Activation Records

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

December 3, 2000

1 Tiger Program

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

2 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())))
```

3 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
```

4 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
```

5 Subjects

- Compile-Time Information on Variables
- Stack Frames
- Limitations
- The Frame Pointer
- The Static Pointers and Nesting Levels
- Registers
- Parameter Passing
- Return Address
- Memory Management in the Tiger Language

6 Compile-Time Information on Variables

- Name
- Type
- Scope — when is it recognized
- Duration — when does its value exist
- Size — How many bytes are required at run-time to represent the object at run-time
- Address (Fixed, Relative, Dynamic)

7 Stack Frames

		↑ higher addresses
	argument n	
	.	
incoming	.	previous
arguments	.	frame
	argument 2	
	argument 1	
frame pointer →	static link	
	local	
	variables	
	return address	
	temporaries	
		current
		frame
	saved	
	registers	
	argument m	
	.	
outgoing	.	
arguments	.	
	argument 2	
	argument 1	
stack pointer →	static link	
		next
		frame
		↓ lower addresses

8 80386 Pascal Stack Frames

		$\uparrow$ higher addresses
incoming arguments	argument 1 argument 2 . . argument . argument n static link	previous frame
	return address	
bp=frame pointer $\rightarrow$	previous bp	
	local variables	
	temporaries	current frame
outgoing arguments	argument 1 argument 2 . . argument . argument m static link	
	return address	
sp= stack pointer $\rightarrow$	previous bp	
		next frame $\downarrow$ lower addresses

9 The Frame Pointer

- The **caller** is the calling routine/function
- The **callee** is the called routine/function
- **caller** responsibilities:
    1. Calculate arguments and save in the stack
    2. Store static link

- **call** instruction: $M[--SP] := RA; PC := callee$
- **callee** responsibilities:
  1. $FP := SP$
  2. $SP := SP -$ frame-size

10 Example Variable Size Frames

```
program foo ;
const max = 4 ;
var m1, m2, m3: array [1..max, 1..max] of integer;
var i, j: integer;
procedure mult(a, b : array [1..l, 1..l] of integer;
               var c : array [1..l, 1..l] of integer);
var i, j, k: integer;
begin { mult }
    for i := 1 to l do
        for j := 1 to l do begin
            c[i, j] := 0 ;
            for k := 1 to l do
                c[i, j] := c[i, j] + a[i, k] * b[k, j]
        end
end; { mult}
begin { foo}
    for i := 1 to max do
        for j := 1 to max do begin
            m1[i, j] := i * j ;
            m2[i, j] := i + j
        end ;
    mult(m1, m2, m3)
end. { foo }
```

11 Limitations of Stack Frames

- A local variable of P cannot be stored in the activation record of P if its duration exceeds the duration of P
- Example 1: Static variables in C (own variables in Algol)

```
void p(int x)
{
  static int y = 6 ;
  y += x;
}
```

- Example 2: Features of the C language

```
int * f()
{
  int x ;
  return &x ;
}
```

- Example 3: Dynamic allocation

```
int * f()
{
  return (int *) malloc(sizeof(int));
}
```

12 Example 4: Higher Order Functions with Nesting

```
fun f(x) =
    let fun g(y) = x + y
        in g
end

val h = f(3)
val j = f(4)

val z = h(5)
val w = j(7)
```

13 Simplifying Assumption

No higher order functions and heap allocated storage

14 Static Scoping

- Languages such as Pascal, Scheme, ML, and Tiger allow nested scope rules
- Different implementations

static-pointer Whenever a function f is called from g it is passed an extra pointer to the beginning of the stack-frame of the most recent frame of the procedure h in which f is declared — The linked list is traversed to find the appropriate copy of the variable

display A global array d holds the nested frame-pointers of the active procedure p

- $d[i]$ is the beginning of the stack-frame at nesting-level i
- The display can be stored in the stack for a non-leaf procedure p

lambda-lifting Pass non-local variables as extra parameters

15 Example Tiger Program

```
type tree = { key: string, left: tree, right: tree }
function pretyprint(tree: tree): string =
        let
           var output := “”
           function write(s: string) =
                   output := concat(output, s)

           function show(n: int, t: tree) =
                   let function indent(s: string) =
                              (for i := 1 to n
                                  do write(“ ”);
                                  output := concat(output, s))
                      in if t= nil then indent(“.”)
                           else (indent(t.key);
                                show(n+1, t.left);
                                show(n+1, t.right))
                      end {show}
        in show(0, tree); output
end
```

16 Simplified Assumption

Only static-pointers is explained ignoring procedural parameters

17 Registers

- Machine registers are used for various tasks
- In RISC — all computations in registers
- **caller-save** vs. **callee-save**
  - Hardware support may be available
  - Different performance depending on architecture and usages

- Cases where no-save are necessary
  - "Dead" variables in registers before the call (caller-save)
  - Interprocedural analysis
- A register allocation algorithm is described in Chapter 11

18 Parameter Passing

1960s In memory — No recursion is allowed

1970s In stack — Slow

1980s In registers — First k parameters are passed in registers ($k = 4$ or $k = 6$)

Where is time saved?

1. Most procedures are **leaf** procedures
2. Interprocedural register allocation

   If f calls h, r_1 and r_2 may be allocated for f and r_3 to h
3. Even for a procedure f which is not a leaf, many of f's registers may be dead by the time it calls h
4. Register windows are allocated in some architectures per call (e.g., sun Sparc)

19 Modern Architectures

- **return-address** is also normally saved in a register on a call
  - a non leaf procedure saves this value on the stack
  - No stack support in the hardware
- **function-result** is also normally saved in a register on a call — a non leaf procedure saves this value on the stack

20 Frame-Resident Variables

A variable x of procedure p is stored in the stack when:

- x is passed by reference
- Address of x is taken (in C $\&x$)
- x is addressed via pointer arithmetic on the stack-frame (C varags)
- x is accessed from a procedure nested within p

- The value is too big to fit into a single register
- The variable is an array
- The register of x is needed for other purposes
- Too many local variables

An **escape** variable:

- Passed by reference
- Address is taken
- Addressed via pointer arithmetic on the stack-frame
- Accessed from a nested procedure

21 Calculating Escape(escape.c)

- Top down traversal of the tree
- Locate references non-local references variables
- Compares nesting levels of declarations and usages
- C code

```
/* local function prototypes */
static void traverseExp(S_table env, int depth, A_exp a);
static void traverseDec(S_table env, int depth, A_dec dec);

struct escapeEntry_ {bool *escape; int depth;};
typedef struct escapeEntry_ *escapeEntry;

static void traverseVar(S_table env, int depth, A_var v)
{
 switch (v->kind) {
        case A_simpleVar:
                        {
                         escapeEntry x = (escapeEntry)S_look(env,v->u.simple);
                         if (x && depth > x->depth)
                         (x->escape) = TRUE; /* changing escape field of absyn structure */
                         break;
                        }
        case A_fieldVar:
                      traverseVar(env, depth, v->u.field.var);
                      break;
        case A_subscriptVar:
```

```
                traverseVar(env, depth, v→u.subscript.var);
                traverseExp(env, depth, v→u.subscript.exp);
                break;
        } /* end switch */
}
```

22 Memory Management in the Tiger Compiler — Goals

- Isolate architecture dependent parts in a separate module (**Frame**)
- Isolate programming language dependent parts in a separate module (**Translate**)
- Isolate labels and register temporaries in a separate module (**Temp**)

23 Two Layers of Abstraction

$\boxed{\text{semant.c}}$	
translate.h	
$\boxed{\text{translate.c}}$	
frame.h	temp.h
$\boxed{\mu\text{frame.c}}$	$\boxed{\text{temp.c}}$

24 Example invocations (translate.c)

- When a function $g(x, y, z)$ where x escapes is encountered —

  ```
  f = F_newFrame (g, U_BoolList(TRUE,
                     U_BoolList(FALSE,
                     U_BoolList(FALSE, NULL)) ) )
  ```

- When a local variable v is encountered —

  ```
  a = F_allocLocal(f, escape)
  ```

- When a variable is accessed —- `F_Exp(f, a)` returns the generated code

25 Local Variables

- `F_allocLocal(f, true)` causes the allocation of a new space in the frame
- `InFrame(-4)`, `InFrame(-8)`, etc are allocated
- `F_allocLocal(f, false)` may return `InReg(`t_{481}`)`

26 Temporaries and Labels

```
/* temp.h */
typedef struct Temp_temp_ *Temp_temp;
Temp_temp Temp_newtemp(void);

typedef struct Temp_tempList_ *Temp_tempList;
struct Temp_tempList_ { Temp_temp head; Temp_tempList tail;};
Temp_tempList Temp_TempList(Temp_temp h, Temp_tempList t);
typedef S_symbol Temp_label;
Temp_label Temp_newlabel(void);
Temp_label Temp_namedlabel(string name);
string Temp_labelstring(Temp_label s);

⋮
/* temp.c */
⋮
```

27 Representation of Frame Descriptors (Hidden in the Frame Module)

- The location of the formals
- Machine instructions to implement "shift-of-view" (prologue)
- The number of locals allocated so far
- The label in which the machine code starts

28 The Frame Interface

```
/* frame.h */
typedef struct F_frame_ *F_Frame;
typedef struct F_access_ *F_access;

typedef struct F_accessList_ *F_accessList;
struct F_accessList_ { F_access head; F_accessList tail; }

F_frame F_newFrame(Temp_label name, U_boolList formals);
F_label F_name(F_frame d);
F_accessList F_formals(F_frame f);
F_access f_allocLocal(F_frame f, bool escape);
Temp_temp F_FP(void);
```

```
extern const int F_wordsize;
T_exp F_Exp(F_access acc, ···);
```

29 The MIPS Implementation(mipsframe.c)

```
#include "temp.h"
#include "frame.h"
struct F_frame_ {Temp_label name; int formalsCount; int localsCount;
               F_accessList formals; ···};

typedef enum {inFrame, inReg} F_access_kind;
struct F_access_ {
      F_access_kind kind;
      union { int offset; /* frame offset */
            Temp_temp reg; /* register */
      } u;
};
⋮
F_access F_allocLocal(F_frame f, bool escape)
{
 assert(f);
 if (escape)
   return F_allocInFrame(-1 * F_wordSize * ++f→localsCount);
 else return F_allocInRegister();
}
```

30 The Frames in Different Architectures

	Pentium	MIPS	Sparc
formal 1	InFrame(8)	InFrame(0)	InFrame(68)
formal 2	InFrame(12)	InReg(t_{157})	InReg(t_{157})
formal 3	InFrame(16)	InReg(t_{158})	InReg(t_{158})
View	$M[sp+0] := fp$	$sp := sp - K$	save %sp, -K, %sp
Shift	$fp := sp$	$M[sp+K+0] := r_2$	$M[fp+68] := i_0$
	$sp := sp - K$	$t_{157} := r_4$	$t_{157} := i_1$
		$t_{158} := r_5$	$t_{158} := i_2$

31 The Need for Register Copies

```
function m(x: int, y: int) = (h(y, y); h(x, x))
```

32 Nesting of Blocks in Tiger

```
function f() =
let var v := 6
    in print(v);
       let var v := 7
           in print(v)
       end;
       print(v);
       let var v := 8
           in print(v)
       end;
       print(v);
end
```

33 Managing Static Links

- Implemented in the translate module (translate.c)
- The static pointer is passed as extra argument
- For every function records the frame of function in which it is defined
- Generate instruction sequences for non-local references