Intermediate Code

Lecture 7
IR Types

structure Tree : TREE =
struct
    type label = Temp.label   type size = int
datatype exp
    = BINOP of binop * exp * exp
    | ESEQ of stm * exp
    | NAME of label
    | CONST of int
    | CALL of exp * exp list
    | FETCH of lexp
and lexp
    = MEM of exp
    | TEMP of Temp.temp
and stm
    = SEQ of stm * stm
    | LABEL of label
    | JUMP of exp * label list
    | CJUMP of relop * exp * exp * label * label
    | MOVE of lexp * exp
    | EXP of exp
and binop = PLUS | MINUS | MUL | DIV
    | AND | OR | LSHIFT | RSHIFT | ARSHIFT | XOR
and relop = EQ | NE | LT | GT | LE | GE
    | ULT | ULE | UGT | UGE
end (* structure Tree *)
Expressions

exp:

- **CONST i**: integer constant `i`
- **NAME lab**: symbolic label `lab` (assembly label)
- **BINOP(o,e1,e2)**: binary operator application
- **CALL(f, args)**: function/procedure call
- **ESEQ(s, e)**: sequence of stmt `s` and exp `e`
- **FETCH(le)**: fetch from memory or register

lexp:

- **MEM(e)**: location at address `e`
- **TEMP t**: temporary; a virtual register
Statements

stm:

MOVE(TMP t, e)  load value of e into t

MOVE(MEM e₁, e₂) store value of e₂ into address e₁

EXP(e)  evaluate e and discard result

JUMP(e,labs)  jump to label denoted by e, which must be in list labs

CJUMP(o,e₁,e₂,t,f)  evaluate relational op o on values of e₁,e₂; jump to label t or f depending on result

SEQ(s₁,s₂)  stm s₁ followed by s₂

LABEL(n)  define n as label of current address
l-values and r-values

lexp denote locations where values can be stored (also known as l-values)

MEM of exp location at address e

TEMP temp temporary; a virtual register

These can be used as targets of a MOVE stm representing either storing at a memory location or loading a register.

To use an lexp in an expression, it must be transformed into an r-value by applying FETCH.

FETCH(MEM(e)) contents of location at address e

FETCH(TEMP(t)) contents of register t
signature TRANSLATE =
sig

  val transExp : TREnv.env * TransUtil.loopend option
               * TransUtil.level * TransUtil.compilation
               -> Absyn.exp -> TransUtil.gexp

  val transDecs : TREnv.env * TransUtil.loopend option
                 * TransUtil.level * TransUtil.compilation
                 -> Absyn.dec list
                 -> TREnv.env * TransUtil.gexp list

  val transProg : Absyn.exp -> TransUtil.Frame.frag list

end (* signature TRANSLATE *)
Translation Arguments

transExp :

Context arguments:
- TREnv.env: translation environment
- TU.loopend option: label for BREAK to go to (if in loop)
- TU.level: chain of statically nested frames (for computing static links)
- TU.compilation: container to store code fragments

Expression:
- Absyn.exp: expression to translate

Result:
- TU.gexp: unified IR expressions

transDec :
- similar, but takes Absyn.dec and produces gexp list
structure TREnv : TR_ENV =
struct
  (* map identifiers to access info instead of types *)

  type access = TransUtil.access
  type level = TransUtil.level
  type label = Temp.label

  datatype enventry
    = VARentry of {access: access}
    | FUNentry of {level: level, label: label}

  type env = enventry Symbol.table

  val base_env = ... (* environment initialization *)
end (* structure TREnv *)
Translation Info

The defn of TREnv.env uses the following info:

```plaintext
datatype level =                         (* TransUtil *)
    Level of {frame: Frame.frame,
              parent: level option,
              id : unit ref}

type access = level * Frame.access     (* TransUtil *)

type label = Temp.label
```

`level` represents a chain of frame info for the current function and statically enclosing functions. This is used to generate expressions to compute static links (given a base level and destination level).

`access` represents info for accessing local or nonlocal variables (while Frame.access is for local access relative to a frame). Used to generate code to access the value of local or nonlocal variables.

`label` identifies the code for a particular function
Fragments, Compilations

Code for each function body and for the top-level program is collected in separate *fragments*. Additional fragments are created for each string literal.

```
datatype frag    (* Frame.frag *)
    = PROC of {frame: frame, body: Tree.stm}
    | STRING of Temp.label * string
```

The function body code is combined with the associated frame information. These fragments will be stitched together to create the final code later.

STRING fragments represent data for string literals, with labels for accessing them.

Fragments are collected in a compilation, which is a ref to a list of fragments:

```
type compilation = Frame.frag list ref  (* TransUtil *)
```
Unifying Expressions

The IR (Tree) language is broken down into three types:

\[ \text{exp}, \text{lexp}, \text{stm} \]

We define a single type `gexp` to unify `exp` and `stm`, plus a separate form to handle conditionals (`T = \text{Tree}`):

```
datatype gexp
    = Ex of T.exp (* value-carrying expression *)
    | Nx of T.stm (* value-less expression *)
    | Cx of Temp.label * Temp.label -> T.stm (* conditional builder *)
```

Our utility functions for translating various syntax forms will generally build gexps.

Conditionals are represented by functions from a pair of true and false destination labels to a stm.

NOTE: `gexp` is called `exp` in Appel
Conditionals in gexp

\( a < b \)

\[ \text{gexp} = \text{Cx}(\text{fn} (t,f) \Rightarrow \text{CJUMP}(\text{GT},a,b,t,z)) \]

\( a < b \mid c < d \)

\[ \text{gexp} = \text{Cx}(\text{fn} (t,f) \Rightarrow \text{SEQ} (\text{CJUMP}(\text{GT},a,b,t,z), \text{SEQ}(\text{LABEL} z, \text{CJUMP}(\text{LT},c,d,t,f)))) \]

where \( z \) is a new label
gexp conversions

In different circumstances, we will need to treat an arbitrary gexp as an exp, a stm, or a conditional building function. We define three conversion functions:

\[
\begin{align*}
\text{unEx: } & \text{gexp} \rightarrow \text{exp} \quad (* \text{convert to Tree.exp} *) \\
\text{unNx: } & \text{gexp} \rightarrow \text{stm} \quad (* \text{convert to Tree.stm} *) \\
\text{unCx: } & \text{gexp} \rightarrow (\text{label} \times \text{label} \rightarrow \text{stm}) \\
& \quad (* \text{interpret as condition fn} *)
\end{align*}
\]

For example, if we have a conditional (Cx form) as the right-hand side of an assignment, unEx will convert it to an appropriate form:

\[
x := (a > b)
\]

\[
\text{MOVE} (\text{TEMP}_x, \text{unEx(gexp)})
\]

where \( gexp = \text{Cx}(\text{fn} \ (t,f) \Rightarrow \text{CJUMP}(\text{GT},a,b,t,z)) \)
fun unEx (Ex e) = e \quad (*\text{strip off} \ \text{Ex constructor} *) \\
| unEx (Nx s) = T.\text{ESEQ}(s, T.\text{CONST 0}) \\
\quad (*\text{execute} \ s \ \text{and then return dummy value} \ 0 *) \\
| unEx (Cx stmfn) = \\
\quad \begin{array}{l}
\text{let} \ \text{val} \ r = \text{Temp.newTemp()} \quad (*\text{temp for result} *) \\
\text{val} \ t = \text{Temp.newLabel()} \quad (*\text{label for true branch} *) \\
\text{val} \ f = \text{Temp.newLabel()} \quad (*\text{label for false branch} *) \\
\text{in} \ T.\text{ESEQ}(\text{seq}[T.\text{MOVE}(T.\text{TEMP} \ r, T.\text{CONST} \ 1),
\quad \text{stmfn}(t,f),
\quad T.\text{LABEL} \ f,
\quad T.\text{MOVE}(T.\text{TEMP} \ r, T.\text{CONST} \ 0),
\quad T.\text{LABEL} \ t],
\quad T.\text{TEMP} \ r)
\end{array}
\end{array}
end

In the Cx case, the resulting code will return 1 or 0 according to whether the condition takes the true branch or false branch.
The other two conversion functions are simpler:

```plaintext
fun unNx (Nx s) = s
    | unNx (Ex e) = T.EXP e
    | unNx (Cx genstm) = 
        let val l = Temp.newlabel ()
        in seq [genstm (l, l), T.LABEL l]
        end

fun unCx (Cx stmfn) = stmfn
    | unCx (Ex (T.CONST 0)) = 
        (fn (t, f) => T.JUMP (T.NAME f, [f]))
    | unCx (Ex (T.CONST _)) = 
        (fn (t, f) => T.JUMP (T.NAME t, [t]))
    | unCx (Ex e) = 
        (fn (t, f) => T.CJUMP (T.EQ, e, T.CONST 0, f, t))
    | unCx (Nx s) = ErrorMsg.impossible "unCx (Nx s)"
```
Translating SimpleVar\{name,pos\}
with:  current level = level0,
      environment = env

1. Look up name in env to get VARentry(access)
   where access : TransUtil.access = level \times Frame.access,
   so access = (level_var, faccess)

2. Use:  TransUtil.framePtr(level_var,level0)
   to compute an expression sl\_exp for the static link.
   If level_var = level0 (i.e. the variable is local), then
   sl\_exp will be TEMP(Registers.FP).

3. Use:  Frame.accessToExp faccess sl\_exp
   to compute the final access exp.

   \[
   \text{fun accessToExp(InFrame n) sl\_exp =}
   \begin{align*}
   & \quad \text{T.MEM(T.BINOP(T.PLUS,sl\_exp,T.CONST n))} \\
   | & \quad \text{accessToExp(InReg t) sl\_exp =} \\
   & \quad \text{T.TEMP t}
   \end{align*}
   \]
Computing Static Link Expressions

We are using the simplifying assumption that every function call will pass a static link as the first argument, and that this implicit static link argument will be escaping. So the static link will always be found at 0($fp$).

```
function f(x: int) =                (* level1 *)
    let function g(y: int) =          (* level2 *)
        let function h(z: int) =    (* level3 *)
            (x + y + z)
            in h(0)
        end
    in g(1)
end

look(env,x) => VARentry(level1,InFrame(1))
```

```
sl_exp = FETCH(MEM(+CONST 0,
          FETCH(MEM(+CONST 0,
                  FETCH(TEMP(R.FP))))))
level 1 fp

x_exp = FETCH(MEM(+sl_exp, CONST 1))
```

level 2 fp

level 3 fp
Variable Access via Static Links

\[
\text{mem}_x = (1 + \text{fetch}(0 + \text{fetch}(0 + \text{fetch}(\text{fp}))))
\]
Record and Array Access

(* recordField: lexp * int -> lexp *)
(* used to translate RecordVar *)
(* the index is statically known *)

fun recordField (rlexp, index) =
    T.MEM (T.BINOP (T.PLUS,
        T.FETCH rlexp,  (* base addr of record *)
        T.CONST (index * Frame.wordSize)))

(* subscript: lexp * gexp -> lexp *)
(* used to translate ArrayVar *)
(* the index is computed by an index expression *)

fun subscript (alexp, index_exp) =
    (* without bounds checking *)
    T.MEM (T.BINOP (T.PLUS,
        T.FETCH alexp,  (* base addr of array *)
        T.BINOP (T.LSHIFT,  (* multiply by 4 *)
            T.BINOP (T.LSHIFT,  (* multiply by 4 *)
                unEx index_exp,
                T.CONST Frame.wordSizeLog))))
Conditionals

(* statement branches *)

type condgen = Temp.label * Temp.label -> Tree.stm

fun IfStm (testgen: condgen, thenStm: T.stm, elseStm: T.stm) =
  let val l1 = Temp.newlabel ()
    val l2 = Temp.newlabel ()
    val l3 = Temp.newlabel ()
    in Nx (seq [testgen (l1, l2),
               T.LABEL l1,
               thenStm,
               T.JUMP (T.NAME l3, [l3]),
               T.LABEL l2,
               elseStm,
               T.LABEL l3])
  end
fun IfExp (testgen: condgen, thenExp: T.exp, elseExp: T.exp) =
let val testgen = unCx teste
  val r = T.TEMP (Temp.newtemp ())
  val l1 = Temp.newlabel ()
  val l2 = Temp.newlabel ()
  val l3 = Temp.newlabel ()

  in Ex (T.ESEQ (seq [testgen (l1, l2),
       T.LABEL l1,
       T.MOVE (r, thenExp),
       T.JUMP (T.NAME l3, [l3]),
       T.LABEL l2,
       T.MOVE (r, elseExp),
       T.LABEL l3],
       T.FETCH r))
  end
Strings

Strings are represented by a word containing the length, followed by the characters of the string.

```
| 12 | Hello World
```

(* creating a string *)

```latex
fun String (comp, s) =
  let val l = Temp.newlabel ()
    in addFrag(comp,Frame.STRING (l, s));
      Ex (T.NAME l)
  end
```
Records

Creating records:
1. Move field values into new temps
2. Call runtime system function allocRecord with #fields
3. Store field values in slots of record

fun Record fieldexps = 
    let val rt = T.TEMP(Temp.newtemp())  (* address of new record *)
        fun acc i =
            T.MEM(T.BINOP(T.PLUS, T.FETCH rt, T.CONST(Frame.wordSize * i)))
        val fieldtemps = map (fn _ => T.TEMP(Temp.newtemp ())) fieldexps
        val tempInits =
            ListPair.map (fn (e, t) => T.MOVE (t, unEx e))
                (fieldexps, fieldtemps)
        fun fieldMoves (_, []) = []
            | fieldMoves (i, t::ts) =
                (T.MOVE (acc i, T.FETCH t))::fieldMoves(i+1,ts)
        val fieldInits = fieldMoves(0,fieldtemps)
        val recsize = length fieldexps * Frame.wordSize
        val recordAlloc =
            T.MOVE(rt, Frame.externalCall("allocRecord", [T.CONST recsize]))
    in Ex (T.ESEQ (seq (tempInits @ [recordAlloc] @ fieldInits),
                        T.FETCH rt))
    end
Function Calls

Function Call expression:

\[
\text{CALL}(\text{NAME } \text{flabel}, [\text{slexp}, \text{argexp}_1, \ldots, \text{argexp}_n])
\]

- \text{flabel} : the label for the function being called
- \text{slexp} : the static link
- \text{argexp}_i : the expressions for the normal arguments

fun \text{Call} \{f, \text{declared}, \text{current}, \text{el}\} =
  let val Level \{frame = curframe, \ldots\} = \text{current}
      val _ = \text{Frame.call} curframe (length \text{el})
          (* record \# of arguments passed, to be used
           * to determine frame size (maxcallargs). *)
      val \text{el} = \text{map unEx} \text{el} (* \text{args are expressions} *)
      val \text{args} = \text{framePtr}(\text{declared}, \text{current}) :: \text{el}
          (* add static link as implicit argument *)
  in \text{Ex} (\text{T.CALL} (\text{T.NAME} f, \text{args}))
  end
Function Definitions

Prolog:

1. pseudo instruction introducing function code (".text")
2. label definition for function name
3. decrement stack ptr to allocate frame
4. move arguments to frame (escaping) or temps
5. store instructions for callee-save registers (fp, ra)

Function body:

6. function body code

Epilog:

7. move return value to $v0 (if any)
8. load instructions to restore callee-saves regs
9. reset stack pointer to old value, popping frame
10. return instruction (jump to return address)