15 Functional Programming Languages

func-tion: a mathematical correspondence that assigns exactly one element of one set to each element of the same or another set

Webster's Dictionary

The mathematical notion of function is that if $f(x) = a$ "this time," then $f(x) = a$ "next time"; there is no other value equal to $f(x)$. This allows the use of *equational reasoning* familiar from algebra: that if $a = f(x)$ then $g(f(x), f(x))$ is equivalent to $g(a, a)$. *Pure functional* programming languages encourage a kind of programming in which equational reasoning works, as it does in mathematics.

Imperative programming languages have similar syntax: $a \leftarrow f(x)$. But if we follow this by $b \leftarrow f(x)$ there is no guarantee that $a = b$; the function f can have *side effects* on global variables that make it return a different value each time. Furthermore, a program might assign into variable x between calls to $f(x)$, so $f(x)$ really means a different thing each time.

Higher-order functions. Functional programming languages also allow functions to be passed as arguments to other functions, or returned as results. Functions that take functional arguments are called *higher-order* functions.

Higher-order functions become particularly interesting if the language also supports *nested functions* with *lexical scope* (also called *block structure*). As in Tiger, lexical scope means that each function can refer to variables and parameters of any function in which it is nested. A *higher-order functional language* is one with nested scope and higher-order functions.

What is the essence of functional programming: is it equational reasoning or is it higher-order functions? There is no clear agreement about the answer to this question. In this chapter I will discuss three different flavors of "functional" language:

Fun-Tiger The Tiger language with higher-order functions. Because side effects are still permitted (and thus, equational reasoning won't work), this is an *impure, higher-order functional language;* other such languages are Scheme, ML, and Smalltalk.

PureFun-Tiger A language with higher-order functions and no side effects, capturing the essence of *strict, pure functional languages* (like the pure functional subset of ML).

Lazy-Tiger A *non-strict, pure functional language* that uses lazy evaluation like the language Haskell. Non-strict pure functional languages support equational reasoning very well (see Section 15.7).

A *first-order, pure functional language* such as SISAL supports equational reasoning but not higher-order functions.

15.1 A SIMPLE FUNCTIONAL LANGUAGE

To make the new language Fun-Tiger, we add *function types* to Tiger:

$$\begin{array}{lll} ty & \rightarrow & ty \texttt{ -> } ty \\ & \rightarrow & \texttt{(} ty \{\texttt{,}\ ty\} \texttt{)} \texttt{ -> } ty \\ & \rightarrow & \texttt{()} \texttt{ -> } ty \end{array}$$

The type `int->string` is the type of functions that take a single integer argument and return a string result. The type `(int,string)->intarray` describes functions that take two arguments (one integer, one string) and return an `intarray` result. The `getchar` function has type `()->string`.

Any variable can have a functional type; functions can be passed as arguments and returned as results. Thus, the type `(int->int)->int->int` is perfectly legal; the `->` operator is right-associative, so this is the type of functions that take an `int->int` argument and return an `int->int` result.

We also modify the format of a CALL expression, so that the function being called is an arbitrary expression, not just an identifier:

$$exp \rightarrow exp\ (\ exp\ \{\ ,\ exp\ \}\ )$$
$$exp \rightarrow exp\ (\ )$$

```
let
    type intfun = int -> int

    function add(n: int) : intfun =
       let function h(m: int) : int = n+m
        in h
       end

    var addFive : intfun := add(5)
    var addSeven : intfun := add(7)
    var twenty := addFive(15)
    var twentyTwo := addSeven(15)

    function twice(f: intfun) : intfun =
       let function g(x: int) : int = f(f(x))
        in g
       end

    var addTen : intfun := twice(addFive)

    var seventeen := twice(add(5))(7)
    var addTwentyFour := twice(twice(add(6)))

 in addTwentyFour(seventeen)
end
```

PROGRAM 15.1. A Fun-Tiger program.

Program 15.1 illustrates the use of function types. The function `add` takes an integer argument `n` and returns a function `h`. Thus, `addFive` is a version of `h` whose `n` variable is 5, but `addSeven` is a function $\texttt{h}(x) = 7 + x$. The need for each different instance of `h` to "remember" the appropriate value for a *nonlocal* variable `n` motivates the implementation technique of *closures*, which is described later.

The function `twice` takes an argument `f` that is a function from `int` to `int`, and the result of `twice(f)` is a function `g` that applies `f` twice. Thus, `addTen` is a function $\texttt{g}(x) = \texttt{addFive}(\texttt{addFive}(x))$. Each instance of $\texttt{g}(x)$ needs to remember the right `f` value, just as each instance of `h` needs to remember `n`.

15.2 CLOSURES

In languages (such as C) without nested functions, the run-time representation of a function value can be the address of the machine code for that function. This address can be passed as an argument, stored in a variable, and so on; when it is time to call the function, the address is loaded into a machine register, and the "call to address contained in register" instruction is used.

In the Tree intermediate representation, this is easy to express. Suppose the function starts at label L_{123}; we assign the address into a variable t_{57} using

$$\text{MOVE}(\text{TEMP}(t_{57}), \text{NAME}(L_{123}))$$

and then call the function with something like

$$\text{CALL}(\text{TEMP}(t_{57}), \ldots \textit{parameters} \ldots).$$

But this will not work for nested functions; if we represent the `h` function by an address, in what outer frame can it access the variable `n`? Similarly, how does the `g` function access the variable `f`?

The solution is to represent a function-variable as *closure*: a record that contains the machine-code pointer and a way to access the necessary nonlocal variables. One simple kind of closure is just a pair of code pointer and static link; the nonlocal variables can be accessed by following the static link. The portion of the closure giving access to values of variables is often called the *environment*.

Closures need not be based on static links; any other data structure that gives access to nonlocal variables will do. Using static links has some serious disadvantages: it takes a chain of pointer dereferences to get to the outermost variables, and the garbage collector cannot collect the intermediate links along this chain even if the program is going to use only the outermost variables. However, in this chapter I will use static-link closures for simplicity.

HEAP-ALLOCATED ACTIVATION RECORDS

Using static links in closures means that the activation record for `add` must not be destroyed when `add` returns, because it still serves as the environment for `h`. To solve this problem, we could create activation records on the heap instead of on the stack. Instead of explicitly destroying `add`'s frame when `add` returns, we would wait until the garbage collector determines that it is safe to reclaim the frame; this would happen when all the pointers to `h` disappear.

A refinement of this technique is to save on the heap only those variables that *escape* (that are used by inner-nested functions). The stack frame will hold spilled registers, return address, and so on, and also a pointer to the *escaping-variable record.* The escaping-variable record holds (1) any local variables that an inner-nested procedure might need and (2) a static link to the environment (escaping-variable record) provided by the enclosing function; see Figure 15.2.

Modifications to the Tiger compiler. In each Fun-Tiger function we make a temporary called the *escaping-variables pointer* or EP that will point to the record of escaping variables. All static link computations, whether to access nonlocal variables or to compute a static link to pass to some other function, will be based on the EP, not the FP. The EP itself is a nonescaping local temporary that will be spilled as needed, just like any other temporary. The static-link formal parameter passed to this function escapes (as does the static link of an ordinary Tiger function) since inner nested functions need to access it; thus, the static link is stored into the escaping-variables record.

In the `Frame` module of the compiler, the interface functions that create formals and locals (`newFrame` and `allocLocal`) must be modified to make accesses (for escaping variables) that are offsets from EP instead of FP. The escaping-variables record must be allocated by instructions produced in `procEntryExit1`.

15.3 IMMUTABLE VARIABLES

The Fun-Tiger language has higher-order functions with nested scope, but it is still not really possible to use *equational reasoning* about Fun-Tiger programs. That is, $f(3)$ may return a different value each time. To remedy this situation, we prohibit *side effects* of functions: when a function is called, it must return a result without changing the "world" in any observable way.

Thus, we make a new *pure functional programming* language PureFun-Tiger, in which the following are prohibited:

- ⊘ Assignments to variables (except as initializations in `var` declarations);
- ⊘ Assignments to fields of heap-allocated records;
- ⊘ Calls to external functions that have visible effects: `print`, `flush`, `getchar`, `exit`.

This seems rather Draconian: how is the program to get any work done?

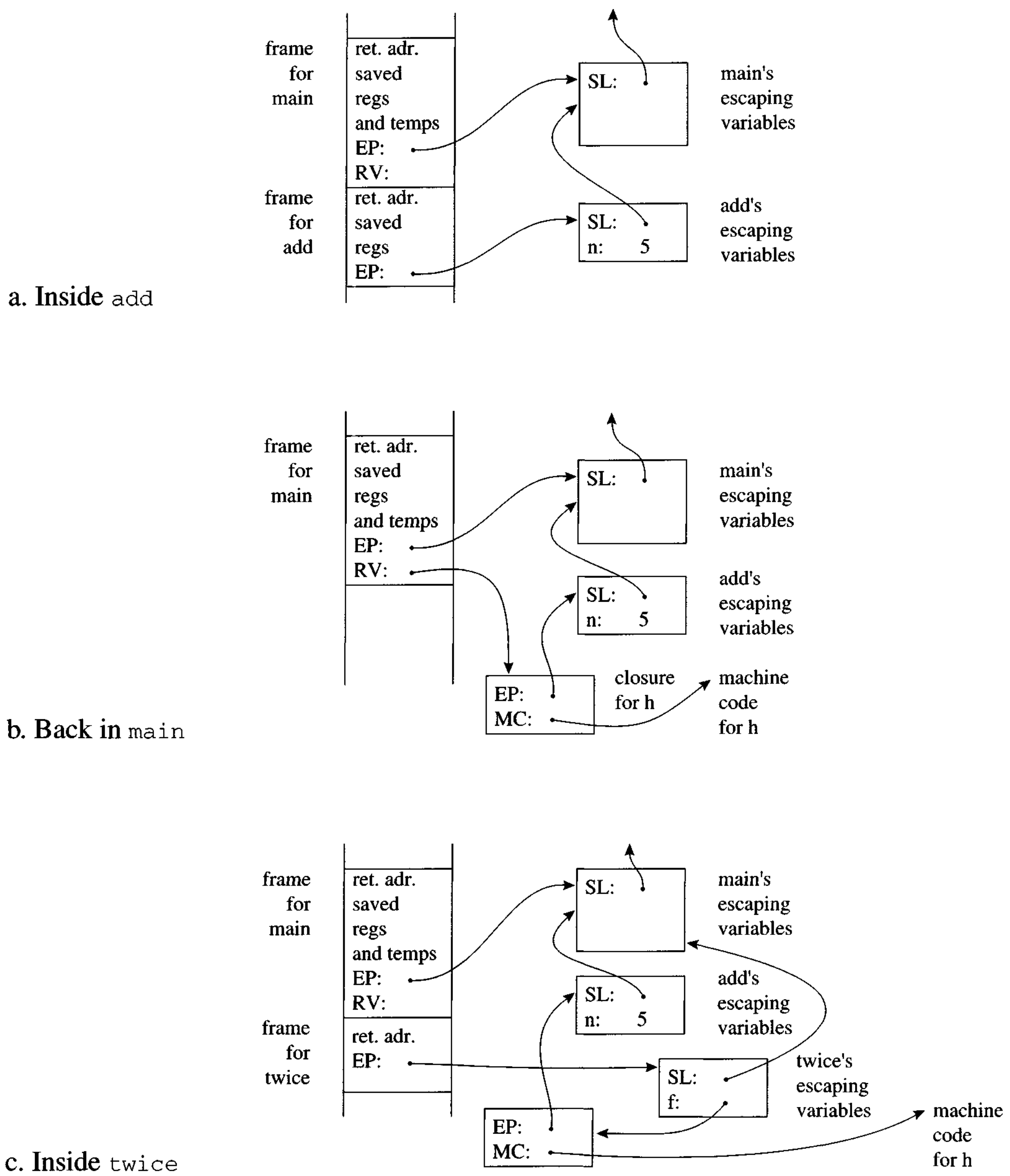

FIGURE 15.2. Closures for execution of `twice(add(5))`. SL=static link; RV=return value; EP=escaping-variables-pointer or environment-pointer.

```
type key = string
type binding = int
type tree = {key: key,
             binding: binding,
             left: tree,
             right: tree}

function look(t: tree, k: key)
                       : binding =
  if k < t.key
       then look(t.left,k)
  else if k > t.key
       then look(t.right,k)
  else t.binding

function enter(t: tree, k: key,
               b: binding) =
  if k < t.key
    then if t.left=nil
      then t.left :=
              tree{key=k,
                   binding=b,
                   left=nil,
                   right=nil}
      else enter(t.left,k,b)
  else if k > t.key
    then if t.right=nil
      then t.right :=
              tree{key=k,
                   binding=b,
                   left=nil,
                   right=nil}
      else enter(t.right,k,b)
  else t.binding := b
```

(a) Imperative

```
type key = string
type binding = int
type tree = {key: key,
             binding: binding,
             left: tree,
             right: tree}

function look(t: tree, k: key)
                       : binding =
  if k < t.key
       then look(t.left,k)
  else if k > t.key
       then look(t.right,k)
  else t.binding

function enter(t: tree, k: key,
               b: binding) : tree =
  if k < t.key
    then
     tree{key=t.key,
          binding=t.binding,
          left=enter(t.left,k,b),
          right=t.right}

  else if k > t.key
    then
     tree{key=t.key,
          binding=t.binding,
          left=t.left,
          right=enter(t.right,k,b)}
  else tree{key=t.key,
            binding=b,
            left=t.left,
            right=t.right}
```

(b) Functional

PROGRAM 15.3. Binary search trees implemented in two ways.

```
type answer
type stringConsumer = string -> answer
type cont = () -> answer

function getchar(c: stringConsumer) : answer
function print(s: string, c: cont) : answer
function flush(c: cont) : answer
function exit() : answer
```

PROGRAM 15.4. Built-in types and functions for PureFun-Tiger.

To program without assignments, in a functional style, you produce new values instead of updating old ones. For example, Program 15.3 shows the implementation of binary search trees in imperative and functional styles. As explained in Section 5.1 (page 108), the imperative program updates a tree-node, but the functional program returns a new tree much like the old one, though the path from the root to a "new" leaf has been copied. If we let `t1` be the tree in Figure 5.4a on page 108, we can say

```
var t2 := enter(t1,"mouse",4)
```

and now `t1` and `t2` are both available for the program to use. On the other hand, if the program returns `t2` as the result of a function and discards `t1`, then the root node of `t1` will be reclaimed by the garbage collector (the other nodes of `t1` will not be reclaimed, because they are still in use by tree `t2`).

Similar techniques can allow functional programs to express the same wide variety of algorithms that imperative programs can, and often more clearly, expressively and concisely.

CONTINUATION-BASED I/O

Producing new data structures instead of updating old ones makes it possible to obey the "no assignments" rules, but how is the program to do input/output? The technique of *continuation-based I/O* expresses input/output in a functional framework. As shown in Program 15.4, the predefined types and functions in PureFun-Tiger rely on the notion of an `answer`: this is the "result" returned by the *entire program*.

The built-in `getchar` function does not return a string (as in Tiger); instead, `getchar` takes an argument that is a `stringConsumer` and passes the newly read character to that consumer. Whatever answer the consumer produces will also be the answer of the `getchar`.

```
let
type intConsumer = int -> answer

function isDigit(s : string) : int =
      ord(s)>=ord("0") & ord(s)<=ord("9")

function getInt(done: intConsumer) =
 let function nextDigit(accum: int) =
       let function eatChar(dig: string) =
            if isDigit(dig)
             then nextDigit(accum*10+ord(dig))
             else done(accum)
        in getchar(eatChar)
       end
  in nextDigit(0)
 end

function putInt(i: int, c: cont) =
  if i=0 then c()
  else let var rest := i/10
           var dig := i - rest * 10
           function doDigit() = print(chr(dig), c)
        in putInt(rest, doDigit)
       end

function factorial(i: int) : int =
  if i=0 then 1 else i * factorial(i-1)

function loop(i) =
  if i > 12 then exit()
  else let function next() = getInt(loop)
        in putInt(factorial(i), next)
       end
in
   getInt(loop)
end
```

PROGRAM 15.5. PureFun-Tiger program to read i, print $i!$.

Similarly, `print` takes a string to print as well as a *continuation* (`cont`); `print` outputs a string and then calls the `cont` to produce an answer.

The point of these arrangements is to allow input/output while preserving equational reasoning. Interestingly, input/output is now "visible" to the type-checker: any function which does I/O will have `answer` in its result type.

LANGUAGE CHANGES

The following modifications of Fun-Tiger make the new language PureFun-Tiger:

- Add the predefined types `answer`, `stringConsumer`, and `cont`; and modify the types of the predefined I/O functions – as shown in Program 15.4.
- A "procedure" (a function without an explicit return type) is now considered to return type `answer`.
- Assignment statements, **while** loops, **for** loops, and compound statements (with semicolon) are deleted from the language.

Program 15.5 shows a complete PureFun-Tiger program that loops, reading integers and printing the factorial of each integer, until an integer larger than 12 is input.

OPTIMIZATION OF PURE FUNCTIONAL LANGUAGES

Because we have only deleted features from Fun-Tiger, and not added any new ones (except changing some predefined types), our Fun-Tiger compiler can compile PureFun-Tiger right away. And, in general, functional-language compilers can make use of the same kinds of optimizations as imperative-language compilers: inline expansion, instruction selection, loop-invariant analysis, graph-coloring register allocation, copy propagation, and so on. Calculating the control-flow graph can be a bit more complicated, however, because much of the control flow is expressed through function calls, and some of these calls may to be function-variables instead of statically defined functions.

A PureFun-Tiger compiler can also make several kinds of optimizations that a Fun-Tiger compiler cannot, because it can take advantage of equational reasoning.

Consider this program fragment, which builds a record `r` and then later fetches fields from it:

```
type recrd = {a: ···, b: ···}

var a1 := 5
var b1 := 7
var r := recrd{a := a1, b := b1}

var x := f(r)

var y :=  r.a + r.b
```

```
let
    type list = {head: int, tail: list}
    type observeInt  = (int,cont) -> answer

    function doList(f: observeInt, l: list, c: cont) =
      if l=nil then c()
      else let function doRest() = doList(f, l.tail, c)
             in f(l.head, doRest)
            end

    function double(j: int) : int = j+j

    function printDouble(i: int, c: cont) =
        let function again() = putInt(double(i),c)
         in putInt(i, again)
        end

    function printTable(l: list, c: cont) =
       doList(printDouble, l, c)

    var mylist := ···

 in printTable(mylist, exit)
end
```

PROGRAM 15.6. `printTable` in PureFun-Tiger.

In a pure functional language, the compiler knows that when the computation of `y` refers to `r.a` and `r.b`, it is going to get the values `a1` and `b1`. In an imperative (or impure functional) language, the computation `f(r)` might assign new values to the fields of `r`, but not in PureFun-Tiger.

Thus, within the scope of `r` every occurrence of `r.a` can be replaced with `a1`, and similarly `b1` can be substituted for `r.b`. Also, since no other part of the program can assign any new value to `a1`, it will contain the same value (5) for all time. Thus, 5 can be substituted for `a1` everywhere, and 7 for `b1`. Thus, we end up with `var y := 5+7` which can be turned into `var y := 12`; thus, 12 can be substituted for `y` throughout its scope.

The same kind of substitution works for imperative languages too; it's just that a compiler for an imperative language is often not sure whether a field or variable is updated between the point of definition and the point of use. Thus, it must conservatively approximate – assuming that the variable may have been modified – and thus, in most cases, the substitution cannot be performed. See also *alias analysis* (Section 17.5).

```
let
 type list = {head: int,
              tail: list}

 function double(j: int): int =
      j+j

 function printDouble(i: int) =
      (putInt(i);
       putInt(double(i)))

 function printTable(l: list) =
    while l <> nil
       do (printDouble(l.head);
           l := l.tail)

 var mylist := ···

in printTable(mylist)
end
```

(a) As written

```
let
 type list = {head: int,
              tail:list}

 function printTable(l: list) =
    while l <> nil
       do let var i := l.head
           in putInt(i);
              putInt(i+i);
              l := l.tail
          end

 var mylist := ···

in printTable(mylist)
end
```

(b) Optimized

PROGRAM 15.7. Regular Tiger `printTable`.

The ML language has pure-functional records, which cannot be updated and on which this substitution transformation is always valid, and also has updatable reference cells, which can be assigned to and which behave like records in a conventional imperative language.

15.4 INLINE EXPANSION

Because functional programs tend to use many small functions, and especially because they pass functions from one place to another, an important optimization technique is *inline expansion* of function calls: replacing a function call with a copy of the function body.

For example, in Program 15.6, an `observeInt` is any function (like the `putInt` of Program 15.5) that "observes" an integer and then continues. `doList` is a function that applies an observer `f` to a list `l`, and then continues. In this case, the observer is not `putInt` but `printDouble`, which prints i followed by $2i$. Thus, `printTable` prints a table of integers, each followed by its double.

For comparison, Program 15.7a is a regular Tiger program that does the same thing.

Program 15.6 uses a generic list-traverser, `doList`, for which any function can be plugged in. Although in this case `printDouble` is used, the same program could reuse `doList` for other purposes that print or "observe" all the integers in the list. But Program 15.7a lacks this flexibility – it calls `printDouble` directly, because the ordinary Tiger language lacks the ability to pass functions as arguments.

If compiled naively, the pure-functional program – which passed `printDouble` as an argument – will do many more function calls than the imperative program. By using inline expansion and tail-call optimizations (described in Section 15.6), Program 15.6 can be optimized into machine instructions equivalent to the efficient loop of Program 15.7b.

Avoiding variable capture. We must be careful about variable names when doing inlining in Tiger (or ML) where a local declaration creates a "hole" in the scope of an outer variable:

```
1 let var x := 5
2     function g(y: int): int =
3        y+x
4     function f(x: int): int =
5        g(1)+x
6  in f(2)+x
7 end
```

The formal parameter `x` on line 4 creates a hole in the scope of the variable `x` declared on line 1, so that the `x` on line 5 refers to the formal parameter, not the variable. If we were to inline-expand the call to `g(1)` on line 5 by substituting the body of `g` for the call, we could not simply write `1+x`, for then we'd have

```
4     function f(x: int) : int =
5        (1+x)+x
```

but the first `x` on line 5 is now incorrectly referring to `f`'s parameter instead of the variable declared on line 1.

To solve this problem, we could first rename, or *α-convert*, the formal parameter of `f`, then perform the substitution:

```
1 let var x := 5                 let var x := 5
2     function g(y:int):int=         function g(y:int):int=
3        y+x                            y+x
4     function f(a:int):int=         function f(a:int):int=
5        g(1)+a                         (1+x)+a
6  in f(2)+x                      in f(2)+x
7 end                            end
```

(a) When the actual parameters are simple variables $i_1, \ldots, i_n$.
Within the scope of:

`function` $f(a_1, \ldots, a_n) = B$

the expression

$f(i_1, \ldots, i_n)$

rewrites to

$B[a_1 \mapsto i_1, \ldots, a_n \mapsto i_n]$

(b) When the actual parameters are nontrivial expressions, not just variables.
Within the scope of:

`function` $f(a_1, \ldots, a_n) = B$

the expression

$f(E_1, \ldots, E_n)$

rewrites to

```
let var $i_1 := E_1$
      ⋮
    var $i_n := E_n$
in  $B[a_1 \mapsto i_1, \ldots, a_n \mapsto i_n]$
end
```

where $i_1, \ldots, i_n$ are previously unused names.

ALGORITHM 15.8. Inline expansion of function bodies. We assume that no two declarations declare the same name.

Alternately, we can rename the *actual* parameters instead of the formal parameters, and define the substitution function to avoid substituting for x inside the scope of a new definition of x.

But the best solution of all for avoiding variable capture is to have an earlier pass of the compiler rename all variables so that the same variable-name is never declared twice. This simplifies reasoning about, and optimizing, the program.

Rules for inlining. Algorithm 15.8 gives the rules for inline expansion, which can apply to imperative or functional programs. The function body B is used in place of the function call $f(\ldots)$, but within this copy of B, each actual parameter is substituted for the corresponding formal parameter. When the actual parameter is just a variable or a constant, the substitution is very simple (Algorithm 15.8a). But when the actual parameter is a nontrivial expression, we must first assign it to a new variable (Algorithm 15.8b).

For example, in Program 15.6 the function-call `double(i)` can be replaced by a copy of `j+j` in which each `j` is replaced by the actual parameter `i`. Here we have used Algorithm 15.8a, since `i` is a variable, not a more complicated expression.

Suppose we wish to inline expand `double(g(x))`; if we improperly use Algorithm 15.8a, we obtain `g(x)+g(x)`, which computes `g(x)` twice. Even though the principle of equational reasoning assures that we will compute the same result each time, we do not wish to slow down the computation by repeating the (potentially expensive) computation `g(x)`. Instead, Algorithm 15.8b yields

```
let i := g(x) in i+i end
```

which computes `g(x)` only once.

In an imperative program, not only is `g(x)+g(x)` slower than

```
let i := g(x) in i+i end
```

but – because `g` may have side effects – it may compute a different result! Again, Algorithm 15.8b does the right thing.

Dead function elimination. If all the calls to a function (such as `double`) have been inline expanded, and if the function is not passed as an argument or referenced in any other way, the function itself can be deleted.

Inlining recursive functions. Inlining `doList` into `printTable` yields this new version of `printTable`:

```
function printTable(l: list, c: cont) =
  if l=nil then c()
  else let function doRest() =
                    doList(printDouble, l.tail, c)
        in printDouble(l.head, doRest)
       end
```

This is not so good: `printTable` calls `printDouble` on `l.head`, but to process `l.tail` it calls `doList` as before. Thus, we have inline expanded *only the first iteration of the loop*. We would rather have a fully customized version of `doRest`; therefore, we do not inline expand in this way.

For recursive functions we use a *loop-preheader* transformation (Algorithm 15.9). The idea is to split *f* into two functions: a *prelude* called from outside, and a *loop header* called from inside. Every call to the loop header will be a recursive call from within itself, except for a single call from the prelude. Applying this transformation to `doList` yields

```
function f(a_1, ..., a_n) =        function f(a'_1, ..., a'_n) =
  B                          →       let function f'(a_1, ..., a_n) =
                                           B[f ↦ f']
                                     in f'(a'_1, ..., a'_n)
                                     end
```

ALGORITHM 15.9. Loop-preheader transformation.

If every use of f' within B is of the form $f'(E_1, \ldots, E_{i-1}, a_i, E_{i+1}, \ldots, E_n)$ such that the ith argument is always a_i, then rewrite

```
function f(a'_1, ..., a'_n) =                 function f(a'_1, ..., a'_{i-1}, a_i, a'_{i+1}, ..., a'_n) =
  let function f'(a_1, ..., a_n) = B    →       let function f'(a_1, ..., a_{i-1}, a_{i+1}, ..., a_n) = B
  in f'(a'_1, ..., a'_n)                        in f'(a'_1, ..., a'_{i-1}, a'_{i+1}, ..., a'_n)
  end                                           end
```

where every call $f'(E_1, \ldots, E_{i-1}, a_i, E_{i+1}, \ldots, E_n)$ within B is rewritten as $f'(E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n)$.

ALGORITHM 15.10. Loop-invariant hoisting.

```
function doList(fX: observeInt, lX: list, cX: cont) =
 let function doListX(f: observeInt, l: list, c: cont) =
        if l=nil then c()
        else let function doRest() = doListX(f, l.tail, c)
             in f(l.head, doRest)
             end
  in doListX(fX,lX,cX)
 end
```

where the new `doList` is the prelude, and `doListX` is the loop header. Notice that the prelude function contains the entire loop as an internal function, so that when any call to `doList` is inline expanded, a new copy of `doListX` comes along with it.

Loop-invariant arguments. In this example, the function `doListX` is passing around the values `f` and `c` that are invariant – they are the same in every recursive call. In each case, `f` is `fX` and `c` is `cX`. A *loop-invariant hoisting* transformation (Algorithm 15.10) can replace every use of `f` with `fX`, and `c` with `cX`).

Applying this transformation to `doList` yields

```
function doList(f: observeInt, lX: list, c: cont) =
 let function doListX(l: list) =
       if l=nil then c()
       else let function doRest() = doListX(l.tail)
            in f(l.head, doRest)
            end
  in doListX(lX)
 end
```

Finally, in `printTable` when the call `doList(printDouble,l,c)` is inlined, we obtain:

```
function printTable(l: list, c: cont) =
 let function doListX(l: list) =
       if l=nil then c()
       else let function doRest() = doListX(l.tail)
            in printDouble(l.head, doRest)
            end
  in doListX(l)
 end
```

Cascading inlining. In this version of `printTable`, we have `printDouble` applied to arguments (instead of just passed to `doList`), so we can inline expand that call, yielding

```
function printTable(l: list, c: cont) =
 let function doListX(l: list) =
       if l=nil then c()
       else let function doRest() = doListX(l.tail)
            in let var i := l.head
                in let function again() = putInt(i+i,doRest)
                    in putInt(i,again)
                   end
               end
            end
  in doListX(l)
 end
```

Avoiding code explosion. Inline expansion copies function bodies. This generally makes the program bigger. If done indiscriminantly, the size of the program explodes; in fact, it is easy to construct cases where expanding one function call creates new instances that can also be expanded, ad infinitum.

```
function printTable(l: list, c: cont) =
    let function doListX(l: list) =
          if l=nil then c()
          else let function doRest() =
                                doListX(l.tail)
                   var i := l.head
                   function again() =
                                putInt(i+i,doRest)
                in putInt(i,again)
              end
   in doListX(l)
  end
```

PROGRAM 15.11. `printTable` as automatically specialized.

There are several heuristics that can be used to control inlining:

1. Expand only those function-call sites that are very frequently executed; determine frequency either by static estimation (loop-nest depth) or by feedback from an execution profiler.
2. Expand functions with very small bodies, so that the copied function body is not much larger than the instructions that would have called the function.
3. Expand functions called only once; then *dead function elimination* will delete the original copy of the function body.

Unnesting `lets`. Since the Tiger expression

`let` dec_1 `in let` dec_2 `in` exp `end end`

is exactly equivalent to

`let` dec_1 dec_2 `in` exp `end`

we end up with Program 15.11.

The optimizer has taken a program written with abstraction (with a general-purpose `doList`) and transformed it into a more efficient, special-purpose program (with a special-purpose `doListX` that calls `putInt` directly).

15.5 CLOSURE CONVERSION

A function passed as an argument is represented as a *closure*: a combination of a machine-code pointer and a means of accessing the nonlocal variables (also called *free variables*).

Chapter 6 explained the method of static links for accessing free variables, where the static links point directly to the enclosing functions' stack frames. Figure 15.2 shows that the free variables can be kept in a heap-allocated record, separate from the stack frame. Now, for the convenience of the back end of the compiler, we would like to make the creation and access of those free-variable records explicit in the program.

The *closure conversion* phase of a functional-language compiler transforms the program so that none of the functions appears to access free (nonlocal) variables. This is done by turning each free-variable access into a formal-parameter access.

Given a function $f(a_1, \ldots, a_n) = B$ at nesting depth d with escaping local variables (and formal parameters) $x_1, x_2, \ldots, x_n$ and nonescaping variables $y_1, \ldots, y_n$; rewrite into

$$f(a_0, a_1, \ldots, a_n) = \texttt{let var}\ r := \{a_0, x_1, x_2, \ldots, x_n\}\ \texttt{in}\ B'\ \texttt{end}$$

The new parameter a_0 is the static link, now made into an explicit argument. The variable r is a record containing all the escaping variables *and* the enclosing static link. This r becomes the static-link argument when calling functions of depth $d + 1$.

Any use of a nonlocal variable (one that comes from nesting depth $< d$) within B must be transformed into an access of some offset within the record a_0 (in the rewritten function body B').

Function values. Function values are represented as closures, comprising a code pointer and environment. Instead of heap allocating a two-word record to hold these two, when the programmer passes a function as an argument, the compiler should pass the code pointer and environment as two adjacent arguments.

Program 15.12 is the result of closure-converting Program 15.11. We can see that each function creates an explicit record to hold escaping variables. In fact, the function `doListX` creates two different records `r2` and `r3`, because the variables `i` and `doRestC` are not available at the time `r2` must be created. Functions in closure-converted programs access *only* local variables, so that later phases of the compiler need not worry about nonlocal-variable access or static links.

Unknown types of static links in closures. The types of all escaping-variable records are given by record declarations at the top of Program 15.12. But

```
type mainLink = { ··· }
type printTableLink= {SL: mainLink, cFunc: cont, cSL: ?}
type cont = ? -> answer
type doListXLink1 = {SL: printTableLink, l: list}
type doListXLink2 = {SL: doListXLink1, i: int,
                     doRestFunc: cont, doRestSL: doListXLink1}

function printTable(SL: mainLink, l: list, cFunc: cont, cSL: ?) =
    let var r1 := printTableLink{SL=SL,cFunc=cFunc,cSL=cSL}
        function doListX(SL: printTableLink, l: list) =
          let var r2 := doListXLink1{SL: printTableLink, l=l}
           in if r2.l=nil then SL.cFunc(SL.cSL)
              else let function doRest(SL: doListXLink1) =
                           doListX(SL.SL, SL.l.tail)
                       var i := r2.l.head
                       var r3 := doListXLink2{SL=r2, i=i,
                                      doRestFunc=doRest, doRestSL=r2}
                       function again(SL: doListXLink2) =
                           putInt(SL.SL.SL, SL.i+SL.i,
                                  SL.doRest.func, SL.doRestSL)
                    in putInt(SL.SL,i, again,r3)
                   end
    in doListX(r1,l)
   end
```

PROGRAM 15.12. `printTable` after closure conversion.

what is the type of `cont`'s static link argument? It must be the type of the escaping-variable record of the function that encloses the `cont` function.

But there are several different functions of type `cont`:

- the `c` argument of `printTable`, which comes from `main` (examination of Program 15.6 shows that this will in fact be the `exit` function);
- `doRest`;
- and `again`.

Each of these functions has a *different* kind of static link record. Thus, the type of the `SL` field of `contClosure` varies, and *cannot always be known* by the caller. The type of the static-link argument of the `cont` type is shown as a question-mark. That is, although we can write closure-converted Fun-Tiger or PureFun-Tiger programs in Tiger syntax, these programs do not type-check in a conventional sense.

15.6 EFFICIENT TAIL RECURSION

Functional programs express loops and other control flow by function calls. Where Program 15.7b has a **while** loop, Program 15.12 has a function call to `doListX`. Where Program 15.7b's `putInt` simply returns to its two points of call within `printTable`, Program 15.11 has continuation functions. The Fun-Tiger compiler must compile the calls to `doListX`, `doRest`, and `again` as efficently as the Tiger compiler compiles loops and function returns.

Many of the function calls in Program 15.11 are in *tail position.* A function call $f(x)$ within the body of another function $g(y)$ is in tail position if "calling f is the last thing that g will do before returning." More formally, in each of the following expressions, the B_i are in tail contexts, but the C_i are not:

1. `let var` x `:=` C_1 `in` B_1 `end`
2. C_1`(`C_2`)`
3. `if` C_1 `then` B_1 `else` B_2
4. C_1 `+` C_2

For example, C_2 in expression 4 is not in a tail context, even though it seems to be "last," because after C_2 completes there will still need to be an **add** instruction. But B_1 in expression 3 is in a tail context, even though it is not "last" syntactically.

If a function call $f(x)$ is in a tail context with respect to its enclosing expression, and that expression is in a tail context, and so on all the way to the body of the enclosing function definition `function` $g(y) = B$, then $f(x)$ is a tail call.

Tail calls can be implemented more efficiently than ordinary calls. Given

```
g(y) = let var x := h(y) in f(x) end
```

Then `h(y)` is not a tail call, but `f(x)` is. When `f(x)` returns some result `z`, then `z` will also be the result returned from `g`. Instead of pushing a new return address for `f` to return to, `g` could just give `f` the return address given to `g`, and have `f` return there directly.

That is, a tail call can be implemented more like a jump than a call. The steps for a tail call are:

1. Move actual parameters into argument registers.
2. Restore callee-save registers.
3. Pop the stack frame of the calling function, *if it has one.*
4. Jump to the callee.

printTable:	allocate record `r1`	printTable:	allocate stack frame
	jump to doListX		jump to whileL
doListX:	allocate record `r2`	whileL:	
	if `l=nil` goto doneL		if `l=nil` goto doneL
	`i := r2.l.head`		`i := l.head`
	allocate record `r3`		
	jump to `putInt`		call `putInt`
again:	add `SL.i+SL.i`		add `i+i`
	jump to `putInt`		call `putInt`
doRest:	jump to doListX		jump to whileL
doneL :	jump to `SL.cFunc`	doneL:	return

(a) Functional program (b) Imperative program

FIGURE 15.13. `printTable` as compiled.

In many cases, item 1 (moving parameters) is eliminated by the copy-propagation (coalescing) phase of the compiler. Often, items 2 and 3 are eliminated because the calling function has no stack frame – any function that can do all its computation in caller-save registers needs no frame. Thus, a tail call can be as cheap as a jump instruction.

In Program 15.12, *every* call is a tail call! Also, none of the functions in this program needs a stack frame. This need not have been true; for example, the call to `double` in Program 15.6 is not in tail position, and this nontail call only disappeared because the inline expander did away with it.

Tail calls implemented as jumps. The compilation of Programs 15.12 and 15.7b is instructive. Figure 15.13 shows that the pure-functional program and the imperative program are executing almost exactly the same instructions! The figure does not show the functional program's fetching from static-link records; and it does not show the imperative program's saving and restoring callee-save registers.

The remaining inefficiency in the functional program is that it creates three heap-allocated records, `r1,r2,r3`, while the imperative program creates only one stack frame. However, more advanced closure-conversion algorithms can succeed in creating only one record (at the beginning of `printTable`). So the difference between the two programs would be little more than a heap-record creation versus a stack-frame creation.

Allocating a record on the garbage-collected heap may be more expensive than pushing and popping a stack frame. Optimizing compilers for functional languages solve this problem in different ways:

- Compile-time *escape analysis* can identify which closure records do not outlive the function that creates them. These records can be stack-allocated. In the case of `printTable`, this would make the "functional" code almost identical to the "imperative" code.
- Or heap allocation and garbage collection can be made extremely cheap. Then creating (and garbage-collecting) a heap-allocated record takes only four or five instructions, making the functional `printTable` almost as fast as the imperative one (see Section 13.7).

15.7 LAZY EVALUATION

Equational reasoning aids in understanding functional programs. One important principle of equational reasoning is *β-substitution:* if $f(x) = B$ with some function body B then any application $f(E)$ to an expression E is equivalent to B with every occurrence of x replaced with E:

$$f(x) = B \quad \text{implies that} \quad f(E) \equiv B[x \mapsto E]$$

But consider the PureFun-Tiger program fragments,

```
let                                 let
  function loop(z:int):int=           function loop(z:int):int=
    if z>0 then z                       if z>0 then z
           else loop(z)                        else loop(z)
  function f(x:int):int=              function f(x:int):int=
    if y>8 then x                       if y>8 then x
           else -y                             else -y
in                                   in if y>8 then loop(y)
   f(loop(y))                                  else -y
end                                 end
```

If the expression B is `if y>8 then x else -y`, and expression E is `loop(y)`, then clearly the program on the left contains $f(E)$ and the program on the right contains $B[x \mapsto E]$. So these programs are equivalent, using equational reasoning.

However, *the programs do not always behave the same!* If $y = 0$, then the program on the right will return 0, but the program on the left will first get stuck in a call to $loop(0)$, which infinite-loops.

Clearly, if we want to claim that two programs are equivalent then they must behave the same. In PureFun-Tiger, if we obtain program A by doing substition on program B, then A and B will never give different results *if they both halt*; but A or B might not halt on the same set of inputs.

To remedy this (partial) failure of equational reasoning, we can introduce *lazy evaluation* into the programming language. Haskell and Miranda are the most widely used lazy languages. A program compiled with lazy evaluation will not evaluate any expression unless its value is demanded by some other part of the computation. In contrast, *strict* languages such as Tiger, PureFun-Tiger, ML, C, and Java evaluate each expression as the control flow of the program reaches it.

To explore the compilation of lazy functional languages, we will use the Lazy-Tiger language. Its syntax is identical to PureFun-Tiger, and its semantics are almost identical, except that lazy evaluation is used in compiling it.

CALL-BY-NAME EVALUATION

Most programming languages (Pascal, C, ML, Java, Tiger, PureFun-Tiger) use *call-by-value* to pass function arguments: to compute $f(g(x))$, first $g(x)$ is computed, and this value is passed to f. But if f did not actually need to use its argument, then computing $g(x)$ will have been unnecessary.

To avoid computing expressions before their results are needed, we can use *call-by-name* evaluation. Essentially, each variable is not a simple value, but is a *thunk:* a function that computes the value on demand. The compiler replaces each expression of type `int` with a function value of type `()->int`, and similarly for all other types.

At each place where a variable is created, the compiler creates a function value; and everywhere a variable is used, the compiler puts a function application.

Thus the Lazy-Tiger program

```
let var a := 5+7  in   a + 10  end
```

is automatically transformed to

```
let function a() = 5+7  in   a() + 10  end
```

Where are variables created? At `var` declarations and at function-parameter bindings. Thus, each `var` turns into a `function`, and at each function-call site, we need a little `function` declaration for each actual-parameter expression.

```
type tree = {key: ()->key,
             binding: ()->binding,
             left: ()->tree,
             right: ()->tree}

function look(t: ()->tree, k: ()->key) : ()->binding =
  if k() < t().key() then look(t().left,k)
  else if k() > t().key() then look(t().right,k)
  else t().binding
```

PROGRAM 15.14. Call-by-name transformation applied to Program 15.3a.

Program 15.14 illustrates this transformation applied to the `look` function of Program 15.3a.

The problem with call-by-name is that each thunk may be executed many times, each time (redundantly) yielding the same value. For example, suppose there is a tree represented by a thunk `t1`. Each time `look(t1,k)` is called, `t1()` is evaluated, which rebuilds the (identical) tree every time!

CALL-BY-NEED

Lazy evaluation, also called *call-by-need*, is a modification of call-by-name that never evaluates the same thunk twice. Each thunk is equipped with a *memo* slot to store the value. When the thunk is first created, the memo slot is empty. Each evaluation of the thunk checks the memo slot: if full, it returns the *memo-ized* value; if empty, it calls the thunk function.

To streamline this process, we will represent a lazy thunk as a two-element record containing a *thunk function* and a *memo slot*. An *unevaluated* thunk contains an arbitrary thunk function, and the memo slot is a static link to be used in calling the thunk function. An *evaluated* thunk has the previously computed value in its memo slot, and its thunk function just returns the memo-slot value.

For example, the Lazy-Tiger declaration `var twenty:=addFive(15)` (in Program 15.1) is compiled in a context where the environment pointer EP will point to a record containing the `addFive` function. The representation of `addFive(15)` is not a function call that will go and compute the answer *now*, but a thunk that will remember how to compute it on demand, *later*. We might translate this fragment of the Lazy-Tiger program into Fun-Tiger as follows:

```
/* EP already points to a record containing addFive */
var twenty := intThunk{func=twentyFunc, memo=EP}
```

which is supported by the auxiliary declarations

```
type intThunk = {func: ?->int, memo: ?}
type intfunc = {func: (?,intThunk)->int, SL: ?}
type intfuncThunk = {func: ?->intfunc, memo: ?}

function evaluatedFunc(th: intThunk) : int =
      th.memo

function twentyFunc(mythunk: intThunk) : int =
 let var EP := mythunk.memo
     var add5thunk : intfuncThunk := EP.addFive
     var add5 : intfunc := add5thunk.func(add5thunk)
     var fifteenThunk:=intThunk{func=evaluatedFunc,memo=15}
     var result : int := add5.func(add5.SL, fifteenThunk)
  in mythunk.memo := result;
     mythunk.func := evaluatedFunc;
     result
 end
```

To *touch* a lazy thunk `t`, we just compute `t.func(t)`. For `t=twenty`, the first time `t` is touched, `twentyFunc(twenty)` will execute, making `twenty.memo` point at the integer result computed by $addFive(15)$ and making `twenty.func` point to the special function `evaluatedFunc`. Any subsequent time that `twenty` is touched, `evaluatedFunc` will simply return the `twenty.memo` field, which will contain the integer 20.

EVALUATION OF A LAZY PROGRAM

Here is a program that uses the `enter` function of Program 15.3b to build a tree mapping {three $\mapsto$ 3!, minusOne $\mapsto$ $(-1)!$}:

```
let function fact(i: int) : int =
        if i=0 then 1 else i * fact(i-1)
    var t1 := enter(nil, "minusOne", fact(-1))
    var t2 := enter(t1,  "three",    fact(3))
 in putInt(look(t2,"three"), exit)
end
```

A curious thing about this program is that `fact(-1)` is undefined. Thus, if this program is compiled by a (strict) PureFun-Tiger compiler, it will infinite-loop (or will eventually overflow the machine's arithmetic as it keeps subtracting 1 from a negative number).

But if compiled by a Lazy-Tiger compiler, the program will succeed, returning three factorial! First, variable `t1` is defined; but this does not actually call `enter` – it merely makes a thunk which will do so on demand. Then, `t2` is defined, which also does nothing but make a thunk. Then a thunk is created for `look(t2,"three")` (but `look` is not actually called).

Finally, a thunk for the expression `putInt(...,exit)` is created. This is the result of the program. But the runtime system then "demands" an `answer` from this program, which can be computed only by calling the outermost thunk. So the body of `putInt` executes, which immediately demands the integer value of its first argument; this causes the `look(t2,"three")` thunk to evaluate.

The body of `look` needs to compare `k` with `t.key`. Since `k` and `t` are each thunks, we can compute an integer by evaluating `k()` and a tree by evaluating `t()`. From the tree we can extract the `key` field, but each field is a thunk, so we must actually do `(t().key)()` to get the integer.

The `t.key` value will turn out to be -1, so `look(t().right,k)` is called. *The program never evaluates the* `binding` *thunk in the* `minusOne` *node,* so `fact(-1)` is never given a chance to infinite-loop.

OPTIMIZATION OF LAZY FUNCTIONAL PROGRAMS

Lazy functional programs are subject to many of the same kinds of optimizations as strict functional programs, or even imperative programs. Loops can be identified (these are simply tail-recursive functions), induction variables can be identified, common subexpressions can be eliminated, and so on.

In addition, lazy compilers can do some kinds of optimizations that strict-functional or imperative compilers cannot, using equational reasoning.

Invariant hoisting. For example, given a loop

```
type intfun = int->int

function f(i: int) : intfun =
  let function g(j: int) = h(i) * j
   in g
  end
```

an optimizer might like to hoist the invariant computation `h(i)` out of the function `g`. After all, `g` may be called thousands of times, and it would be better not to recompute `h(i)` each time. Thus we obtain

```
type intfun = int->int

function f(i: int) : intfun =
  let var hi := h(i)
      function g(j: int) = hi * j
   in g
  end
```

and now each time `g` is called, it runs faster.

This is valid in a lazy language. But in a strict language, this transformation is invalid! Suppose after `var a := f(8)` the function `a` is never called at all; and suppose `h(8)` infinite-loops; before the "optimization" the program would have terminated successfully, but afterward we get a nonterminating program. Of course, the transformation is also invalid in an impure functional language, because `h(8)` might have side effects, and we are changing the number of times `h(8)` is executed.

Dead-code removal. Another subtle problem with strict programming languages is the removal of *dead code*. Suppose we have

```
function f(i: int) : int =
 let var d := g(x)
  in i+2
 end
```

The variable `d` is never used; it is *dead* at its definition. Therefore, the call to `g(x)` should be removed. In a conventional programming language, such as Tiger or Fun-Tiger, we cannot remove `g(x)` because it might have side effects that are necessary to the operation of the program.

In a strict, purely functional language such as PureFun-Tiger, removing the computation `g(x)` could optimize a nonterminating computation into a terminating one! Though this seems benign, it can be very confusing to the programmer. We do not want programs to change their input/output behavior when compiled with different levels of optimization.

In a lazy language, it is perfectly safe to remove dead computations such as `g(x)`.

Deforestation. In any language, it is common to break a program into one module that produces a data structure and another module that consumes it. Program 15.15 is a simple example; `range(i,j)` generates a list of the integers from `i` to `j`, `squares(l)` returns the square of each number, and `sum(l)` adds up all the numbers.

```
type intList = {head: int, tail: intList}
type intfun = int->int
type int2fun = (int,int) -> int

function sumSq(inc: intfun, mul: int2fun, add: int2fun) : int =
let
  function range(i: int, j: int) : intList =
    if i>j then nil else intList{head=i, tail=range(inc(i),j)}

  function squares(l: intList) : intList =
    if l=nil then nil
    else intList{head=mul(l.head,l.head), tail=squares(l.tail)}

  function sum(accum: int, l: intList) : int =
    if l=nil then accum else sum(add(accum,l.head), l.tail)

 in sum(0,squares(range(1,100)))
end
```

PROGRAM 15.15. Summing the squares.

First `range` builds a list of 100 integers; then `squares` builds another list of 100 integers; finally `sum` traverses this list.

It is wasteful to build these lists. A transformation called *deforestation* removes intermediate lists and trees (hence the name) and does everything in one pass. The deforested `sumSq` program looks like this:

```
function sumSq(inc:intfun, mul:int2fun, add:int2fun):int =
 let function f(accum: int, i: int, j: int) : int =
      if i>j then accum else f(add(accum,mul(i,i)),inc(i))
  in f(0,1,100)
 end
```

In impure functional languages (where functions can have side effects) deforestation is not usually valid. Suppose, for example, that the functions `inc`, `mul`, and `add` alter global variables, or print on an output file. The deforestation transformation has rearranged the order of calling these functions; instead of

`inc`(1), `inc`(2), ... `inc`(100),
`mul`(1, 1), `mul`(2, 2), ... `mul`(100, 100),
`add`(0, 1), `add`(1, 4), ... `add`(328350, 10000)

the functions are called in the order

```
mul(1, 1),      add(0, 1),          inc(1),
mul(2, 2),      add(1, 4),          inc(2),
                    ⋮
mul(100, 100), add(328350, 10000), inc(100)
```

Only in a pure functional language is it always legal to make this transformation.

STRICTNESS ANALYSIS

Although laziness allows certain new optimizations, the overhead of thunk creation and thunk evaluation is very high. If no attention is paid to this problem, then the lazy program will run slowly no matter what other optimizations are enabled.

The solution is to put thunks only where they are needed. If a function $f(x)$ is certain to evaluate its argument x, then there is no need to pass a thunk for x; we can just pass an evaluated x instead. We are trading an evaluation now for a certain eventual evaluation.

Definition of strictness. We say a function $f(x)$ is *strict in x* if, whenever some actual parameter a would fail to terminate, then $f(a)$ would also fail to terminate. A multi-argument function $f(x_1, \ldots, x_n)$ is strict in x_i if, whenever a would fail to terminate, then $f(b_1, \ldots, b_{i-1}, a, b_{i+1}, \ldots, b_n)$ also fails to terminate, regardless of whether the b_j terminate.

Let us take an example:

```
function f(x: int, y: int) : int = x + x + y

function g(x: int, y: int) : int = if x>0 then y else x

function h(x: string, y: int) : tree =
                tree{key=x,binding=y,left=nil,right=nil}

function j(x: int) : int = j(0)
```

The function f is *strict* in its argument x, since if the result f(x,y) is demanded then f will certainly touch (demand the value of) x. Similarly, f is strict in argument y, and g is strict in argument x. But g is not strict in its second argument, because g can sometimes compute its result without touching y.

```
function look(t: tree, k: key) : ()->binding =
  if k < t.key() then look(t.left(),k)
  else if k > t.key() then look(t.right(),k)
  else t.binding
```

PROGRAM 15.16. Partial call-by-name using the results of strictness analysis; compare with Program 15.14.

The function `h` is not strict in either argument. Even though it appears to "use" both `x` and `y`, it does not demand (string or integer) values from them; instead it just puts them into a data structure, and it could be that no other part of the program will ever demand values from the `key` or `binding` fields of that particular `tree`.

Curiously, by our definition of strictness, the function `j` is strict in `x` even though it never uses `x`. But the purpose of strictness analysis is to determine whether it is safe to evaluate `x` before passing it to the function `j`: will this cause a terminating program to become nonterminating? In this case, if `j` is going to be called, it will infinite-loop anyway, so it doesn't matter if we perform a (possibly nonterminating) evaluation of `x` beforehand.

Using the result of strictness analysis. Program 15.16 shows the result of transforming the `look` function (of Program 15.3a) using strictness information. A call-by-name transformation has been applied here, as in Program 15.14, but the result would be similar using call-by-need. Function `look` is strict in both its arguments `t` and `key`. Thus, when comparing `k<t.key`, it does not have to *touch* `k` and `t`. However, the `t.key` field still points to a thunk, so it must be touched.

Since `look` is strict, callers of `look` are expected to pass evaluated values, not thunks. This is illustrated by the recursive calls, which must explicitly *touch* `t.left` and `t.right` to turn them from thunks to values.

Approximate strictness analysis. In some cases, such as the functions `f,g,h` above, the strictness or nonstrictness of a function is obvious – and easily determined by an optimizing compiler. But in general, exact strictness analysis is not computable – like exact dynamic liveness analysis (see page 218) and many other dataflow problems.

Thus, compilers must use a conservative approximation; where the exact strictness of a function argument cannot be determined, the argument must be assumed nonstrict. Then a thunk will be created for it; this slows down the

Function M:

$$
\begin{aligned}
M(7,\sigma) &= 1\\
M(\mathtt{x},\sigma) &= \mathtt{x} \in \sigma\\
M(E_1 + E_2,\sigma) &= M(E_1,\sigma) \wedge M(E_2,\sigma)\\
M(\mathtt{record}\{E_1,\ldots,E_n\},\ \sigma) &= 1\\
M(\mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3,\ \sigma) &= M(E_1,\sigma) \wedge (M(E_2,\sigma) \vee M(E_3,\sigma))\\
M(\mathtt{f}(E_1,\ldots,E_n),\ \sigma) &= (\mathtt{f},\ (M(E_1,\sigma),\ldots,M(E_n,\sigma))) \in H
\end{aligned}
$$

Calculation of H:

```
$H \leftarrow \{\}$
repeat
    $done \leftarrow$ true
    for each function $\mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n) = B$
        for each sequence $(b_1,\ldots,b_n)$ of booleans (all $2^n$ of them)
            if $(\mathtt{f},\ (b_1,\ldots,b_n)) \notin H$
                $\sigma \leftarrow \{\mathtt{x}_i \mid b_i = 1\}$        ($\sigma$ is the set of x's corresponding
                if $M(B,\sigma)$                     to 1's in the b vector)
                    $done \leftarrow$ false
                    $H \leftarrow H \cup \{(\mathtt{f},\ (b_1,\ldots,b_n))\}$
until $done$
```

Strictness (after the calculation of H terminates):
f is strict in its ith argument if

$$(\mathtt{f},\ (\underbrace{1,1,\ldots,1}_{i-1},0,\underbrace{1,1,\ldots,1}_{n-i})) \notin H$$

ALGORITHM 15.17. First-order strictness analysis.

program a bit, but at least the optimizer will not have turned a terminating program into an infinite-looping program.

Algorithm 15.17 shows an algorithm for computing strictness. It maintains a set H of tuples of the form $(\mathtt{f},(b_1,\ldots,b_n))$, where n is the number of arguments of f and the b_i are booleans. The meaning of a tuple $(\mathtt{f},(1,1,0))$ is this: if f is called with three arguments (thunks), and the first two may halt but the third never halts, then f may halt.

If (f, (1, 1, 0)) is in the set H, then f might not be strict in its third argument. If (f, (1, 1, 0)) is never put into H, then f must be strict in its third argument.

We also need an auxiliary function to calculate whether an *expression* may terminate. Given an expression E and a set of variables σ, we say that $M(E, \sigma)$ means "E may terminate if all the variables in σ may terminate." If E_1 is i+j, and there is some possibility that the thunks i and j may halt, then it is also possible that E_1 will halt too: $M(\texttt{i} + \texttt{j}, \{\texttt{i}, \texttt{j}\})$ is true. But if E_2 is if k then i else j, where i and j could conceivably halt but k never does, then certainly E_2 will not halt, so $M(E_2, \{\texttt{i}, \texttt{j}\})$ is false.

Algorithm 15.17 will not work on the full Lazy-Tiger language, because it does not handle functions passed as arguments or returned as results. But for *first-order* programs (without higher-order functions), it does a good job of computing (static) strictness. More powerful algorithms for strictness analysis handle higher-order functions.

FURTHER READING

Church [1941] developed the λ-calculus, a "programming language" of nested functions that can be passed as arguments and returned as results. He was hampered by having no machines to compile for.

Closures. Landin [1964] showed how to interpret λ-calculus on an abstract machine, using closures allocated on a heap. Steele [1978] used closure representations specialized to different patterns of function usage, so that in many cases nonlocal variables are passed as extra arguments to an inner function to avoid heap allocating a record. Cousineau et al. [1985] showed how closure conversion can be expressed as a transformation back into the source language, so that closure analysis can be cleanly separated from other phases of code generation.

Static links are actually not the best basis for doing closure conversion; for many reasons it is better to consider each nonlocal variable separately, instead of always grouping together all the variables at the same nesting level. Kranz et al. [1986] performed *escape analysis* to determine which closures can be stack-allocated because they do not outlive their creating function and also integrated closure analysis with register allocation to make a high-performance optimizing compiler. Shao and Appel [1994] integrate closures with the use of callee-save registers to minimize the load/store traffic caused by accessing local and nonlocal variables. Appel [1992, Chapters 10 and 12] has a good overview of closure conversion.

Continuations. Tail calls are particularly efficient and easy to analyze. Strachey and Wadsworth [1974] showed that the control flow of any program (even an imperative one) can be expressed as function calls, using the notion of *continuations*. Steele [1978] transformed programs into *continuation-passing style* early in compilation, turning all function calls into tail calls, to simplify all the analysis and optimization phases of the compiler. Kranz et al. [1986] built an optimizing compiler for Scheme using continuation-passing style; Appel [1992] describes a continuation-based optimizing compiler for ML.

Inline expansion. Cocke and Schwartz [1970] describe inline expansion of function bodies; Scheifler [1977] shows that it is particularly useful for languages supporting data abstraction, where there tend to be many tiny functions implementing operations on an abstract data type. Appel [1992] describes practical heuristics for controlling code explosion.

Continuation-based I/O. Wadler [1995] describes the use of monads to generalize the notion of continuation-based interaction.

Lazy evaluation. Algol-60 [Naur et al. 1963] used call-by-name evaluation for function arguments, implemented using thunks – but also permitted side effects, so programmers needed to know what they were doing! Most of its successors use call-by-value. Henderson and Morris [1976] and Friedman and Wise [1976] independently invented lazy evaluation (call-by-need). Hughes [1989] argues that lazy functional languages permit clearer and more modular programming than imperative languages.

Several lazy pure-functional languages were developed in the 1980s; the community of researchers in this area designed and adopted the language Haskell [Hudak et al. 1992] as a standard. Peyton Jones [1987; 1992] describes many implementation and optimization techniques for lazy functional languages; Peyton Jones and Partain [1993] describe a practical algorithm for higher-order strictness analysis. Wadler [1990] describes deforestation.

PROGRAM COMPILING FUNCTIONAL LANGUAGES

a. Implement Fun-Tiger. A function value can be allocated as a heap-allocated two-element record, containing function-address and static-link fields.

b. Implement PureFun-Tiger. This is just like Fun-Tiger, except that several "impure" features are removed and the predefined functions have different interfaces.

c. Implement optimizations on PureFun-Tiger. This requires changing the `Tree` intermediate language so that it can represent an entire program, including function entry and exit, in a machine-independent way. After inline expansion (and other) optimizations, the program can be converted into the standard `Tree` intermediate representation of Chapter 7.

d. Implement Lazy-Tiger.

EXERCISES

15.1 Draw a picture of the closure data structures representing `add24` and `a` in Program 15.1 just at the point where `add24(a)` is about to be called. Label all the components.

***15.2** Figure 15.13 summarizes the instructions necessary to implement `printTable` in a functional or an imperative style. But it leaves out the MOVE instructions that pass parameters to the calls. Flesh out both the functional and imperative versions with all omitted instructions, writing pseudo-assembly language in the style of the program accompanying Graph 11.1 on page 230. Show which MOVE instructions you expect to be deleted by copy propagation.

***15.3** Explain why there are no cycles in the graph of closures and records of a PureFun-Tiger program. Comment on the applicability of reference-count garbage collection to such a program. **Hint:** Under what circumstances are records or closures updated after they are initialized?

15.4
a. Perform Algorithm 15.9 (loop-preheader transformation) on the `look` function of Program 15.3a.

b. Perform Algorithm 15.10 (loop-invariant hoisting) on the result.

c. Perform Algorithm 15.8 (inline expansion) on the following call to `look` (assuming the previous two transformations have already been applied):

```
look(mytree, a+1)
```

15.5 Perform Algorithm 15.17 (strictness analysis) on the following program, showing the set H on each pass through the **repeat** loop.

```
function f(w: int, x: int, y: int, z: int) =
  if z=0 then w+y else f(x,0,0,z-1) + f(y,y,0,z-1)
```

In which arguments is `f` strict?