

- need to determine out locations of variables and addresses of functions
 - we have var and func names in symbol table
 - generate code will need memory addresses and registers
- frames have this info
 - need to keep track of SP and FP
 - need to keep track of nesting levels
- separate machine-dependent and –independent parts – use abstract frame type and abstract register names and address labels

Stack Frames in Tiger

- Using **abstraction** to avoid the machine-level details
- What do we need to know about each stack frame at compile time ?
 - 1) offsets of incoming arguments and the static links
 - 2) offsets of all local variables
 - 3) the frame size

```
signature FRAME =
sig
  . . . . .
end
structure MipsFrame : FRAME =
struct
  datatype access = InFrame of int | InReg of Temp.temp
  type frame = {name: Temp.label, formals: access list
                locals: int ref}
  . . . . .
end
structure SparcFrame : FRAME = . . . . .
```

Frames for Tiger

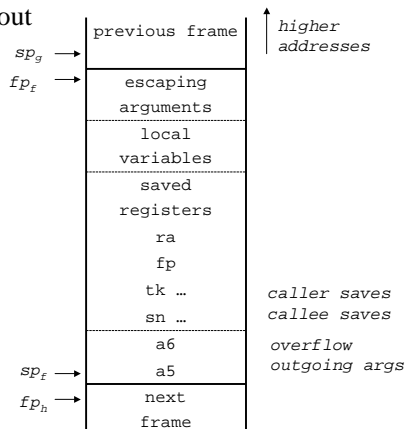
```
signature FRAME =
sig
  type frame
  type access (* InFrame or InReg *)
  val newFrame : {name: Temp.label, (*label of called fun*)
                 formals : bool list} -> frame
                (* formals specified with an escape flag *)
  val name : frame -> Temp.label
                (* the function label of a frame *)
  val formals : frame -> access list
                (* access info for formal parameters *)
  val allocLocal : frame -> bool -> access
                (* will go InReg or InFrame based on bool *)

  (* more to come ... *)
end (* signature FRAME *)
```

MIPS registers

Hardware	Name	Description
0	\$ zero	constant zero
1	\$ at	assembler temporary
2-3	\$ v0- \$ v1	function return value
4-7	\$ a0- \$ a3	incoming args
8-15	\$ t0- \$ t7	caller-saves temporaries
16-23	\$ s0- \$ s7	callee-saves temporaries
24-25	\$ t8- \$ t9	caller-saves temporaries
26-27	\$ k0- \$ k1	exception handling
28	\$ gp	global data pointer
29	\$ sp	stack pointer
30	\$ fp (\$s8)	frame pointer
31	\$ ra	return address

Frame Layout



Incoming Arguments

```
function f(x0,x1,x2,x3,y4,y5) =
  let function g(u: int): int =
    x0 + x2 + u /* x0 and x2 escaping */
  in g(3)
end
```

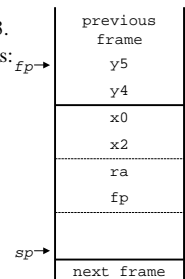
Arguments x_0, \dots, x_3 start in registers $\$a_0, \dots, \a_3 .

They are moved to temp registers or frame slots: $fp \rightarrow$

- $x_0:\$a_0 \Rightarrow 0(\$fp)$ -- escaping
- $x_1:\$a_1 \Rightarrow t17$
- $x_2:\$a_2 \Rightarrow -4(\$fp)$ -- escaping
- $x_3:\$a_3 \Rightarrow t18$

Arguments y_4, y_5 are stored in previous frame:

- $y_4: 4(\$fp)$
- $y_5: 8(\$fp)$



- Frame module (MipsFrame) will have to figure out how
 - parameters will be seen from inside the function – i.e. in register or in frame
 - to implement “view shift” – from what the caller saw to what the callee sees – i.e., update SP, FP, move escaping vars into Mm, assign temp registers, ...
 - m(x,y) { h(y,y), h(x,x) } -> have to move \$a0...
 - this is for later!

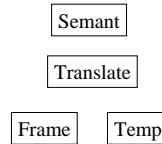
Computing Escaping Variables

- An escaping variable must be declared before the nested function that it appears in.
- The escaping property can be determined by comparing the function nesting depth of the variable’s declaration with the nesting depths of its applied occurrences.
- The escaping property is recorded in the escape field (a bool ref) of the abstract syntax construct that declares it.
 - VarDec for variable declarations
 - ForExp for for loop index variables
 - field record for function parameters
- An environment can be used to record function nesting depth and escape field ref for each variable within its scope (textbook!)

Registers and Addresses

- cannot determine actual values yet
- use abstract *names* - temporary registers and labels for addresses -> another environment
- virtual registers assigned uniquely from unlimited pool... actual assignments done later
- labels assigned uniquely – cannot generate from function name

- Frame and Temp hide machine detail for locations and registers
- Translate deals with nested scoping and translates names and addresses to correct address
- find escapes before semant



```

signature ENV =
sig
  datatype ententry =
    VarEntry of {access: Translate.access, ty: ty}
  | FunEntry of {level: Translate.level,
                 label: Temp.label,
                 formals: ty list,
                 result: ty}
..
end
  
```

```

structure TRANSLATE =
sig
  type level
  type access (* level * Frame.access,
              i.e., level * offset *)

  val outermost: level
  val newLevel: {parent: level, name: Temp.label,
                 formals: bool list} -> level
  val formals: level -> access list
  val allocLocal: level -> bool -> access
end

structure Translate: TRANSLATE = ...

datatype level = TOP or parent's level *Frame.frame* ref...
...
  
```

- TransDec for a function creates newLevel, this creates newFrame
- for a variable we need info which level is it declared at, and at what location:
Translate.access is level * Frame.access

Static Links

- Frame knows that it needs to save an escaping var
- Translate knows about nesting, and passes static link as first parameter with escape = true (SL will be in \$a0).
- Translate initiates SL chasing to get to frame where variable/label was declared