6 Activation Records

stack: an orderly pile or heap

Webster's Dictionary

In almost any modern programming language, a function may have *local* variables that are created upon entry to the function. Several invocations of the function may exist at the same time, and each invocation has its own *instantiations* of local variables.

In the Tiger function,

```
function f(x: int) : int =
  let var y := x+x
   in if y < 10
          then f(y)
          else y-1
  end
```

a new instantiation of `x` is created (and initialized by `f`'s caller) each time that `f` is called. Because there are recursive calls, many of these `x`'s exist simultaneously. Similarly, a new instantiation of `y` is created each time the body of `f` is entered.

In many languages (including C, Pascal, and Tiger), local variables are destroyed when a function returns. Since a function returns only after all the functions it has called have returned, we say that function calls behave in last-in-first-out (LIFO) fashion. If local variables are created on function entry and destroyed on function exit, then we can use a LIFO data structure – a stack – to hold them.

```
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)
```

(a) Written in ML

```
int (*)() f(int x) {
  int g(int y) {return x+y;}
  return g;
}

int (*h)() = f(3);
int (*j)() = f(4);

int z = h(5);
int w = j(7);
```

(b) Written in pseudo-C

PROGRAM 6.1. An example of higher-order functions.

HIGHER-ORDER FUNCTIONS

But in languages supporting both nested functions *and* function-valued variables, it may be necessary to keep local variables after a function has returned! Consider Program 6.1: This is legal in ML, but of course in C one cannot really nest the function g inside the function f.

When $f(3)$ is executed, a new local variable x is created for the activation of function f. Then g is returned as the result of $f(x)$; but g has not yet been called, so y is not yet created.

At this point f has returned, but it is too early to destroy x, because when $h(5)$ is eventually executed it will need the value $x = 3$. Meanwhile, $f(4)$ is entered, creating a *different* instance of x, and it returns a *different* instance of g in which $x = 4$.

It is the combination of *nested functions* (where inner functions may use variables defined in the outer functions) and *functions returned as results* (or stored into variables) that causes local variables to need lifetimes longer than their enclosing function invocations.

Pascal (and Tiger) have nested functions, but they do not have functions as returnable values. C has functions as returnable values, but not nested functions. So these languages can use stacks to hold local variables.

ML, Scheme, and several other languages have both nested functions and functions as returnable values (this combination is called *higher-order functions*). So they cannot use stacks to hold all local variables. This complicates the implementation of ML and Scheme – but the added expressive power of higher-order functions justifies the extra implementation effort.

For the remainder of this chapter we will consider languages with stackable local variables and postpone discussion of higher-order functions to Chapter 15.

6.1 STACK FRAMES

The simplest notion of a *stack* is a data structure that supports two operations, *push* and *pop*. However, it turns out that local variables are pushed in large batches (on entry to functions) and popped in large batches (on exit). Furthermore, when local variables are created they are not always initialized right away. Finally, after many variables have been pushed, we want to continue accessing variables deep within the stack. So the abstract *push* and *pop* model is just not suitable.

Instead, we treat the stack as a big array, with a special register – the *stack pointer* – that points at some location. All locations beyond the stack pointer are considered to be garbage, and all locations before the stack pointer are considered to be allocated. The stack usually grows only at the entry to a function, by an increment large enough to hold all the local variables for that function, and, just before the exit from the function, shrinks by the same amount. The area on the stack devoted to the local variables, parameters, return address, and other temporaries for a function is called the function's *activation record* or *stack frame*. For historical reasons, run-time stacks usually start at a high memory address and grow toward smaller addresses. This can be rather confusing: stacks grow downward and shrink upward, like icicles.

The design of a frame layout takes into account the particular features of an instruction set architecture and the programming language being compiled. However, the manufacturer of a computer often prescribes a "standard" frame layout to be used on that architecture, where possible, by all compilers for all programming languages. Sometimes this layout is not the most convenient one for a particular programming language or compiler. But by using the "standard" layout, we gain the considerable benefit that functions written in one language can call functions written in another language.

Figure 6.2 shows a typical stack frame layout. The frame has a set of *incoming arguments* (technically these are part of the previous frame but they are at a known offset from the frame pointer) passed by the caller. The *return address* is created by the CALL instruction and tells where (within the calling function) control should return upon completion of the current function. Some *local variables* are in this frame; other local variables are kept in

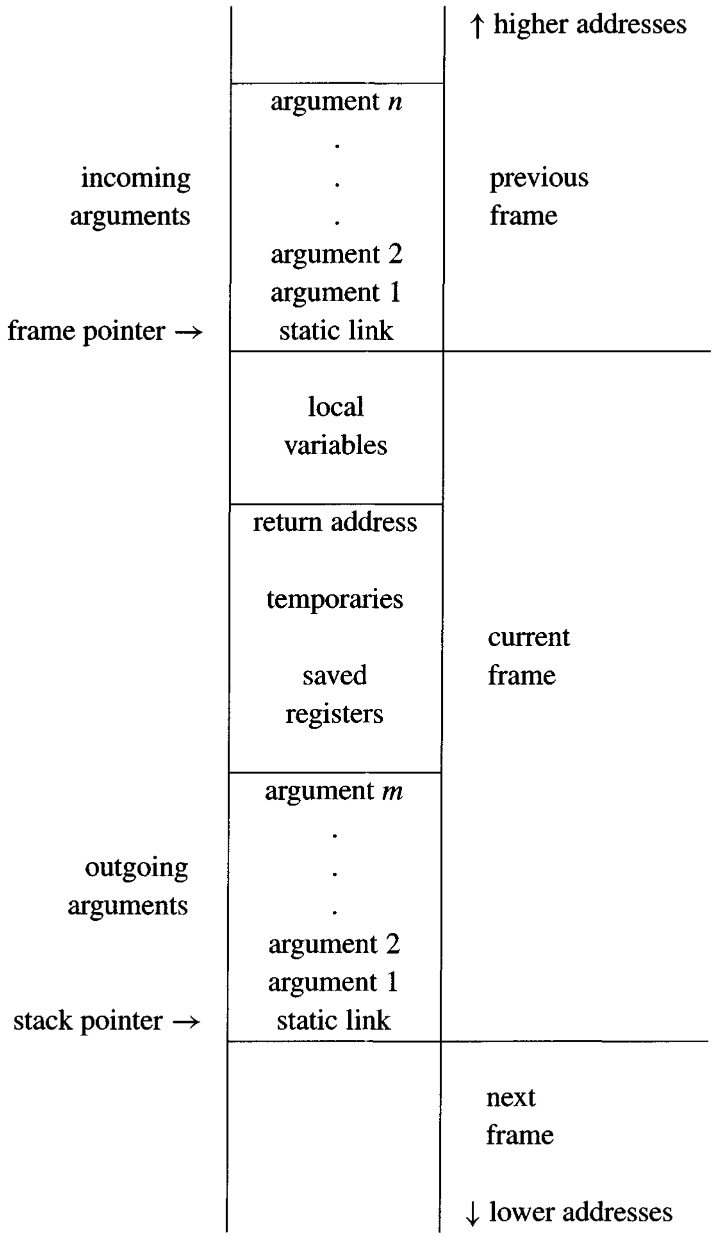


FIGURE 6.2. A stack frame.

machine registers. Sometimes a local variable kept in a register needs to be *saved* into the frame to make room for other uses of the register; there is an area in the frame for this purpose. Finally, when the current function calls other functions, it can use the *outgoing argument* space to pass parameters.

THE FRAME POINTER

Suppose a function $g(\ldots)$ calls the function $f(a_1, \ldots, a_n)$. We say g is the *caller* and f is the *callee*. On entry to f, the stack pointer points to the first argument that g passes to f. On entry, f allocates a frame by simply subtracting the frame size from the stack pointer SP.

The old SP becomes the current *frame pointer* FP. In some frame layouts, FP is a separate register; the old value of FP is saved in memory (in the frame) and the new FP becomes the old SP. When f exits, it just copies FP back to SP and fetches back the saved FP. This arrangement is useful if f's frame size can vary, or if frames are not always contiguous on the stack. But if the frame size is fixed, then for each function f the FP will always differ from SP by a known constant, and it is not necessary to use a register for FP at all – FP is a "fictional" register whose value is always SP+*framesize*.

Why talk about a frame pointer at all? Why not just refer to all variables, parameters, etc. by their offset from SP, if the frame size is constant? The frame size is not known until quite late in the compilation process, when the number of memory-resident temporaries and saved registers is determined. But it is useful to know the offsets of formal parameters and local variables much earlier. So, for convenience, we still talk about a frame pointer. And we put the formals and locals right near the frame pointer at offsets that are known *early*; the temporaries and saved registers go farther away, at offsets that are known *later*.

REGISTERS

A modern machine has a large set of *registers* (typically 32 of them). To make compiled programs run fast, it's useful to keep local variables, intermediate results of expressions, and other values in registers instead of in the stack frame. Registers can be directly accessed by arithmetic instructions; on most machines, accessing memory requires separate *load* and *store* instructions. Even on machines whose arithmetic instructions can access memory, it is faster to access registers.

A machine (usually) has only one set of registers, but many different procedures and functions need to use registers. Suppose a function f is using register r to hold a local variable and calls procedure g, which also uses r for its own calculations. Then r must be saved (stored into a stack frame) before g uses it and restored (fetched back from the frame) after g is finished using it. But is it f's responsibility to save and restore the register, or g's? We say that r is a *caller-save* register if the caller (in this case, f) must save and restore the register, and r is *callee-save* if it is the responsibility of the callee (in this case, g).

On most machine architectures, the notion of caller-save or callee-save register is not something built into the hardware, but is a convention described in the machine's reference manual. On the MIPS computer, for example, registers 16–23 are preserved across procedure calls (callee-save), and all other registers are not preserved across procedure calls (caller-save).

Sometimes the saves and restores are unnecessary. If f knows that the value of some variable x will not be needed after the call, it may put x in a caller-save register *and not save it* when calling g. Conversely, if f has a local variable i that is needed before and after several function calls, it may put i in some callee-save register r_i and, save r_i just once (upon entry to f) and fetch it back just once (before returning from f). Thus, the wise selection of a caller-save or callee-save register for each local variable and temporary can reduce the number of stores and fetches a program executes. We will rely on our register allocator to choose the appropriate kind of register for each local variable and temporary value.

PARAMETER PASSING

On most machines whose calling conventions were designed in the 1970s, function arguments were passed on the stack.[1] But this causes needless memory traffic. Studies of actual programs have shown that very few functions have more than four arguments, and almost none have more than six. Therefore, parameter-passing conventions for modern machines specify that the first k arguments (for $k = 4$ or $k = 6$, typically) of a function are passed in registers $r_p, ..., r_{p+k-1}$, and the rest of the arguments are passed in memory.

Now, suppose $f(a_1, \ldots, a_n)$ (which received its parameters in $r_1, \ldots, r_n$, for example) calls $h(z)$. It must pass the argument z in r_1; so f saves the old contents of r_1 (the value a_1) in its stack frame before calling h. But there is the memory traffic that was supposedly avoided by passing arguments in registers! How has the use of registers saved any time?

[1]Before about 1960, they were passed not on the stack but in statically allocated blocks of memory, which precluded the use of recursive functions.

There are four answers, any or all of which can be used at the same time:

1. Some procedures don't call other procedures – these are called *leaf* procedures. What proportion of procedures are leaves? Well, if we make the (optimistic) assumption that the average procedure calls either no other procedures or calls at least two others, then we can describe a "tree" of procedure calls in which there are more leaves than internal nodes. This means that *most* procedures called are leaf procedures.

   Leaf procedures need not write their incoming arguments to memory. In fact, often they don't need to allocate a stack frame at all. This is an important savings.
2. Some optimizing compilers use *interprocedural register allocation*, analyzing all the functions in an entire program at once. Then they assign different procedures different registers in which to receive parameters and hold local variables. Thus $f(x)$ might receive x in r_1, but call $h(z)$ with z in r_7.
3. Even if f is not a leaf procedure, it might be finished with all its use of the argument x by the time it calls h (technically, x is a dead variable at the point where h is called). Then f can overwrite r_1 without saving it.
4. Some architectures have *register windows*, so that each function invocation can allocate a fresh set of registers without memory traffic.

If f needs to write an incoming parameter into the frame, where in the frame should it go? Ideally, f's frame layout should matter only in the implementation of f. A straightforward approach would be for the caller to pass arguments $a_1, ..., a_k$ in registers and $a_{k+1}, ..., a_n$ at the end of its own frame – the place marked *outgoing arguments* in Figure 6.2 – which become the *incoming arguments* of the callee. If the callee needed to write any of these arguments to memory, it would write them to the area marked *local variables*.

The C programming language actually allows you to take the address of a formal parameter and guarantees that all the formal parameters of a function are at consecutive addresses! This is the `varargs` feature that `printf` uses. Allowing programmers to take the address of a parameter can lead to a *dangling reference* if the address outlives the frame – as the address of `x` will in `int *f(int x){return &x;}` – and even when it does not lead to bugs, the consecutive-address rule for parameters constrains the compiler and makes stack-frame layout more complicated. To resolve the contradiction that parameters are passed in registers, but have addresses too, the first k parameters are passed in registers; but any parameter whose address is taken must be written to a memory location on entry to the function. To satisfy `printf`, the memory locations into which register arguments are written must all be consecutive with the memory locations in which arguments $k+1$, $k+2$, etc. are written. Therefore, C programs can't have some of the arguments saved in one place and some saved in another – they must all be saved contiguously.

So in the standard calling convention of many modern machines the *calling* function reserves space for the register arguments in its own frame, next to the place where it writes argument $k+1$. But the calling function does not actually write anything there; that space is written into *by the called function*, and only if the called function needs to write arguments into memory for any reason.

A more dignified way to take the address of a local variable is to use *call-by-reference*. With call-by-reference, the programmer does not explicitly manipulate the address of a variable x. Instead, if x is passed as the argument to $f(y)$ where y is a "by-reference" parameter, the compiler generates code to pass the address of x instead of the contents of x. At any use of y within the function, the compiler generates an extra pointer dereference. With call-by-reference, there can be no "dangling reference," since y must disappear when f returns, and f returns before x's scope ends.

RETURN ADDRESSES

When function g calls function f, eventually f must return. It needs to know where to go back to. If the *call* instruction within g is at address a, then (usually) the right place to return to is $a+1$, the next instruction in g. This is called the *return address*.

On 1970s machines, the return address was pushed on the stack by the *call* instruction. Modern science has shown that it is faster and more flexible to pass the return address in a register, avoiding memory traffic and also avoiding the need to build any particular stack discipline into the hardware.

On modern machines, the *call* instruction merely puts the return address (the address of the instruction after the call) in a designated register. A non-leaf procedure will then have to write it to the stack (unless interprocedural register allocation is used), but a leaf procedure will not.

FRAME-RESIDENT VARIABLES

So a modern procedure-call convention will pass function parameters in registers, pass the return address in a register, and return the function result in a register. Many of the local variables will be allocated to registers, as will the intermediate results of expression evaluation. Values are written to memory (in the stack frame) only when necessary for one of these reasons:

- the variable will be passed by reference, so it must have a memory address (or, in the C language the `&` operator is anywhere applied to the variable);
- the variable is accessed by a procedure nested inside the current one;[2]
- the value is too big to fit into a single register;[3]
- the variable is an array, for which address arithmetic is necessary to extract components;
- the register holding the variable is needed for a specific purpose, such as parameter passing (as described above), though a compiler may move such values to other registers instead of storing them in memory;
- or there are so many local variables and temporary values that they won't all fit in registers, in which case some of them are "spilled" into the frame.

We will say that a variable *escapes* if it is passed by reference, its address is taken (using C's `&` operator), or it is accessed from a nested function.

When a formal parameter or local variable is declared, it's convenient to assign it a location – either in registers or in the stack frame – right at that point in processing the program. Then, when occurrences of that variable are found in expressions, they can be translated into machine code that refers to the right location. Unfortunately, the conditions in our list don't manifest themselves early enough. When the compiler first encounters the declaration of a variable, it doesn't yet know whether the variable will ever be passed by reference, accessed in a nested procedure, or have its address taken; and doesn't know how many registers the calculation of expressions will require (it might be desirable to put some local variables in the frame instead of in registers). An industrial-strength compiler must assign provisional locations to all formals and locals, and decide later which of them should really go in registers.

STATIC LINKS

In languages that allow nested function declarations (such as Pascal, ML, and Tiger), the inner functions may use variables declared in outer functions. This language feature is called *block structure*.

For example, in Program 6.3, `write` refers to the outer variable `output`, and `indent` refers to outer variables `n` and `output`. To make this work, the function `indent` must have access not only to its own frame (for variables `i` and `s`) but also to the frames of `show` (for variable `n`) and `prettyprint` (for variable `output`).

[2]However, with register allocation across function boundaries, local variables accessed by inner functions can sometimes go in registers, as long as the inner function knows where to look.

[3]However, some compilers spread out a large value into several registers for efficiency.

```
1       type tree = {key: string, left: tree, right: tree}
2
3       function prettyprint(tree: tree) : string =
4        let
5            var output := ""
6
7            function write(s: string) =
8                output := concat(output,s)
9
10           function show(n:int, t: tree) =
11              let function indent(s: string) =
12                      (for i := 1 to n
13                        do write(" ");
14                       output := concat(output,s); write("\n"))
15               in if t=nil then indent(".")
16                  else (indent(t.key);
17                        show(n+1,t.left);
18                        show(n+1,t.right))
19              end
20
21        in show(0,tree); output
22       end
```

PROGRAM 6.3. Nested functions.

There are several methods to accomplish this:

- Whenever a function f is called, it can be passed a pointer to the frame of the function statically enclosing f; this pointer is the *static link*.
- A global array can be maintained, containing – in position i – a pointer to the frame of the most recently entered procedure whose *static nesting depth* is i. This array is called a *display*.
- When g calls f, each variable of g that is actually accessed by f (or by any function nested inside f) is passed to f as an extra argument. This is called *lambda lifting*.

I will describe in detail only the method of static links. Which method should be used in practice? See Exercise 6.7.

Whenever a function f is called, it is passed a pointer to the stack frame of the "current" (most recently entered) activation of the function g that *immediately encloses* f in the text of the program.

For example, in Program 6.3:

Line #

21 `prettyprint` calls `show`, passing `prettyprint`'s own frame pointer as `show`'s static link.

10 `show` stores its static link (the address of `prettyprint's` frame) into its own frame.
15 `show` calls `indent`, passing its own frame pointer as `indent`'s static link.
17 `show` calls `show`, passing its own static link (not its own frame pointer) as the static link.
12 `indent` uses the value *n* from `show`'s frame. To do so, it fetches at an appropriate offset from `indent`'s static link (which points at the frame of `show`).
13 `indent` calls `write`. It must pass the frame pointer of `prettyprint` as the static link. To obtain this, it first fetches at an offset from its own static link (from `show`'s frame), the static link that had been passed to `show`.
14 `indent` uses the variable `output` from `prettyprint`'s frame. To do so it starts with its own static link, then fetches `show`'s, then fetches `output`.[4]

So on each procedure call or variable access, a chain of zero or more fetches is required; the length of the chain is just the *difference* in static nesting depth between the two functions involved.

6.2 FRAMES IN THE Tiger COMPILER

What sort of stack frames should the Tiger compiler use? Here we face the fact that every target machine architecture will have a different standard stack frame layout. If we want Tiger functions to be able to call C functions, we should use the standard layout. But we don't want the specifics of any particular machine intruding on the implementation of the semantic analysis module of the Tiger compiler.

Thus we must use *abstraction*. Just as the `Symbol` module provides a clean interface, and hides the internal representation of `table` from its clients, we must use an abstract representation for frames.

The frame interface will look something like this:

```
signature FRAME =
sig type frame
    type access
    val newFrame : {name: Temp.label,
                     formals: bool list} -> frame
    val name : frame -> Temp.label
    val formals : frame -> access list
    val allocLocal : frame -> bool -> access
        ⋮
end
```

[4]This program would be cleaner if `show` called `write` here instead of manipulating `output` directly, but it would not be as instructive.

The abstract signature `FRAME` is implemented by a module specific to the target machine. For example, if compiling to the MIPS architecture, there would be

```
structure MipsFrame : FRAME = struct ··· end
```

In general, we may assume that the machine-independent parts of the compiler have access to this implementation of `FRAME`; for example,

```
structure Frame : FRAME = MipsFrame
```

In this way the rest of the compiler may access `Frame` without knowing the identity of the target machine (except an occurrence of the word `MipsFrame` here and there).

The type `frame` holds information about formal parameters and local variables allocated in this frame. To make a new frame for a function f with k formal parameters, call `newFrame{name=`f`,formals=`l`}`, where l is a list of k booleans: `true` for each parameter that escapes and `false` for each parameter that does not. The result will be a `frame` object. For example, consider a three-argument function named g whose first argument escapes (needs to be kept in memory). Then

```
    Frame.newFrame{name=g,formals=[true,false,false]}
```

returns a new frame object.

The `access` type describes formals and locals that may be in the frame or in registers. This is an *abstract data type*, so its implementation as a `datatype` is visible only inside the `Frame` module:

```
structure MipsFrame : FRAME = struct
           ⋮
    datatype access = InFrame of int | InReg of Temp.temp
```

`InFrame(`X`)` indicates a memory location at offset X from the frame pointer; `InReg(`t_{84}`)` indicates that it will be held in "register" t_{84}. `Frame.access` is an abstract data type, so outside of the module the `InFrame` and `InReg` constructors are not visible. Other modules manipulate accesses using interface functions to be described in the next chapter.

The `Frame.formals` interface function extracts a list of k "accesses" denoting the locations where the formal parameters will be kept at run time, as seen from inside the callee. Parameters may be seen differently by the caller and the callee. For example, if parameters are passed on the stack, the caller may put a parameter at offset 4 from the stack pointer, but the callee sees it at offset 4 from the frame pointer. Or the caller may put a parameter into register 6, but the callee may want to move it out of the way and always access it from register 13. On the Sparc architecture, with register windows, the caller puts a parameter into register `o1`, but the `save` instruction shifts register windows so the callee sees this parameter in register `i1`.

Because this "shift of view" depends on the calling conventions of the target machine, it must be handled by the `Frame` module, starting with `newFrame`. For each formal parameter, `newFrame` must calculate two things:

- How the parameter will be seen from inside the function (in a register, or in a frame location);
- What instructions must be produced to implement the "view shift."

For example, a frame-resident parameter will be seen as "memory at offset X from the frame pointer," and the view shift will be implemented by copying the stack pointer to the frame pointer on entry to the procedure.

REPRESENTATION OF FRAME DESCRIPTIONS

The implementation module `Frame` is supposed to keep the representation of the `frame` type secret from any clients of the `Frame` module. But really it's a data structure holding:

- the locations of all the formals,
- instructions required to implement the "view shift,"
- the number of locals allocated so far,
- and the `label` at which the function's machine code is to begin (see page 140).

Table 6.4 shows the formals of the three-argument function g (see page 135) as `newFrame` would allocate them on three different architectures: the Pentium, MIPS, and Sparc. The first parameter escapes, so it needs to be `InFrame` on all three machines. The remaining parameters are `InFrame` on the Pentium, but `InReg` on the other machines.

The freshly created temporaries t_{157} and t_{158}, and the *move* instructions that copy `r4` and `r5` into them (or on the Sparc, `i1` and `i2`) may seem superfluous. Why shouldn't the body of g just access these formals directly from the registers in which they arrive? To see why not, consider

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

If x stays in "parameter register 1" throughout m, and y is passed to h in parameter register 1, then there is a problem.

		Pentium	MIPS	Sparc
	1	`InFrame`(8)	`InFrame`(0)	`InFrame`(68)
Formals	2	`InFrame`(12)	`InReg`(t_{157})	`InReg`(t_{157})
	3	`InFrame`(16)	`InReg`(t_{158})	`InReg`(t_{158})
		$M[\text{sp}+0] \leftarrow fp$	$\text{sp} \leftarrow \text{sp} - K$	`save %sp,-K,%sp`
View		$\text{fp} \leftarrow \text{sp}$	$M[\text{sp}+K+0] \leftarrow$ `r2`	$M[\text{fp}+68] \leftarrow$ `i0`
Shift		$\text{sp} \leftarrow \text{sp} - K$	$t_{157} \leftarrow$ `r4`	$t_{157} \leftarrow$ `i1`
			$t_{158} \leftarrow$ `r5`	$t_{158} \leftarrow$ `i2`

TABLE 6.4. Formal parameters for $g(x_1, x_2, x_3)$ where x_1 escapes.

The register allocator will eventually choose which machine register should hold t_{157}. If there is no interference of the type shown in function `m`, then (on the MIPS) the allocator will take care to choose register `r4` to hold t_{157} and `r5` to hold t_{158}. Then the *move* instructions will be unnecessary and will be deleted at that time.

See also pages 168 and 261 for more discussion of the view shift.

LOCAL VARIABLES

Some local variables are kept in the frame; others are kept in registers. To allocate a new local variable in a frame f, the semantic analysis phase calls

```
Frame.allocLocal(f)(true)
```

This returns an `InFrame` access with an offset from the frame pointer. For example, to allocate two local variables on the Sparc, `allocLocal` would be called twice, returning successively `InFrame(-4)` and `InFrame(-8)`, which are standard Sparc frame-pointer offsets for local variables.

The boolean argument to `allocLocal` specifies whether the new variable escapes and needs to go in the frame; if it is false, then the variable can be allocated in a register. Thus, `allocLocal(`f`)(false)` might create `InReg`(t_{481}).

The calls to `allocLocal` need not come immediately after the frame is created. In a language such as Tiger or C, there may be variable-declaration blocks nested inside the body of a function. For example,

```
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
```

```
void f()
{int v=6;
 print(v);
 {int v=7;
  print(v);
 }
 print(v);
 {int v=8;
  print(v);
 }
 print(v);
}
```

In each of these cases, there are three different variables v. Either program will print the sequence 6 7 6 8 6. As each variable declaration is encountered in processing the Tiger program, `allocLocal` will be called to allocate a temporary or new space in the frame, associated with the name v. As each `end` (or closing brace) is encountered, the association with v will be forgotten – but the space is still reserved in the frame. Thus, there will be a distinct temporary or frame slot for every variable declared within the entire function.

The register allocator will use as few registers as possible to represent the temporaries. In this example, the second and third v variables (initialized to 7 and 8) could be held in the same temporary. A clever compiler might also optimize the size of the frame by noticing when two frame-resident variables could be allocated to the same slot.

CALCULATING ESCAPES

Local variables that do not escape can be allocated in a register; escaping variables must be allocated in the frame. A `FindEscape` function can look for escaping variables and record this information in the `escape` fields of the abstract syntax. The simplest way is to traverse the entire abstract syntax tree, looking for escaping uses of every variable. This phase must occur before semantic analysis begins, since `Semant` needs to know whether a variable escapes *immediately* upon seeing that variable for the first time.

The traversal function for `FindEscape` will be a mutual recursion on abstract syntax `exp`'s and `var`'s, just like the type-checker. And, just like the type-checker, it will use environments that map variables to bindings. But in this case the binding will be very simple: it will be the `bool ref` that is to be set if the particular variable escapes:

```
structure FindEscape: sig val findEscape: Absyn.exp -> unit
                      end =
struct
 type depth = int
 type escEnv = (depth * bool ref) Symbol.table

 fun traverseVar(env:escEnv, d:depth, s:Absyn.var): unit = ···
 and traverseExp(env:escEnv, d:depth, s:Absyn.exp): unit = ···
 and traverseDecs(env, d, s: Absyn.dec list): escEnv = ···

 fun findEscape(prog: Absyn.exp) : unit = ···
end
```

Whenever a variable or formal-parameter declaration is found at static function-nesting depth d, such as

`VarDec`{name=symbol("a"), escape=r,...}

then the bool ref r is assigned `false`; the binding `a` $\mapsto (d, r)$ is entered into the environment.

This new environment is used in processing expressions within the scope of the variable; whenever a is used at depth $> d$, then r is set to `true`.

For a language where addresses of variables can be taken explicitly by the programmer, or where there are call-by-reference parameters, a similar `FindEscape` can find variables that escape in those ways.

TEMPORARIES AND LABELS

The compiler's semantic analysis phase will want to choose registers for parameters and local variables, and choose machine-code addresses for procedure bodies. But it is too early to determine exactly which registers are available, or exactly where a procedure body will be located. We use the word *temporary* to mean a value that is temporarily held in a register, and the word *label* to mean some machine-language location whose exact address is yet to be determined – just like a label in assembly language.

`Temps` are abstract names for local variables; `labels` are abstract names for static memory addresses. The `Temp` module manages these two distinct sets of names.

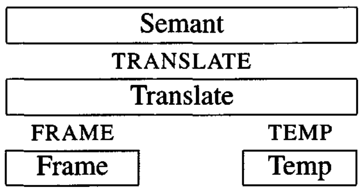
```
structure Temp :
sig
  eqtype temp
  val newtemp : unit -> temp
  structure Table : TABLE sharing type Table.key = temp
  val makestring: temp -> string

  type label = Symbol.symbol
  val newlabel : unit -> label
  val namedlabel : string -> label
end
```

`Temp.newtemp()` returns a new temporary from an infinite set of temps. `Temp.newlabel()` returns a new label from an infinite set of labels. And `Temp.namedlabel(`*string*`)` returns a new label whose assembly-language name is *string*.

When processing the declaration `function f(···)`, a label for the address of `f`'s machine code can be produced by `newlabel()`. It's tempting to call `namedlabel("f")` instead – the assembly-language program will be easier to debug if it uses the label `f` instead of `L213` – but unfortunately there could be two different functions named `f` in different scopes.

TWO LAYERS OF ABSTRACTION

Our Tiger compiler will have two layers of abstraction between semantic analysis and frame-layout details:



The FRAME and TEMP interfaces provide machine-independent views of memory-resident and register-resident variables. The `Translate` module augments this by handling the notion of nested scopes (via static links), providing the interface TRANSLATE to the `Semant` module.

It is essential to have an abstraction layer at FRAME, to separate the source-language semantics from the machine-dependent frame layout. Separating `Semant` from `Translate` at the TRANSLATE interface is not absolutely necessary: we do it to avoid a huge, unwieldy module that does both type-checking and semantic translation.

In Chapter 7, we will see how `Translate` provides ML functions that are useful in producing intermediate representation from abstract syntax. Here, we need to know how `Translate` manages local variables and static function nesting for `Semant`.

```
signature TRANSLATE =
sig
    type level
    type access    (* not the same as Frame.access *)

    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 = ···
```

In the semantic analysis phase of the Tiger compiler, `transDec` creates a new "nesting level" for each function by calling `Translate.newLevel`. That function in turn calls `Frame.newFrame` to make a new frame. `Semant` keeps this `level` in its `FunEntry` data structure for the function, so that when it comes across a function call it can pass the called function's `level` back to `Translate`. The `FunEntry` also needs the `label` of the function's machine-code entry point:

```
signature ENV =
sig
(* new versions of VarEntry and FunEntry *)
  datatype enventry =
             VarEntry of {access: Translate.access, ty: ty}
           | FunEntry of {level: Translate.level,
                          label: Temp.label,
                          formals: ty list, result: ty}
   ⋮
end
```

When `Semant` processes a local variable declaration at level `lev`, it calls Translate.allocLocal(lev)(esc) to create the variable in this level; the argument `esc` specifies whether the variable escapes. The result is a `Translate.access`, which is an abstract data type (not the same as `Frame.access`, since it must know about static links). Later, when the variable is used in an expression, `Semant` can hand this `access` back to `Translate` in order to generate the machine code to access the variable. Meanwhile, `Semant` records the access in each `VarEntry` in the value-environment.

The abstract data type `Translate.access` can be implemented as a pair consisting of the variable's `level` and its `Frame.access`:

```
type access = level * Frame.access
```

so that `Translate.allocLocal` calls `Frame.allocLocal`, and also remembers what level the variable lives in. The level information will be necessary later for calculating static links, when the variable is accessed from a (possibly) different level.

MANAGING STATIC LINKS

The `Frame` module should be independent of the specific source language being compiled. Many source languages do not have nested function declarations; thus, `Frame` should not know anything about static links. Instead, this is the responsibility of `Translate`.

`Translate` knows that each frame contains a static link. The static link is passed to a function in a register and stored into the frame. Since the static link behaves so much like a formal parameter, we will treat it as one (as much as possible). For a function with k "ordinary" parameters, let l be the list of booleans signifying whether the parameters escape. Then

l' `= true ::` l

is a new list; the extra `true` at the front signifies that the static link "extra parameter" does escape. Then `newFrame(`*label*, l'`)` makes the frame whose formal parameter list includes the "extra" parameter.

Suppose, for example, function $f(x, y)$ is nested inside function g, and the `level` (previously created) for g is called `lev`$_g$. Then `Semant.transDec` can call

`Translate.newLevel{parent=lev`$_g$`,name=`f`,formals=[false,false]}`

assuming that neither x nor y escapes. Then `Translate.newLevel` adds an extra element to the formal-parameter list (for the static link), and calls

`Frame.newFrame(` *label*`, [true,false,false])`

What comes back is a `frame`. In this frame are three frame-offset values, accessible by calling `Frame.formals(frame)`. The first of these is the static-link offset; the other two are the offsets for x and y. When `Semant` calls

Translate.formals(level), it will get these two offsets, suitably converted into access values.

KEEPING TRACK OF LEVELS

With every call to newLevel, Semant must pass the enclosing level value. When creating the level for the "main" Tiger program (one not within any Tiger function), Semant should pass a special level value: Translate.outermost. This is not the level of the Tiger main program, it is the level within which that program is nested. All "library" functions are declared (as described at the end of Section 5.2) at this outermost level, which does not contain a frame or formal parameter list.

The function transDec will make a new level for each Tiger function declaration. But newLevel must be told the enclosing function's level. This means that transDec must know, while processing each declaration, the current static nesting level.

This is easy: transDec will now get an additional argument (in addition to the type and value environments) that is the current level as given by the appropriate call to newLevel. And transExp will also require this argument, so that transDec can pass a level to transExp, which passes it in turn to transDec to process declarations of nested functions. For similar reasons, transVar will also need a level argument.

PROGRAM FRAMES

Augment semant.sml to allocate locations for local variables, and to keep track of the nesting level. To keep things simple, assume every variable escapes.

Implement the Translate module as translate.sml.

If you are compiling for the Sparc, implement the SparcFrame structure (matching the FRAME signature) as sparcframe.sml. If compiling for the MIPS, implement MipsFrame, and so on.

Try to keep *all* the machine-specific details in your machine-dependent Frame module, not in Semant or Translate.

To keep things simple, handle *only* escaping parameters. That is, when implementing newFrame, handle only the case where all "escape" indicators are true.

If you are working on a RISC machine (such as MIPS or Sparc) that passes the first k parameters in registers and the rest in memory, keep things simple by handling *only* the case where there are k or fewer parameters.

Optional: Implement `FindEscape`, the module that sets the `escape` field of every variable in the abstract syntax. Modify your `transDec` function to allocate nonescaping variables and formal parameters in registers.

Optional: Handle functions with more than *k* formal parameters.

Supporting files available in `$TIGER/chap6` include:

`temp.sig, temp.sml` The module supporting temporaries and labels.

FURTHER READING

The use of a single contiguous stack to hold variables and return addresses dates from Lisp [McCarthy 1960] and Algol [Naur et al. 1963]. Block structure (the nesting of functions) and the use of static links are also from Algol.

Computers and compilers of the 1960s and '70s kept most program variables in memory, so that there was less need to worry about which variables escaped (needed addresses). The VAX, built in 1978, had a procedure-call instruction that assumed all arguments were pushed on the stack, and itself pushed program counter, frame pointer, argument pointer, argument count, and callee-save register mask on the stack [Leonard 1987].

With the RISC revolution [Patterson 1985] came the idea that procedure calling can be done with much less memory traffic. Local variables should be kept in registers by default; storing and fetching should be done *as needed*, driven by "spilling" in the register allocator [Chaitin 1982].

Most procedures don't have more than five arguments and five local variables [Tanenbaum 1978]. To take advantage of this, Chow et al. [1986] and Hopkins [1986] designed calling conventions optimized for the common case: the first four arguments are passed in registers, with the (rare) extra arguments passed in memory; compilers use both caller- and callee-save registers for local variables; leaf procedures don't even stack frames of their own if they can operate within the caller-save registers; and even the return address need not always be pushed on the stack.

EXERCISES

6.1 Using the C compiler of your choice (or a compiler for another language), compile some small test functions into assembly language. On Unix this is usually done by `cc -S`. Turn on all possible compiler optimizations. Then evaluate the compiled programs by these criteria:

a. Are local variables kept in registers?

b. If local variable b is live across more than one procedure call, is it kept in a callee-save register? Explain how doing this would speed up the following program:

```
int f(int a) {int b; b=a+1; g(); h(b); return b+2;}
```

c. If local variable x is never live across a procedure call, is it properly kept in a caller-save register? Explain how doing this would speed up the following program:

```
void h(int y) {int x; x=y+1; f(y); f(2);}
```

6.2 If you have a C compiler that passes parameters in registers, make it generate assembly language for this function:

```
extern void h(int, int);
void m(int x, int y) {h(y,y); h(x,x);}
```

Clearly, if arguments to $m(x, y)$ arrive in registers $r_{\text{arg}1}$ and r_{arg2}, and arguments to h must be passed in $r_{\text{arg}1}$ and r_{arg2}, then x cannot stay in $r_{\text{arg}1}$ during the marshalling of arguments to $h(y, y)$. Explain when and how your C compiler moves x out of the $r_{\text{arg}1}$ register so as to call $h(y, y)$.

6.3 For each of the variables a, b, c, d, e in this C program, say whether the variable should be kept in memory or a register, and why.

```
int f(int a, int b)
{ int c[3], d, e;
  d=a+1;
  e=g(c, &b);
  return e+c[1]+b;
}
```

***6.4** How much memory should this program use?

```
int f(int i) {int j,k; j=i*i; k=i?f(i-1):0; return k+j;}
void main() {f(100000);}
```

a. Imagine a compiler that passes parameters in registers, wastes no space providing "backup storage" for parameters passed in registers, does not use static links, and in general makes stack frames as small as possible. How big should each stack frame for f be, in words?

b. What is the maximum memory use of this program, with such a compiler?

c. Using your favorite C compiler, compile this program to assembly language and report the size of f's stack frame.

d. Calculate the total memory use of this program with the real C compiler.

e. Quantitatively and comprehensively explain the discrepancy between (a) and (c).

f. Comment on the likelihood that the designers of this C compiler considered deeply recursive functions important in real programs.

***6.5** Some Tiger functions do not need static links, because they do not make use of a particular feature of the Tiger language.

a. Characterize precisely those functions that do not need a static link passed to them.

b. Give an algorithm (perhaps similar to `FindEscape`) that marks all such functions.

***6.6** Instead of (or in addition to) using static links, there are other ways of getting access to nonlocal variables. One way is just to leave the variable in a register!

```
function f() : int =
  let var a := 5
      function g() : int =
         (a+1)
   in g()+g()
  end
```

If a is left in register r_7 (for example) while g is called, then g can just access it from there.

What properties must a local variable, the function in which it is defined, and the functions in which it is used, have for this trick to work?

***6.7** A *display* is a data structure that may be used as an alternative to static links for maintaining access to nonlocal variables. It is an array of frame pointers, indexed by static nesting depth. Element D_i of the display always points to the most recently called function whose static nesting depth is i.

The bookkeeping performed by a function f, whose static nesting depth is i, looks like:

```
Copy $D_i$ to *save location* in stack frame
Copy frame pointer to $D_i$
  ⋯ body of $f$ ⋯
Copy *save location* back to $D_i$
```

In Program 6.3, function `prettyprint` is at depth 1, `write` and `show` are at depth 2, and so on.

a. Show the sequence of machine instructions required to fetch the variable `output` into a register at line 14 of Program 6.3, using static links.

b. Show the machine instructions required if a display were used instead.

c. When variable x is declared at depth d_1 and accessed at depth d_2, how many instructions does the static-link method require to fetch x?

d. How many does the display method require?

e. How many instructions does static-link maintenance require for a procedure entry and exit (combined)?

f. How many instructions does display maintenance require for procedure entry and exit?

Should we use displays instead of static links? Perhaps; but the issue is more complicated. For languages such as Pascal and Tiger with block structure but no function variables, displays work well.

But the full expressive power of block structure is obtained when functions can be returned as results of other functions, as in Scheme and ML. For such languages, there are more issues to consider than just variable-access time and procedure entry-exit cost: there is closure-building cost, and the problem of avoiding useless data kept live in closures. Chapter 15 explains some of the issues.