6 More about procedures

6.1 Nested procedures

A general feature of many programming languages is the ability to nest constructs within each other. For example, we can nest subexpressions inside larger expressions and loops inside other loops. We can extend the same idea to allow procedures to be nested inside other procedures.

Nested procedures enable a form of encapsulation, in that computations needed by only one procedure can be implemented inside it, rather than at the top level. A nested procedure has access to the variables declared in its enclosing procedures; this reduces the need for global variables and enables more control over the scope of data. As an example, consider the `compilerA5` procedure in `Transformations.scala`, which has the following overall structure:

```
208 def compilerA5(inputProcedures) = {
209   val procedures = ...
210   val paramChunks = ...
211   def compileProcedure(procedure) = {
212     def eliminateCalls(code) = { ... }
213     def addEntryExit(code, frame) = { ... }
214     def eliminateVarAccessesA5(code, frame) = { ... }
215     ...
216   }
217   ...
218 }
```

The job of `compileProcedure` is to compile a single procedure, the `procedure` parameter passed to it, but in the process, it sometimes needs to access information about the whole program, such as the `inputProcedures` parameter to `compilerA5` and variables such as `procedures` and `paramChunks`. The three procedures nested inside `compileProcedure` that implement different phases of the compilation process also have access to the `procedure` currently being compiled and the information about the whole program in `compilerA5`. Without nested procedures, all of this information would need to be passed as explicit parameters to these three phases, increasing the length of their parameter lists. In more complex examples, the parameter lists might get so long that the programmer reaches for global variables to make some data available everywhere. Global variables would make it difficult to see which parts of the code use which data.

Some prominent languages, such as C, do not support nested procedures, but many languages do, including languages as diverse as Scheme/Racket, Pascal, Ada, Modula-3, OCaml, Haskell, Python, Javascript, C#, Scala, and even Fortran 90.

As another example, consider the following program with a while loop:

```
def m() = {
  var i = 0
  var j = 0
  while(i < 10) {
    i = i + 1
    j = j + i
  }
  i+j
}
```

There is a straightforward transformation that can replace any loop with a recursive procedure:

```
def m() = {
  var i = 0
  var j = 0
  def loop(): Unit = {
    if(i < 10) {
      i = i + 1
      j = j + i
      loop()
    }
  }
  loop()
  i+j
}
```

We have changed the `while` keyword to `if`, declared a procedure (`loop`) and called it, and implemented the looping using a recursive call. Notice that the simplicity of this tranformation depends on the ability to nest procedures. Without nested procedures, we would have to pass the values of `i` and `j` as parameters into `loop` and return their modified values out of `loop`, complicating the code significantly. Although a recursive procedure does not make this particular example clearer than the original while loop, procedure calls can express more general control constructs than just while loops.

6.1.1 Static and dynamic scope

We have seen that the key feature that makes nested procedures useful is their ability to access variables of external procedures. Let's consider one more example:

```
241 def f() = {
242   val w = 5
243   def g() = {
244     val w = 7
245     h()
246   }
247   def h() = {
248     val z = 4
249     z + w
250   }
251   g()
252 }
```

In this code, `f` calls `g`, which calls `h`, and `h` accesses both its own variable `z` as well as external variable `w`. Which of the two `w` variables does `h` access? Does it access the `w` declared in `f` with value 5, which would make `h` return $4+5=9$, or does it access the `w` declared in `g` with value 7, which would make `h` return $4+7=11$?

The answer depends on whether we define our language with *dynamic scope* or *static scope*. In a language with *dynamic scope*, `h` accesses the variable `w` of procedure `g` with value 7. The word *dynamic* refers to run time, the execution of the program. During program execution, `g` is called after `f`, and the value 7 is assigned to `w` after the value 5, so `h` sees the most recent value 7. During program execution, the stack contains a frame of `f`, then a frame of `g`, and then a frame of `h`; since the frame of `g` is closer on the stack to the frame of `h`, `h` accesses the variable `w` of `g`.

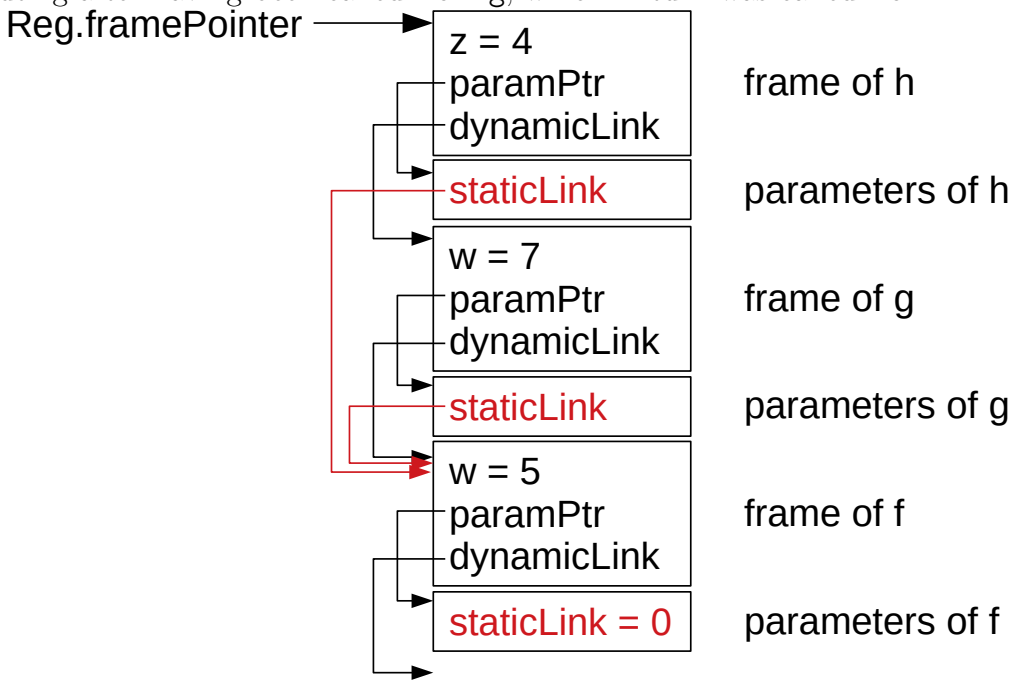
In a language with *static scope*, also called *lexical scope*, `h` accesses the variable `w` of procedure `f` with value 5. The word *static* refers to the *text* of the program, as opposed to its execution. In the program text, procedure `h` is nested inside procedure `f`, but it is not related to `g`. Rather than searching the run-time (dynamic) stack for a frame containing variable `w`, a statically scoped language searches the program text, starting from the current procedure `h` and moving outwards through its enclosing procedures until it finds a procedure that declares variable `w`.

Most modern programming languages are defined with static scope because it is easier to reason about: to find the variable `w` that procedure `h` accesses, just start at `h` and look outwards through the program text. In a dynamically-scoped language, we would have to consider all possible stacks that could occur at run time, and in turn all the procedures throughout the whole program that might call `h`, as well as other procedures that call them. Some languages are defined with dynamic scoping for historical reasons, partly because it is somewhat easier to implement. In this course, we will implement *statically* scoped variables.

6.1.2 Implementing nested procedures

The main issue in implementing statically-scoped nested procedures in machine language is to enable a nested procedure to access (read and write) the variables of its outer enclosing procedures. In the example from the previous section, inside procedure `h`, we need to access variable `w` of procedure `f`. The following diagram shows the layout of the stack when procedure

`h` is executing after having been called from `g`, which in turn was called from `f`.



The stack contains a frame for each of the three procedures. Each frame contains a parameter pointer holding the address of the parameters of the procedure. The procedures in our example, `f`, `g`, and `h`, do not take any parameters, but if they did, the arguments would be stored in the memory chunks pointed to by the parameter pointers, where the static links are; we will ignore the static links for a moment. Each frame also has a dynamic link pointing to the frame of the procedure that called it. The frame pointer register holds the address of the frame of the currently executing procedure, procedure `h`.

When executing procedure `h`, we can access its variables (such as `z`) in its frame, whose address we have in the frame pointer register, but how can we access the variables of its enclosing procedures (such as `w`)? We might consider following the dynamic link. Following the dynamic link from the frame of `h`, we reach the frame of `g`, which contains a variable `w` with the value 7. But recall that we want variables to be statically scoped, and thus we want to access the other variable `w`, in the statically enclosing procedure of `h`, which is procedure `f`. This other variable `w` has the value 5 in our example.

To get to the frame of `f`, we would have to follow the dynamic link twice: the first time to get from the frame of `h` to the frame of `g`, and the second time to get from the frame of `g` to the frame of `f`. But in general, we can't know how many stack frames there are between the frame of `h` and the frame of its enclosing procedure `f`. During program execution, procedure `g` could call an arbitrary number of other procedures before it called one that calls `h`. Thus, the correct number of times we would need to follow the dynamic link to reach the frame of the enclosing procedure depends on the runtime execution, and is generally unknown at compile time, just by (statically) observing the program.

For this reason, we add another pointer, the *static link*. In each procedure, the static link holds the address of the most recent frame of its directly enclosing procedure. Notice that in the diagram, the static link of `h` points to the frame of `f` because in the program text, `h` is nested directly inside `f`. The static link of `g` also points to the frame of `f` because in the

program text, `g` is also nested directly inside `f`. There is no static link that makes sense for procedure `f` because in our example, `f` is a top-level procedure that is not nested inside any other procedure, so we just set its static link to 0.

6.1.3 Accessing outer variables

To generate code to access variable `w` from inside procedure `h`, a compiler generates the following accesses:

1. Read the parameter pointer in the frame of `h` to find the parameters of `h`.
2. Read the static link in the parameters of `h` to find the frame of the directly enclosing procedure `f`.
3. Access the variable `w` in the frame of `f`.

In general, procedures can be nested multiple levels deep. Consider the following example of a more interesting program structure:

```
253 def f() = {
254   val w = 5
255   def g() = { ... }
256   def h() = {
257     def k() = { ... }
258   }
259 }
```

To access variable `w` in procedure `f` from procedure `k`, a compiler would have to generate code that follows the static link *twice*: the first time to get from the frame of procedure `k` to the frame of its directly enclosing procedure `h`, and the second time to get from the frame of procedure `h` to the frame of *its* directly enclosing procedure `f`.

In general, we can use the *nesting depths* of procedures to determine how many times to follow the static link. The *nesting depth* of a procedure is the number of levels of outer procedures that enclose it. For example, the nesting depth of `f` is 0 because it is a top-level procedure, the nesting depth of `h` is 1 because it is enclosed in `f`, and the nesting depth of `k` is 2 because it is enclosed in `h` which is enclosed in `f`.

To access a given variable v, we first calculate the difference in nesting depths $n = \text{depth}(c) - \text{depth}(d)$, where c is the procedure that we are *c*urrently executing or compiling and d is the procedure in which the variable v is *d*eclared in the program text. The difference in depths n is the number of times the generated code needs to follow static links to reach the procedure in whose frame or parameters it can access the variable v. In the example of accessing variable `w` of procedure `f` from procedure `k`, the difference in depths is $n = \text{depth}(\texttt{k}) - \text{depth}(\texttt{f}) = 2 - 0 = 2$, so the generated code needs to follow the static link twice.

6.1.4 Computing static links

So far, we have assumed that the static links correctly point to the frames of the directly enclosing procedures. It remains to discuss how to compute the correct addresses for the

static links. In general, when one procedure (the caller) calls another procedure (the callee), the caller has the information necessary to determine the correct static link for the callee. This is why we have grouped the static link with the *parameters* of a procedure. We will make it the responsibility of the caller to compute the correct static link address and pass it to the callee in the same way as if it were just another extra argument.

Let's consider a few examples first. When procedure `f` calls procedure `h` nested within it, what address should be passed as the static link? The static link should be the address of the frame of the procedure directly enclosing the callee, in this case, the frame of `f`. Thus, `f` should pass *its own frame pointer*, the value in the frame pointer register, as the static link for `h`.

When procedure `h` calls a sibling procedure `g`, which has the same directly enclosing procedure `f`, what address should `h` pass as the static link for `g`? Again, it should be the address of the frame of the procedure directly enclosing the callee, the frame of `f`. Thus, `h` should pass *its own static link* to become the static link of `g`.

What if procedure `k` calls out to procedure `g`? The static link of `g` should be the address of a frame of `f`. The static link of procedure `k` points to a frame of its directly enclosing procedure `h`. From `h`, we can read a second static link that points to a frame of *its* enclosing procedure `f`. Thus, when `k` calls `g`, we need to read static links twice: we read `k`'s static link to get to the frame of `h`, where we then read `h`'s static link to get the address of the frame of `f`.

We can generalize this as follows. The static link should always point to a frame of the procedure *enclosing* that directly encloses the *callee*. Thus,

$$\text{depth}(\textit{enclosing}) = \text{depth}(\textit{callee}) - 1$$

We will define n to be the number of times the caller needs to read a static link, starting from its own frame, to reach a static link that points to a frame of *enclosing*. Each read of a static link reduces the depth by 1; that is, in a procedure at depth d, the static link points to the frame of an enclosing procedure with depth $d - 1$. Therefore, to reach the frame of a procedure with the depth of *enclosing* from the *caller* procedure, we need to read static links

$$n = \text{depth}(\textit{caller}) - \text{depth}(\textit{enclosing})$$

times. Now substitute the first equation into the second to obtain an equation in terms of only *caller* and *callee*:

$$\begin{aligned} n &= \text{depth}(\textit{caller}) - (\text{depth}(\textit{callee}) - 1) \\ &= \text{depth}(\textit{caller}) - \text{depth}(\textit{callee}) + 1 \end{aligned}$$

When $n = 0$, we don't read the static link at all; we pass the frame pointer of the caller to become the static link of the callee, as when procedure `f` calls procedure `h`. When $n = 1$, we read the static link of the caller once, and it becomes the static link of the callee, as when procedure `h` calls procedure `g`. When $n > 1$, we read static links n times, moving outwards in the program starting from the caller procedure, to obtain the value of the static link of the callee procedure. For example, when procedure `k` calls out to procedure `g`, $n = 2 - 1 + 1 = 2$, so we read two static links to determine the static link for `g`.

What if $n < 0$? An example of this would be procedure `f` calling procedure `k` directly, without going through procedure `h`. In this case, $n = 0-2+1 = -1$. This case is not possible because procedure `k` is only visible inside procedure `h` in which it is declared; it cannot be called from outside procedure `h`, but procedure `f` *is* outside `h`. If procedure `f` could call procedure `k` directly, without going through `h`, how could `k` access the variables of `h`, when `h` has never been called and there is no frame of `h` on the stack? Such a call does not make sense. In our implementation in the assignments, there is currently nothing to prevent such an invalid call, but we will add checks to prevent it in Assignment 9. For now, assume that we will not test your implementation with such nonsensical calls.

6.2 Function values

High-level programming languages work with many different kinds of values, such as integers, characters, strings, and larger data structures. For many programming tasks, it is useful to also allow *functions* to be values, so that they can be stored in variables and passed into and returned from procedures in the same way as other values. Such *function values* are also called *first-class functions* to emphasize that anything that can be done with any other value can also be done with a value that is a function.

In Scala, we can create a function value from a procedure as follows by writing just the name of the procedure:

```
260 def procedure(x: Int) = { x + 1 }
261 var increase: (Int)=>Int = procedure
262 // increase is now a variable that holds a function value
```

Here, we have created a function value from `procedure` and stored it in the variable `increase`. In this particular example, Scala requires us to explicitly provide the type for the variable. The type `(Int)=>Int` means a function that takes one parameter of type `Int` and returns a value of type `Int`.

We can now call the function as `increase(5)`, which returns 6. Note that this is quite a different operation than the similar looking call `procedure(5)`, which also returns 6. The second call calls the specific procedure named `procedure`, while the first call reads a function value from the variable `increase`, and then calls this function value.

To see the difference, we can write a new function value into the same variable:

```
263 increase = { x => x + 2 }
```

In this case, we have used the Scala syntax for an *anonymous function*: the syntax p `=>` e denotes a function that takes a parameter named p and whose body is the expression e. However, we could just as well have assigned a function value created from a named procedure such as `def procedure`.

Now, when we call `increase(5)`, the result is no longer 6, but 7. The variable `increase` now holds a different function, one that increases by 2.

Another name for an anonymous function is a *lambda*, since many languages use the letter λ (lambda) in the syntax for anonymous functions. Note that *anonymous functions* (*lambdas*) are not exactly the same thing as *function values* (*first-class functions*): although

anonymous functions denote function values, just as the literal symbol 5 denotes the integer 5, so a language with anonymous functions must also support function values, function values can also be created from named procedures, so a language with function values need not support anonymous functions. The Lacs language that we implement in this course supports function values but not anonymous functions. Anonymous functions are just a convenient alternate syntax for the same concept as named functions and do not in themselves introduce any interesting semantics.

In Scala, function values can not only be stored in variables, but they can also be passed to and returned from procedures:

```
264 List(5,6,7).map(increase)
265
266 def twice(fun: (Int)=>Int): (Int)=>Int = {
267   x => fun(fun(x))
268 }
269 increase = twice(increase)
```

In the first line, we pass the function value in the variable `increase` into `List.map`, which applies the function to each element in the list, returning the list `List(7,8,9)`. In the subsequent lines, we define a procedure `twice` that takes any function `fun` (of the specified type) as a parameter and returns a new function that applies `fun` twice to its parameter `x`. We then call `twice` on the function in `increase` to create a new function that adds two twice to any number; i.e., it adds four. Calling `increase(5)` now returns 9.

Notice that the function to be called can be designated not only by a name such as the name of a procedure (e.g., `procedure`) or the name of a variable (e.g., `increase`), but by any possibly complicated expression that evaluates to a function value. For example, we can call:

```
270 (twice(procedure))(5)
```

This expression creates a function value from the procedure `procedure`, passes it to `twice`, which returns a new function value; it then calls this new function value with argument 5. The new function value adds one twice and returns 7.

6.2.1 Free variables

Suppose we wanted a general function that increases its argument `x` not only by 1, but by any increment:

```
271 increase = { x => x + increment }
```

Scala rejects this expression because the variable `increment` is not defined. We say that `increment` is a *free* variable in this expression because it is not *bound*; in contrast, the variable `x` can be used in the expression `x + increment` because it is *bound* when it is introduced as the parameter to the anonymous function in `{ x => ... }`.

To make such a function work, we can nest it inside a procedure that declares the `increment` variable:

```
272 def increaseBy(increment: Int): (Int)=>Int =
273   { x => x + increment }
274 increase = increaseBy(3)
```

Now, whenever we call `increaseBy` with some increment, it creates a new function value that increments by the given increment, and returns it to us. As an example, we create a function that increments by 3 and assign it to the variable `increase`. Now, the call `increase(5)` returns 8.

We can create a list of such functions and apply all of them to 5:

```
275 List(increaseBy(1), increaseBy(2), increaseBy(3)).
276   map(fun => fun(5))
```

This expression applies the three different functions, each to the argument 5. Its result is the list `List(6,7,8)`. A more interesting equivalent expression is:

```
277 List(1,2,3).map(increaseBy).map(fun => fun(5))
```

This expression applies `increaseBy` to each of a list of integers to create a list of functions, then applies each function in the list to the integer 5 to yield the list of integer results `List(6,7,8)`.

Since the body of `increaseBy` defines an anonymous function, we will not be able write it in Lacs, but we will be able to write an equivalent procedure by giving the nested function a name (such as `procedure`):

```
278 def increaseByLacs(increment: Int): (Int)=>Int = {
279   def procedure(x: Int) = x + increment
280   procedure // closure creation
281 }
282 increase = increaseByLacs(3)
283 increase(5) // closure call
```

6.2.2 Implementing function values

How can we represent the function values exhibited in the previous section in a computer, in machine language? Obviously, the representation of a function value needs to refer to the code of function body. This can be done with a *function pointer*, the address of the code of the body. When we call the function, we will load this address into the program counter so that the CPU executes the body of the function. You may have encountered function pointers in a programming language such as C.

Are function pointers enough to implement function values? In the example in the previous section, which address (function pointer) should the calls `increaseBy(1)`, `increaseBy(2)`, and `increaseBy(3)` return? All three calls return the *same* function pointer, the body of the anonymous function `{ x => x + increment }`. But the function values need to behave *differently*: they should add 1, 2, and 3, respectively. Therefore, the function pointer is not enough. To represent a function value, we additionally need a representation of an

environment, a mapping from the *free variables* in the function body (such as `increment`) to their values (such as 1, 2, or 3). Thus, we will implement a function value using a *closure*, which is a *pair* of a *function pointer* (address of the code of the body of the function) and an *environment* (mapping free variables to their values).

To implement the environment in a closure, we can observe that the environment is the frame of the procedure enclosing the closure body. When the machine language code for the body of the procedure `procedure` nested inside the procedure `increaseByLacs` needs to access the variable `increment`, it will do so by following its static link to reach the frame of `increaseByLacs`, where it will find the variable `increment`. If `increaseByLacs` were itself further nested inside additional enclosing procedures, `procedure` could access the variables of those procedures by following static links starting from the frame of `increaseByLacs`. Therefore, the address of the frame of `increaseByLacs` is sufficient to describe the environment for the free variables of `procedure`. A closure can thus be represented as a pair of two addresses: the address of the code of the function body and the address of the frame of the enclosing procedure of the function.

6.2.3 Closure operations

There are two operations associated with closures: *closure creation*, which creates a closure value from a procedure, and *closure call*, which invokes the function represented by a closure value. In the above example, closure creation occurs on Line 280, when a closure value is created from the procedure `procedure`, and is then returned as the return value of `increaseByLacs`. A closure call occurs on Line 283, when a closure value is read from the variable `increase` and the function that it represents is called with the argument 5. In the `Code` intermediate representation in the assignments, in `ProgramRepresentation.scala`, closure creation is represented by the `Closure` class and closure call is represented by the `CallClosure` class, both at the bottom of the file. Look in `ProgramRepresentation.scala` now at the definitions of these classes.

To implement a *closure creation*, we must compute the two addresses that go into the new closure. The function pointer is straightforward: it is the code address (label) of the procedure from which we are creating the closure (the procedure called `procedure` in our example), and this procedure is specified at the closure creation site (i.e., Line 280 specifies it literally as `procedure`). The environment should be the address of the frame of the procedure directly enclosing the procedure from which we are creating the closure. It turns out that this is the same address that we would pass as the static link if we were directly calling the procedure instead of creating a closure out of it. Therefore, the environment for the closure is computed in exactly the same way as we would compute the static link for a call to the same procedure. As an example, if the code at Line 280 contained a call to `procedure` such as `procedure(42)`, we would pass the frame pointer of `increaseByLacs` as the static link for `procedure`. Since the code instead contains a closure creation at this line, we save this same frame pointer into the closure as the environment address.

To implement a *closure call*, we need to make use of the two addresses in the closure. The code address tells us where the code of the function is; this is the address that we load into the program counter using the `JALR` instruction. The environment provides values for free variables of the function value. Since the code of the function accesses these free variables

using its static link, it follows that we must pass the environment address retrieved from the closure as the static link for the function that we call.

One way to think about this is that closure creation and closure call split the various steps that are normally done in a direct procedure call (like the one we learned about in the previous module) into two parts, which are done at different times. Recall that a direct procedure call involves the following steps:

1. Evaluate the arguments.
2. Compute the static link.
3. Allocate memory for the arguments and static link, and copy them from temporary variables into that memory.
4. Determine the address (label) of the code of the procedure to be called.
5. Set the program counter to that address using the `JALR` instruction.

During *closure creation*, we perform steps 2 and 4. We store the resulting two addresses in the closure value. Then, sometime later, during a *closure call*, we perform steps 1, 3, and 5, reading the addresses that would normally come from steps 2 and 4 out of the closure value.

6.2.4 Extent of environments

In Module 3, we observed that the extent of a local variable of a procedure is one execution of that procedure until it returns, and it is therefore natural to allocate local variables on a stack. In the presence of function values, this observation no longer always holds. Consider the `increment` variable of the `increaseByLacs` procedure. When `increaseByLacs` finishes its execution, it returns a function value, and that function value may access the `increment` variable at a later time, when it is called with a closure call. Thus, the extent of the `increment` variable is now *longer* than the execution of `increaseByLacs`. The extent continues for as long as we have access to the function value that is created inside `increaseByLacs` and returned. If we were to store the `increment` variable on the stack, the program may continue to access the variable long after it has been popped off the stack and possibly overwritten by other data.

The consequence is that to correctly translate the program to machine language, we need to allocate memory for the parameters and frame of procedure `increaseByLacs` not on a stack, but on a heap. A *heap* is a data structure for managing memory that allows segments of memory to be allocated and freed for reuse at arbitrary times. Since heaps are quite complicated, we will leave detailed coverage until near the end of the course. But we need a heap now to implement function values, so in Assignment 6, you will implement a very simple heap that only allocates memory segments and never frees them for reuse. In Assignment 11, you will replace it with a real heap that recycles memory segments when they are no longer needed.

Which procedures require their parameters and frame to be allocated on the heap rather than on the stack? All procedures whose variables could be accessed by a procedure that is made into a function value (i.e., is mentioned at a closure creation site) need their parameters

and frame on the heap. In the example, procedure `procedure` is mentioned in a closure creation site and accesses a variable of procedure `increaseByLacs`, so the parameters and frame of `increaseByLacs` need to be allocated on the heap. As a conservative, safe choice, if some procedure p is made into a function value, we can designate all outer enclosing procedures of p to be allocated on the heap, because the body of p could access their variables. If procedure `increaseByLacs` were itself further nested inside some other procedure `outer`, we would allocate procedure `outer` on the heap as well, because procedure `procedure` could access variables of both `increaseByLacs` and `outer`. This safe choice is recommended for Assignment 6.

A more aggressive compiler could look through the body of procedure `procedure` to list the variables that it actually accesses, and then allocate on the heap only the procedures in which those variables are declared. For example, if procedure `procedure` accesses a variable of procedure `increaseByLacs` but does not access any variables of procedure `outer`, then procedure `outer` could still be allocated on the stack for efficiency. An even more aggressive compiler could split the frame of a procedure, separating the variables that are accessed from function values from those that are not, and allocate part of the frame on the heap and the rest of the frame on the stack.

One final question is whether the frame and parameters of procedure `procedure` should be allocated on the stack or on the heap. Since there is no other procedure nested within it that could become a function value, it seems safe to allocate `procedure` on the stack. However, whenever a closure call site calls a function, it needs to allocate memory for the parameters somewhere, either on the stack or on the heap. When we compile the closure call site, we do not know which function it will call. That information is known only at run time, when the closure call site is actually executed on some closure value. A single closure call site in the program could even call multiple different functions at run time. When we compile the closure call site, we have no way of knowing whether it will call functions at run time that require their parameters on the stack or on the heap. To make a decision, we take the safe choice and allocate the frames and parameters of all procedures that could ever become function values on the heap, so the choice is consistent for all possible function values. Then a closure call site can always allocate the parameters on the heap. In summary, every procedure from which we create a function value (i.e., every procedure mentioned at a closure creation site) also has its frame and parameters allocated on the heap.

6.2.5 Closure representation

It remains to consider how to represent a closure value in our machine language program. Thus far, values have been integers that fit into a 32-bit word, but a closure contains *two* addresses requiring *two* 32-bit words. To communicate both addresses, we allocate memory for a `Chunk` with two variables. The two addresses are stored in this memory. We then use the address of this memory as the representation of the closure value, which we can store in a variable or pass into or return from a procedure. The definition of this `Chunk` can be found in `Transformations.scala` in the handout code as `closureChunk`, containing the variables `closureCode` and `closureEnvironment`.

6.3 Objects

Many languages are object-oriented in that they allow the creation of *objects*, records that contain both data (state) and procedures (behaviour). In Scala, we can define a class and instantiate objects of that class as in the following example:

```
284 class Counter {
285   var value = 0
286   def get() = value
287   def incrementBy(amount: Int) = {
288     value = value + amount
289   }
290 }
291
292 val c = new Counter
293 c.incrementBy(42)
294 c.incrementBy(-5)
295 c.get()
```

Each object of the `Counter` class has its own state, its own instance of the variable `value`, and its methods `incrementBy` and `get` can read and write that state. We could instantiate multiple counter objects, each with its own value, and call their methods to manipulate their values.

The following Scala code implements equivalent functionality in a different way, without using a class, using only nested procedures and function values.

```
296 def newCounter: (()=>Int, (Int)=>Unit) = {
297   var value = 0
298   def get() = value
299   def incrementBy(amount: Int) = {
300     value = value + amount
301   }
302   (get, incrementBy)
303 }
304
305 val c2 = newCounter
306 c2._2(42)
307 c2._2(-5)
308 c2._1()
```

Differences from the previous example are highlighted in yellow. Instead of declaring a class, the code declares a constructor of that class, a procedure that creates and returns new objects. The state of the object is held in variables declared in the constructor. The methods of the object are procedures nested in the constructor. The *object* is a collection of function values with a common shared environment, which the function values can use to maintain and communicate state with each other. As before, we can instantiate multiple counter objects by calling the constructor, the `newCounter` procedure, multiple times. Each such counter

object will have its own value and methods to manipulate it.

The remaining differences are cosmetic. A Scala class provides a way to group multiple values into a single record value where they can be accessed using member names (e.g., `get` and `incrementBy`). In the second example, this functionality is emulated using a pair, whose member names are fixed to be `_1` and `_2`.

This example demonstrates the equivalence of objects and function values (closures). Various programming languages provide explicit support for objects, function values, or both, but this example shows that function values have enough expressive power to implement objects.

6.4 Tail calls

In Section 6.1, we discussed a transformation of loops into recursive nested procedures. Consider the same example again, but with 10 million loop iterations instead of only 10.

```
309 def m() = {
310   var i = 0
311   var j = 0
312   def loop(): Unit = {
313     if(i < 10*1000*1000) {
314       i = i + 1
315       j = j + i
316       loop()
317     }
318   }
319   loop()
320   i+j
321 }
```

Since each call to the `loop` procedure pushes a frame onto the stack, we might expect the stack to grow very large and possibly overflow the memory that is available for it. However, this Scala code runs without running out of stack space no matter how high we increase the iteration count. How is this possible?

Recall that we can deallocate the frame of a procedure when it returns because the extent of the local variables of the procedure ends (assuming no function values are created). In this example, we would pop the frame of procedure `loop` when it ends at Line 318. But notice that the recursive call to `loop` at Line 316 is the last thing that the `loop` procedure does before it returns, and in particular, `loop` never accesses any of its variables after this call (between Lines 316 and 318). Such a call that is executed as the last operation before a procedure returns is called a *tail call*. Notice that a tail call need not be textually at the end of the code of a procedure because it can be wrapped in one or more control structures, such as the `if` statement in this example. Although many tail calls are recursive like the one in this example, a call could be a tail call even if it is not recursive.

In the presence of a tail call, we can safely *shorten* the extent of the variables of the procedure so that it ends just *before* the tail call, instead of immediately after it. If any of the variables are used in arguments to the tail call, we need to be more precise: the extent

needs to end *after* the arguments have been evaluated but just *before* we call the callee. Having shortened the extents, we can then deallocate the frame of the caller immediately *before* the tail call rather than immediately after it. This reduces the number of frames that are on the stack at any given time. For the `loop` example, the number of frames on the stack at the same time is reduced from 10 million to just one, since we will pop the frame of one instance of `loop` just before it recursively calls the next instance of `loop`.

6.4.1 Implementing tail call optimization

To optimize a tail call to free the frame of the caller before the call, we need to interleave the implementation of the call with the implementation of the epilogue of the caller. The following pseudocode lists the tasks that are done when a tail call occurs just before a procedure returns:

```
// (tail) call
evaluate arguments
//
allocate memory for callee's parameters
write argument values into callee's parameters
LIS/JALR

// caller's epilogue
Reg.link = savedPC
Reg.framePointer = dynamicLink
Stack.pop // caller's frame
Stack.pop // caller's parameters
JR(Reg.link (31))
```

Roughly speaking, the tail call optimization needs to interleave the epilogue at the place of the red `//` comment, in between evaluating arguments and allocating memory for the callee's parameters:

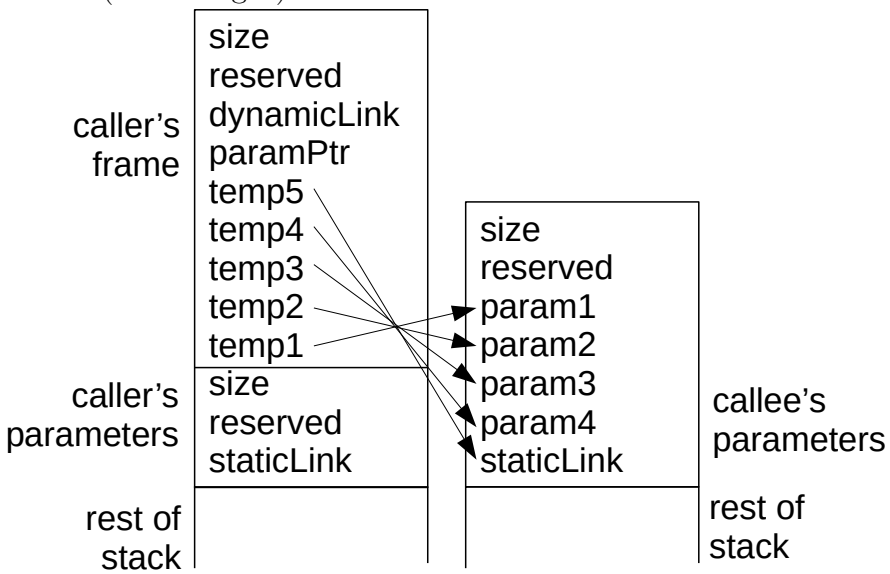
```
// (tail) call
evaluate arguments
// caller's epilogue
Reg.link = savedPC
Reg.framePointer = dynamicLink
Stack.pop // caller's frame
Stack.pop // caller's parameters
allocate memory for callee's parameters
write argument values into callee's parameters
LIS/JALR
JR(Reg.link (31))
```

One immediate problem is the `JALR` instruction immediately followed by `JR(Reg.link (31))`: the `JALR` instruction will overwrite the intended return address that was placed in the link register 31 by the line `Reg.link = savedPC`. Since the callee called by the `JALR` instruction returns to the `JR(Reg.link (31))` instruction using its own `JR(Reg.link (31))` instruction at the end of the callee, we can eliminate the extra step and make the callee's

`JR(Reg.link (31))` return directly to where we wanted the caller's `JR(Reg.link (31))` to return to. This is achieved by replacing the `JALR` instruction with just a `JR` instruction: the address that we will pass to the callee in the link register 31 will directly be the address that the caller should return to, rather than the address of the `JR(Reg.link (31))` instruction in the caller. The pseudocode now looks like this:

```
// (tail) call
evaluate arguments
// caller's epilogue
Reg.link = savedPC
Reg.framePointer = dynamicLink
Stack.pop // caller's frame
Stack.pop // caller's parameters
allocate memory for callee's parameters
write argument values into callee's parameters
LIS/JR
```

A second problem is that the argument values to be stored into the callee's parameters in the second-last line are in temporary variables in the caller's frame, which is popped off the stack just before. The memory locations of these temporary variables are reused for the callee's parameters and there is a risk that the values of some of the temporary variables will be overwritten before we copy them to the correponding parameters of the callee. The following diagram shows an example of one possible layout of the stack before the caller's frame and parameters are popped (on the left) and after space for the callee's parameters is pushed on the stack (on the right):



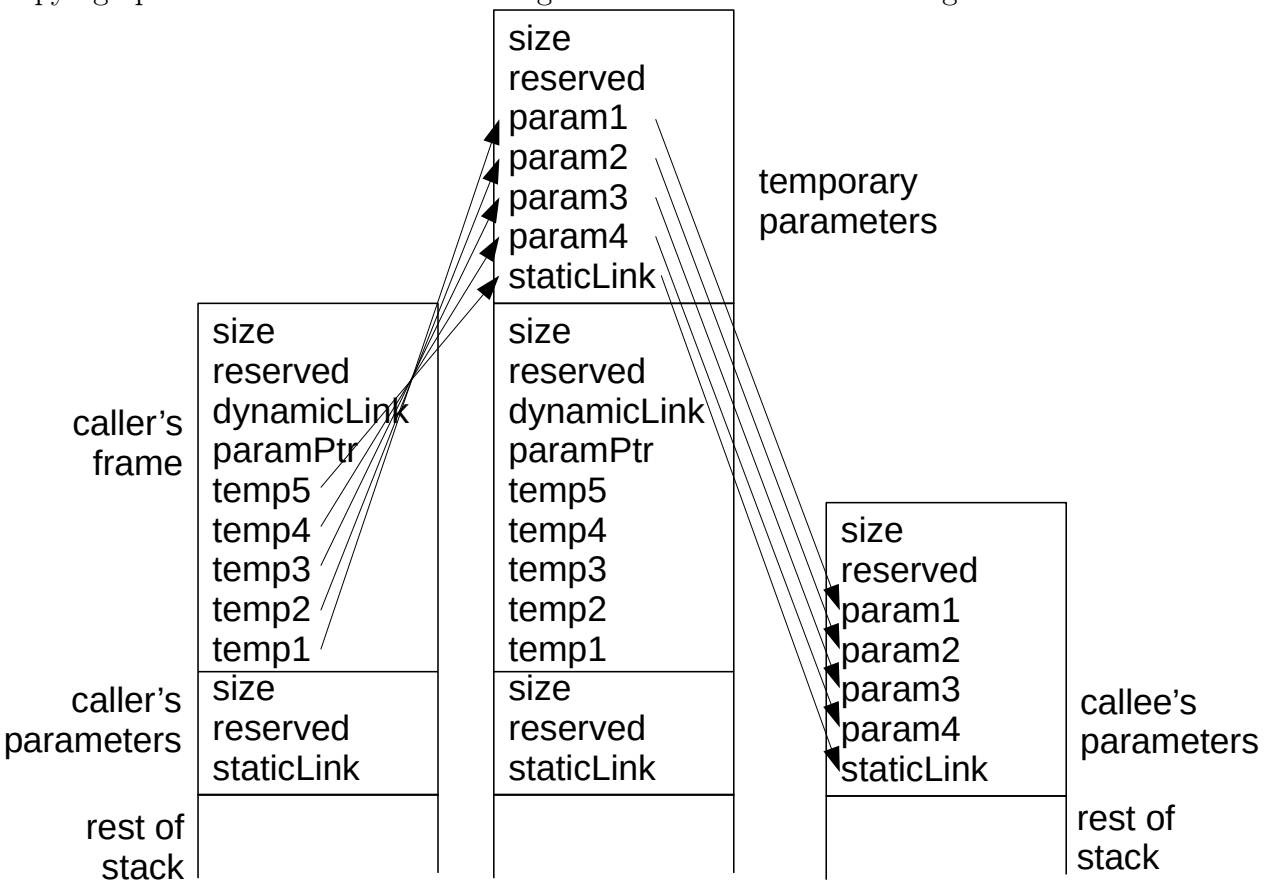
Since we have not specified any particular order in which temporary variables are allocated in a stack frame, one possible order is the one shown in the diagram. In this case, when we write argument values into callee's parameters, we need to copy temp1 into param1 and temp2 into param2, among other things. But temp1 and param2 are in the same memory location, at the same address, and so are temp2 and param1. If we copy temp1 into param1 first, we will overwrite temp2 before we have a chance to copy it into param2. In the opposite order, if we copy temp2 into param2 first, we will overwrite temp1 before we have a chance

to copy it into param1.

One way to fix this issue is to allocate a temporary copy of the callee parameters first before popping the caller's frame from the stack:

```
// (tail) call
evaluate arguments
allocate memory for temporary parameters
write argument values into temporary parameters
// caller's epilogue
Reg.link = savedPC
Reg.framePointer = dynamicLink
Stack.pop // temporary parameters
Stack.pop // caller's frame
Stack.pop // caller's parameters
allocate memory for callee's parameters
copy temporary parameters into callee's parameters
LIS/JR
```

The stack then evolves as shown in the following diagram. We first allocate a temporary copy of the callee parameters, then copy the argument values into it, then pop the temporary parameters, the caller's frame, and the caller's parameters off the stack, and finally copy the temporary parameters into the final parameters for the callee. This ensures that none of the copying operations will overwrite an argument value with another argument value.



Notice that this technique is still risky because it copies values from the temporary param-

eters to the callee's parameters *after* the temporary parameters have already been popped off the stack. Since the stack pointer has been moved below the temporary parameters, any code that writes data on the stack could overwrite the temporary parameters, so we must be very careful not to use the stack for any other purpose in between between these two operations.

A remaining consideration is how this code needs to change when either the caller, the callee, or both have their frame and parameters stored on the heap rather than on the stack. This is left as an exercise for those who implement tail call optimization in Assignment 6. As a hint, the main change is that some of the three `Stack.pop` operations need to be removed in these cases.

As you can see, the result of this general tail call optimization is quite complicated. Some compilers optimize tail calls only in the special case of direct recursion, when the caller and the callee are the same procedure. This special case can be optimized with significantly simpler code: since the caller and the callee are the same procedure, their frames and parameters have the same size and layout, so instead of popping memory for the caller and pushing memory for the callee onto the stack, the optimized code can just directly reuse the memory of the caller's parameters and frame for the callee.