

Simple Variables...

Define the frame and level type for each function definition:

```
type frame = {name: Temp.label, formals: access list,
  locals : int ref} (in ?frame.sml)
datatype level = LEVEL of {frame : Frame.frame,
  parent : level} * unit ref
  | TOP
type access = level * Frame.access (in translate.sml)
```

```
val trSimpleVar = access * level -> exp
```

- The access information for a variable v is a pair (l,k) where l is the level in which v is defined and k is the frame access – in regs or frame.
- The frame offset \leq allocLocal function in the Frame structure (which is architecture-dependant). The access information will be put in the env in the typechecking phase (Semant calls this...).

- To access a local variable v at offset k , assuming the frame pointer is fp , just do

```
MEM(BINOP(PLUS, TEMP fp, CONST k))
```

- To access a non-local variable v inside function f at level lf , assuming v 's access is (lg,k) ; we do the following:

```
MEM(+ (CONST kn, MEM(+ (CONST kn-1, ... MEM(+ (CONST k1, TEMP fp) ...)))
```

- Strip levels from lf , we use the static link offsets $k1, k2, \dots$ from these levels to construct the tree. When we reach lg , we stop.

```
datatype level = LEVEL of {frame : frame,
  parent : level} * unit ref
  | TOP
```

- use “unit ref” to test if two levels are the same one.

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 the MEM operator
 - (MEM means fetch or store, depending on which side of MOVE is it on)

Array and Record Variables

- In Pascal, an array variable stands for the contents of the array --- the assignment will do the copying :

```
var a, b : array [1..12] of integer;
begin
  a := b
end;
```

- In Tiger and ML, an array or record variable just stands for the pointer to the object, not the actual object. The assignment just assigns the pointer.

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

- In C, assignment on array variables are illegal!

```
int a[12], b[12], *c; a = b; is illegal! c = a; is legal!
```

Array Subscription

- If a is an array variable represented as $\text{MEM}(e)$, then array subscription $a[i]$ would be (ws is the word size)
 $\text{MEM}(\text{BINOP}(\text{PLUS}, \text{MEM}(e), \text{BINOP}(\text{MUL}, i, \text{CONST } ws)))$
- To ensure safety, we must do the array-bounds-checking: if the array bounds are $L..H$, and the subscript is i ; then report runtime errors when either $i < L$ or $i > H$ happens.
- Array subscription can be either l-values or r-values --- use it properly.
- Record field selection can be translated in the same way. We calculate the offset of each field at compile time.

```
type point intlist = {hd : int, tl : intlist}
```

the offset for “hd” and “tl” is 0 and 4

Record and Array Creation

- Tiger record creation:

```
var z = foo {f1 = e1, ..., fn = en}
```

– we can implement this by calling the C malloc function, and then move each e_i to the corresponding field of foo .
- Tiger array creation:

```
var z = foo n of initv
```

– by calling a C `initArray(size,initv)` function, which allocates an array of size `size` with initial value `initv`.
– to support array-bounds-checking, we can put the array length in the 0-th field. $z[i]$ is accessed at offset $(i+1)*\text{word_sz}$
- Requirement: a way to call external C functions inside Tiger.
– `CALL(NAME(Temp.namedlabel("initArray")), [a,b])`

Integer and String

- Integer : $\text{absyn IntExp}(i) \Rightarrow \text{itree CONST}(i)$
- Arithmetic: $\text{absyn OpExp}(i) \Rightarrow \text{itree BINOP}(i)$
- Strings: every string literal in Tiger or C is the constant address of a segment of memory initialized to the proper characters.
- During translation from Absyn to itree, we associate a label l for each string literal s :
 - to refer to s , just use `NAME l`
- Later, we'll generate assembly instructions that define and initialize this label l and string literal s .
- String representations:
 - a word containing the length followed by characters (in Tiger)
 - a pointer to a sequence of characters followed by `\000` (in C)
 - a fixed length array of characters (in Pascal)

Conditionals

- Each comparison expression $a < b$ will be translated to a Cx generic expression
– $\text{fn } (t, f) \Rightarrow (\text{LT}, a, b, t, f)$
- Given a conditional expression (in absyn)

```
if e1 then e2 else e3
```

 1. translate e_1, e_2, e_3 into itree generic expressions e_1, e_2, e_3
 2. apply `unCx` to e_1 , and `unEx` to e_2 and e_3
 3. make three labels, then case: t and else case: f and join : j
 4. allocate a temporary r , after label t , move e_2 to r , then jump to j ; after label f , move e_3 to r , then jump to j
 5. apply `unCx`-ed version of e_1 to label t and f
- Need to recognize certain special case:
 $(x < 5) \ \& \ (a > b)$
it is converted to “if $x < 5$ then $a > b$ else 0” in absyn ----- too many labels if using the above algorithm --- inefficient.

Loops

Translating while loops:

```
test:
  if not (condition) goto done
  ... the loop body ...
  goto test
done:
```

- each round executes one conditional branch plus one jump

```
goto test
top:
  ... the loop body ...
test:
  if (condition) goto top
done:
```

- each round executes one conditional branch only

- Translating break statements:
 - just JUMP to done
 - need to pass down the label done of closest enclosing loop when translating the loop body!
 - Translating For loops:
 - for i := lo to hi do body
- ```
let var i:=lo
 var limit := hi
in while i <= limit
 do (body; i:=i+1)
end
```

## Function Calls

- Inside a function g, the function call f(e1,e2, ..., en) is translated into  
`CALL(NAME lf, [sl, e1, e2, ..., en])`
  - sl is the static link --- it is just a pointer to f's parent level
  - stripping the level of g one by one, generate the code that follow g's chains of static links until we reach f's parent level
  - lf is label for f
- When calling external C functions, what kind of static link do we pass ?
- In the future, we need to decide what is the calling convention ----- where the callee is expecting the formal parameters and the static link?

## Declarations

- Variable declaration: need to figure out the offset in the frame, then move the expression on the r.h.s. to the proper slot in the frame.
- Type declaration: no need to generate any itree code!
- Function declaration: build the PROC itree fragment
  - Later we translate PROC(name : label, body : stm, frame) to assembly:

```
_global name
name: prologue
 assembly code for body
 epilogue
```
- The prologue and epilogue captures the calling sequence, and can be figured out from the frame layout information in frame. Prologue and epilogue are often machine-dependant.

## Function Declarations

Generating prologue :

1. psuedo-instructions to announce the beginning of a function
2. a label definiton fo the function name
3. an instruction to adjust the stack pointer (allocating a new frame)
4. store instructions to save callee-save registers and return address
5. store instructions to save arguments and static links

Generating epilogue :

1. an instruction to move the return result to a special register
2. load instructions to restore callee-save registers
3. an instruction to reset the stack pointer (pop the frame)
4. a return instruction (jump to the return address)
5. psuedo-instructions to announce the end of a function

## More Arrays

- in Tiger scalars, wordsize addresses ok for arrays
- $a[i][j]$  -> simple math ok if 'dense' layout
  - consider rate at which index varies, cache behavior, parallel processing...
- vectors of vectors:
  - `char mons[12][9]={"January", "February", ...}`
  - `char *mons[9]={"January", "February", ...}`
- linked lists, realloc, extensible arrays...
- in general:
  - rank, element type, dimensions
  - if possible bounds for dimensions (would like to do runtime checking if possible)
  - could have all these be dynamic

A: Array[D1, D2, D3] of ElemType

$S0 = \text{sizeof}(\text{ElemType})$

$S1 = D3 * S0$  – row size

$S2 = D2 * S1$  – plane size

$S3 = D1 * S2$  – array size

$\text{offset}[i,j,k] = i * S2 + j * S1 + k * s0$

S3 and D1?  $a[[[10]]?$

in frame – array base or pointer to array base

## Records

- `struct {char s[10]; int j;} s;`
- if necessary pad.
- may rearrange fields
- move dynamic content at the end
- don't have to pack
  - tradeoff is speed vs. space

## Case statements

- if case1 then block1 else if case2 then block2....
  - linear time...
- jump 'case' label list..., i.e. jump table
  - ok if dense, so case can be used as index
  - ok if small waste in table
- for sparse tables

- binary search:
  - compare against median value.
  - log time
- partition into dense blocks
  - combo of jump tables and binary trees
- hash table