp left, Tr_exp right)
{
  struct Cx cx;
  cx.stm = T_Cjump(oper, left, right, NULL, NULL);
  cx.trues = PatchList(&cx.stm→u.CJUMP._true, NULL);
  cx.falses = PatchList(&cx.stm→u.CJUMP._false, NULL);
  return Tr_Cx(cx.trues, cx.falses, cx.stm);
}
```

35 Converting Representations

- input

  $flag := a > b$

- Generated Code

```
MOVE(
        “flag”,
        ESEQ(MOVE(TEMP r, CONST 1),
                ESEQ(CJUMP(GT, “a”, “b”, t, f),
                        ESEQ(LABEL(f),
                                ESEQ(MOVE(TEMP r, CONST 0),
                                        ESEQ(LABEL(t),
                                                TEMP r ) ) ) ) ) )
```

36 C code (Convertion into Values)

```
static T_exp unEx(Tr_exp e)
{
  switch (e→kind) {
        case Tr_ex:
              return e→u.ex;
        case Tr_cx: {
              Temp_temp r = Temp_newtemp();
```

```
                Temp_label _true = Temp_newlabel(),
                           _false = Temp_newlabel();
                doPatch(e→u.cx.trues,_true);
                doPatch(e→u.cx.falses,_false);
                return T_Eseq(T_Move(T_Temp(r), T_Const(1)),
                        T_Eseq(e→u.cx.stm,
                        T_Eseq(T_Label(_false),
                        T_Eseq(T_Move(T_Temp(r), T_Const(0)),
                        T_Eseq(T_Label(_true),
                        T_Temp(r))))))); }
          case Tr_nx:
                        return T_Eseq(e→u.nx, T_Const(0));
  }
  assert(0); return NULL;
}
```

37 C code (Convertion into Place)

```
static struct Cx unCx(Tr_exp e)
{
 switch (e→kind) {
        case Tr_ex: {
              struct Cx cx;
              cx.stm = T_Cjump(T_ne, e→u.ex, T_Const(0), NULL, NULL);
              cx.trues = PatchList(&cx.stm→u.CJUMP._true, NULL);
              cx.falses = PatchList(&cx.stm→u.CJUMP._false, NULL);
              return cx; }
        case Tr_cx: return e →u.cx;
        case Tr_nx: assert(0);
 }
 assert(0); {struct Cx cx; return cx;} }
```

38 If-Then-Else Expressions

- A Tiger expression $x < 5$ is translated into:

  s(t,f) = CJUMP(LT, "x", 5)

- Inductive translation of "`if` e_1 `then` e_2 `else` e_3"
  - Convert e_1 into $Cx(t, f)$
  - Convert e_2 and e_3 into Ex
  - Allocate a temporary r
  - Generate a label l

  - In t-branch generate code to set r to e_1 and jump into l
  - In f-branch generate code to set r to e_2 and jump into l
  - r is the return value
- May be very inefficient, e.g., `if` $x < 5$ `then` $a > b$ `else` 0

39 Loops

- `while` e `do` e_1

  test:
  if not e goto done
  e_1
  goto test
  done

- Need to handle break, continue
- For loops can be translated into a while

40 Summary

- Intermediate code simplifies the translation and increases re-use
- Tree-like intermediate code simplifies the translation of expressions
- Abstract syntax helps
- Runtime memory management is interesting