Appendix A
A Tutorial on Combinator Graph Reduction

Since part of the audience for this research is the computer architecture community, it seems appropriate to provide a brief introduction to combinator graph reduction for the nonspecialist. This introduction contains many simplifications and omissions, and uses some informal notation and nomenclature in order to convey the essence of how combinator graph reduction works. The objective is not to teach all the details of the process, but rather to give the reader a feel for how a translation from functional programming languages to combinator graphs is performed. More rigorous treatments may be found in Field & Harrison (1988), Henderson (1980), and Peyton Jones(1987).

A.1. FUNCTIONAL PROGRAMS

The basic operation of a functional program is the application of a function to an argument. The notation

f x

means that the function referred to by f is applied to the argument value referred to by x. Functions are so-called "first class citizens" in functional programming languages, which means they may be used as arguments to other functions, and may be returned as the result of functions. For example:

(f g) x

applies function f to function g, then applies the resulting function to argument x. A familiar example of this process is the map function in LISP, which maps a function across an argument which is a list. By convention, function applications associate to the left, so the expression "(f g) x" is equivalent to "f g x", but is not the same as "f (g x)". (This convention is called Curried notation.)

Functional programming languages do not allow side effects, and so lack an assignable state. The equal sign, therefore, has a slightly different meaning than in imperative languages. In functional languages, an assignment statement is used to bind a value to an identifier. For example,

suc(x) = x + 1

defines a function called suc which returns the value of its input parameter plus 1. Conditional assignments may be handled by using a piecewise definition of the function. For example, the signum function may be defined as:

signum(x) = -1 ; x < 1
= 0 ; x = 0
= 1 ; x > 1

We shall use a function that doubles its input as an example program:

double(x) = x + x

This running example will illustrate the process of converting a function definition to a combinator graph, then executing that combinator graph.

A.2. MAPPING FUNCTIONAL PROGRAMS TO LAMBDA CALCULUS

The first step in compiling a functional program to a combinator graph is to transform it into an expression in the lambda calculus (Barendregt 1981). The lambda calculus is a simple but computationally complete program representation that involves applying functions to input parameters. Lambda expressions have the form:

λ name . body

The "λ" indicates a function abstraction. The "name" indicates the local name of the input variable. The period separates the header from the function definitions. The "body" is the body of the lambda expression in prefix polish notation. A lambda expression that just returns its input is:

λ x . x

A lambda expression which adds three to its input is:

λ x . + x 3

The lambda expression for a doubling function, which will be a running example throughout this tutorial, is:

λ x . + x x

A function which takes two inputs and multiplies them together is:

λ y . λ x . * x y

Note that two λs are needed to specify this function, since only one input parameter may be associated with each λ. What is actually happening is that the function "multiply the input by x" (which corresponds to the expression λ x . * x) is being applied to the argument y to form a new function "multiply y by x".

A.3. MAPPING LAMBDA CALCULUS TO SK-COMBINATORS

SK-combinators are a small set of simple transformation rules for functions. Each combinator has associated with it a runtime action and a definition. The definition is used to translate programs from the lambda calculus into combinator form. The runtime action is used to perform reductions to evaluate expressions.

I is the identity function. At the function application level of abstraction, the action taken by applying **I** to an input x is:

I x → x

Which is to say, the function **I** applied to any argument x simply returns the result x. The lambda calculus equivalent of **I** is simply:

I is defined as λ x . x

K is a cancellator function (**K** and **S** come from the German heritage of Schoenfinkel, the inventor of combinators (Belinfante 1987)) that drops the second argument and returns the first argument:

K c x → c

The first argument c (which must be a constant expression) is returned, while the second argument x is dropped. The lambda calculus implementation of **K** is:

K c is defined as λ x . c for c a constant

S is the distributor function, which distributes an argument to two different functions:

S f g x → f x (g x)

Remember that function applications associate to the right, so a fully parenthesized version of this transformation is:

(((S f) g) x) → ((f x) (g x))

The action of the **S** combinator is to distribute the argument x to two functions f and g, then apply f(x) to g(x). The lambda expression for **S** is:

S (λx .a) (λx .b) is defined as λ x . a b where a and b are lambda expressions.

Amazingly, **S**, **K**, and **I** are sufficient to represent any computable function! An implication of this is that variables are unnecessary for programming, since converting programs to combinator form eliminates all references to variables. As a further simplification, even **I** is not needed, since it is equivalent to the sequence **S K K**, hence the term SK-combinator instead of SKI-combinator. In practice, however, **I** is so useful that it is always included as a basic combinator.

In order to convert a lambda expression to sequence of SK-combinators, we repeatedly apply the lambda definitions of the combinators "in reverse", picking the innermost lambda expressions first. For our example program, we would start with:

(λ x . + x x)

First, let us add more parentheses to emphasize the evaluation order:

(λx .((+ x) x))

In the following sequence, the underlined term represents the next term which will be transformed.

(λx .((+ x) x))

Applying the **S** rule to the λ expression, the **a** term is **(+ x)** and the **b** term is **x**, resulting in:

((S (λx .(+ x))) (λx .x))

Now, we can apply the **I** rule to λx .x, giving:

((S (λx .(+ x))) I)

Continuing,

((S ((S (λx .+)) (λx .x))) I)

((S ((S (λx .+)) I)) I)

Note that the **+** operator is a constant expression, since functions are no different than data elements in this notation. Therefore, we apply a **K** reduction to the expression (λx .+), getting our result.

((S ((S (K +)) I)) I)

The result is a function which takes one argument. The transformation process has been rather mechanical, so it is not at all obvious

that this is the correct answer. To reassure ourselves, let us apply this function to an input, say 123, and use the combinator reduction rules to reduce the expression to a result. Note that we are using the result of a function as the input to the expression. We shall use the same underlining convention to highlight the combinator expression which will be converted using the combinator execution rules. Control is always passed to the leftmost combinator. This passing of control to the leftmost combinator is called normal order reduction, and is an implementation strategy that guarantees lazy evaluation.

((S ((S (K +)) I)) I) 123

For this reduction, **S** takes two functions f and g, and applies the rule:

((S f) g) x → (f x) (g x)

In this case, function **f** is ((**S** (**K** +)) I) and function **g** is just **I**, while the argument **x** is 123.

(((S (K +)) I) 123) 123

Notice that **S** has made a copy of the input parameter 123. Next, another **S** is executed.

(((K +) 123) (I 123)) 123

(+ (I 123)) 123

Note that + is a strict operator. It forces evaluation of both its arguments before it returns a result. Therefore, the next combinator reduced is the **I** and not the **+**.

(+ 123) 123

246

And, we have our result. No understanding of how the combinators are working has been required: blind adherence to the reduction rules guarantees correct operation, and in fact, guarantees complete laziness when performing normal order reduction. The process of performing the reduction is tedious, but follows very simple rules.

A.4. MAPPING SK-COMBINATOR EXPRESSIONS ONTO A GRAPH

There is one step left. How does one map the combinators into a data structure for computation? The data structure of choice is a binary tree (with cycles and sharing permitted to accomplish recursion and common subexpression sharing, respectively). Each node of the tree, shown in Figure A-1, has a left-hand side, which is the function cell, and a

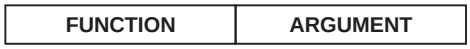

Figure A-1. The function and argument structure of a node.

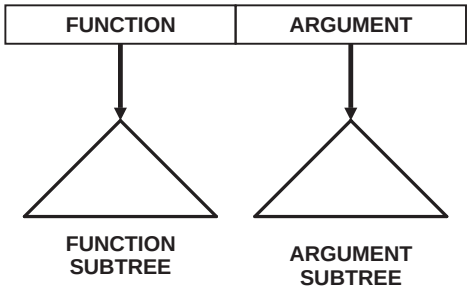

Figure A-2. A function argument pair.

right-hand side, which is the argument cell. The function may be either a combinator or a pointer to a subtree. The argument may be a combinator, a pointer to a subtree, or a constant value. In general, each argument is a subtree, so we shall draw it as shown in Figure A-2, with the understanding that a degenerate subtree case is simply a combinator or constant value.

Since functions are first class citizens, functions and arguments are not distinguishable except by the fact that they are pointed to by a left-hand side or right-hand side. In fact, since sharing of subtrees is permitted, the same subtree may be both a function and an argument for different parent nodes simultaneously, as shown in Figure A-3.

The operation of combinators may be translated directly from parenthesized form to a graphical representation. Each pair of parentheses encloses a list of exactly two objects, which form the function and argument halves of a node. Figure A-4 shows a graph which adds the numbers 11 and 22. In this diagram, the plus operation is paired with 11, and the function (+ 11) is paired with 22 as its second operand. Note that a function is at the leftmost leaf of the subtree, and that the ***n*** arguments for the function are in the ***n*** **right-hand sides of the current and ancestor nodes. This formatting of a function and its arguments is universal among combinators.**

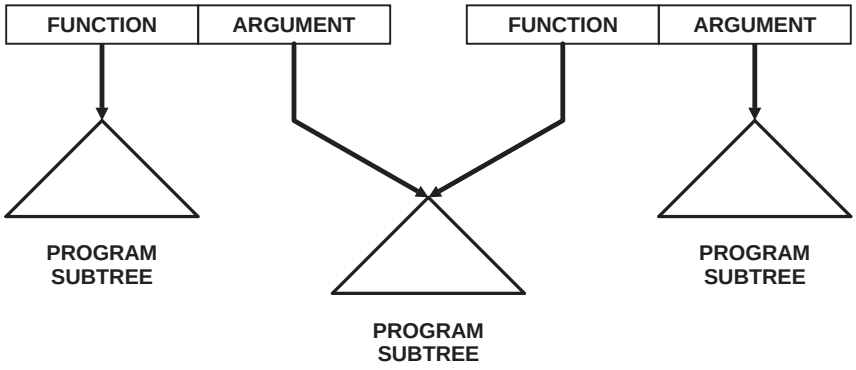

Figure A-3. A shared subtree.

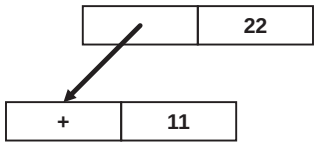

Figure A-4. Graph to add 11 and 22.

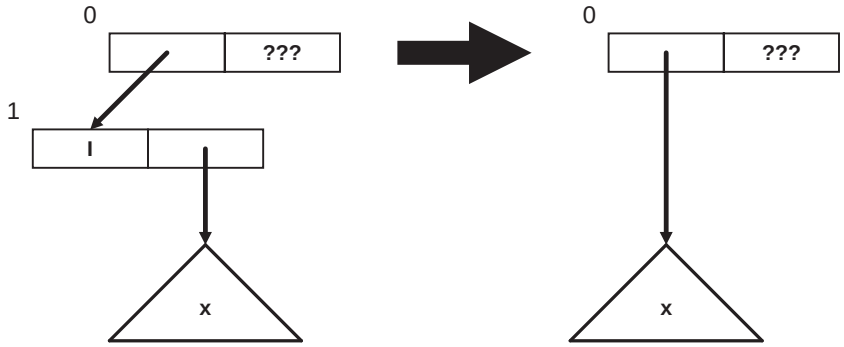

Figure A-5. Operation of the **I** combinator. (I x) → x

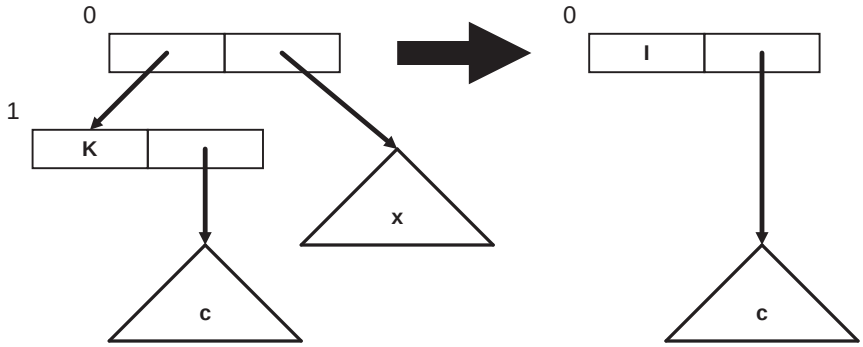

Figure A-6. Operation of the **K** combinator. ((K c) x) → (I x)

The reduction rules for the combinators **S**, **K**, **I**, and **+** are shown in Figures A.5 through A.8. Figure A-5 shows the reduction rule for **I**. Since **I** is the identity function, it simply repoints its parent (node 0) to the subtree X. The right-hand side value of node 0 is unimportant, and does not participate in the reduction.

Figure A-6 shows the reduction rule for the **K** combinator. **K** drops the reference to x from node 0 and replaces it with a reference to c. Since a reference to node c only occupies half a node, an I combinator is placed in the function side of node 0 to keep everything operating properly.

Figure A-7 shows the reduction rule for the **S** combinator. **S** allocates new nodes 3 and 4 from the heap (the data structure used for dynamically allocating memory elements), then repoints node 0 to these new nodes. This new node allocation has very important implications. At the simplest level, it ensures that if nodes 1 or 2 are shared with other nodes, proper operation will result. At a deeper lever, it is used to build new suspension structures to allow recursion. The actual mechanism of the recursion technique is beyond the scope of this brief tutorial.

Figure A-8 shows how the **+** combinator is reduced. Since **+** is strict in both its parameters, subtrees A and B are first reduced to constant values. Then, the sum is computed and placed in node 0 with an indirection node. The result of the A subtree computation is left behind in node 1, so that any node sharing node 1 will not have to reevaluate

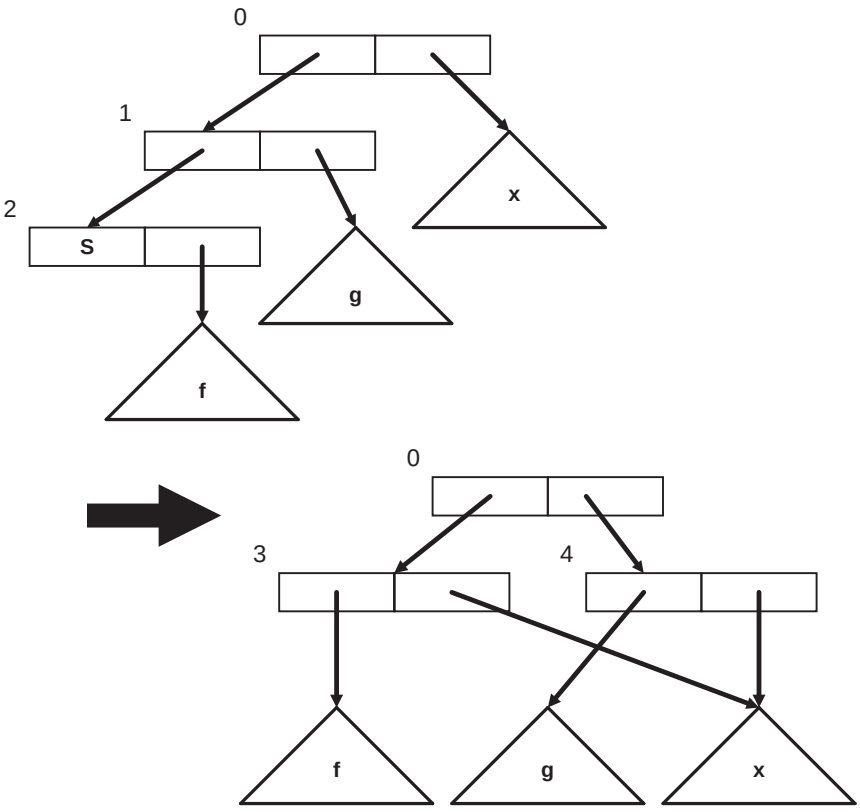

Figure A-7. Operation of the **S** combinator. $(((S\ f)\ g)\ x) \rightarrow (f\ x)\ (g\ x)$

subtree A. The updating of nodes 0 and 1 provide automatic common subexpression sharing at runtime.

Let us now examine how our example program is translated into a combinator graph. Figure A-9 shows the combinator graph for the doubling function. Since a function must be applied to an argument to yield a result, we shall construct a graph with the argument ((+ 100) 23) as shown in Figure A-10. A subtree that computes the sum will be used instead of just a constant to illustrate how sharing is used to eliminate redundant computations. Observe how a parent node has been added above the double function, representing the application of double to the

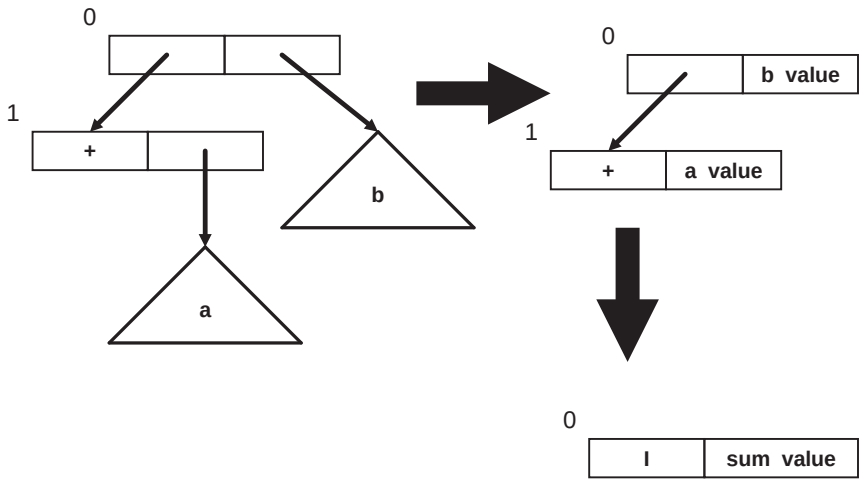

Figure A-8. An addition example. ((+ a) b) → (I sum)

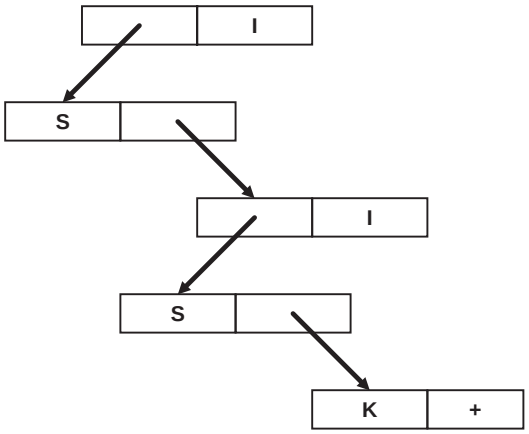

Figure A-9. Doubling function. ((S ((S (K +)) I)) I)

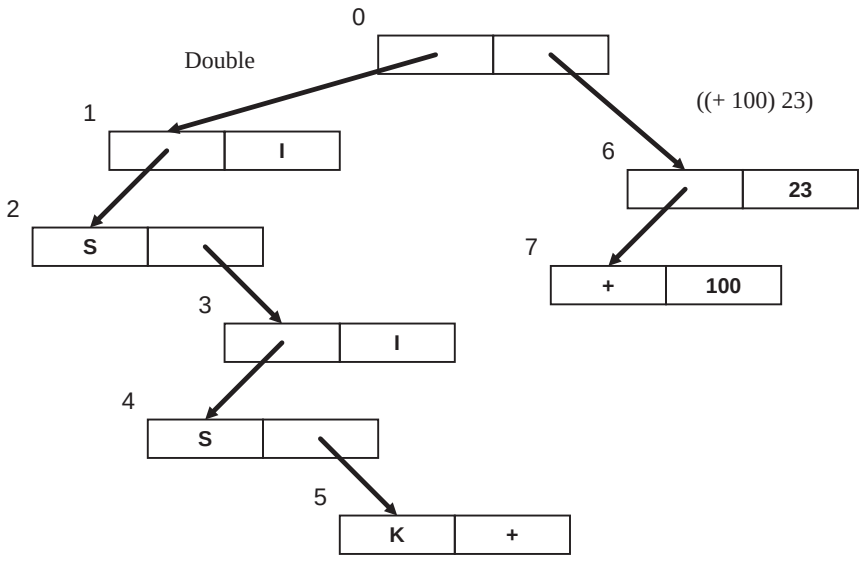

Figure A-10. Doubling function applied to argument. ((+ 100) 23).

input parameter ((+ 100) 23). The nodes have been assigned numeric labels so that we may conveniently refer to them.

For normal order reduction, the left-hand side pointer of each node is followed until a combinator leaf is reached. For this example, this means that the graph reducer would follow the pointer chain through the left-hand side of nodes 0, 1, and 2. Figure A-11 shows the results of this traversal.

Since we are performing a tree traversal, a stack of nodes visited is kept so that we may retrace our steps. Since the leftmost series of pointers we have followed is called the left spine of the graph, the stack is called the spine stack. In these diagrams, the stack grows downward as the tree is traversed from top to bottom.

Once the **S** combinator in Figure A-11 is found, an **S** graph reduction is performed. The **S** combinator knows which nodes are its parents by the contents of the spine stack. Since the **S** combinator uses three nodes as its input, the nodes participating in the reduction are nodes 2, 1, and 0 as indicated by the shaded area and the top three elements on the spine stack. Figure A-12 shows the graph after the **S**

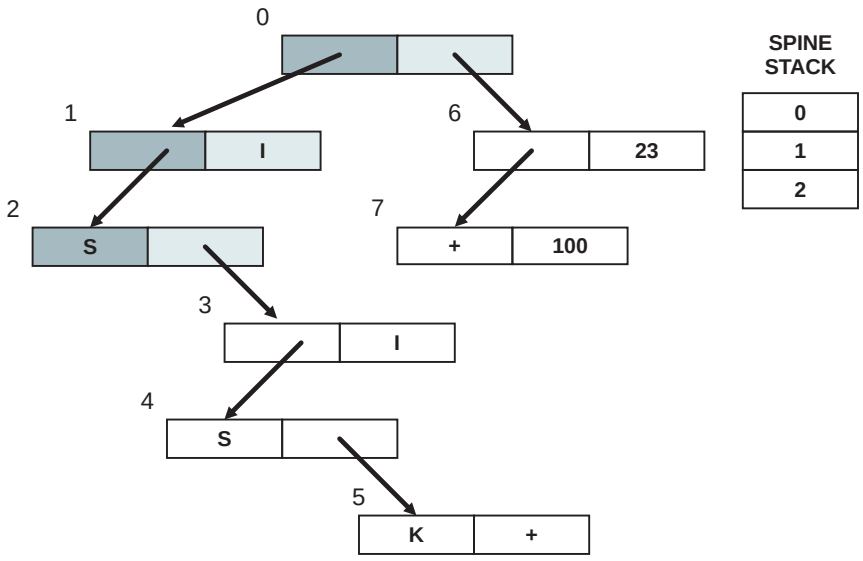

Figure A-11. Reduction step 1.

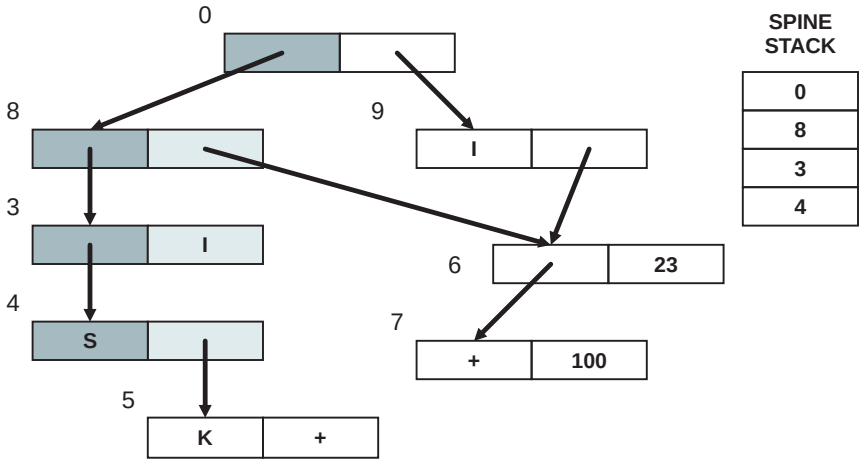

Figure A-12. Reduction step 2.

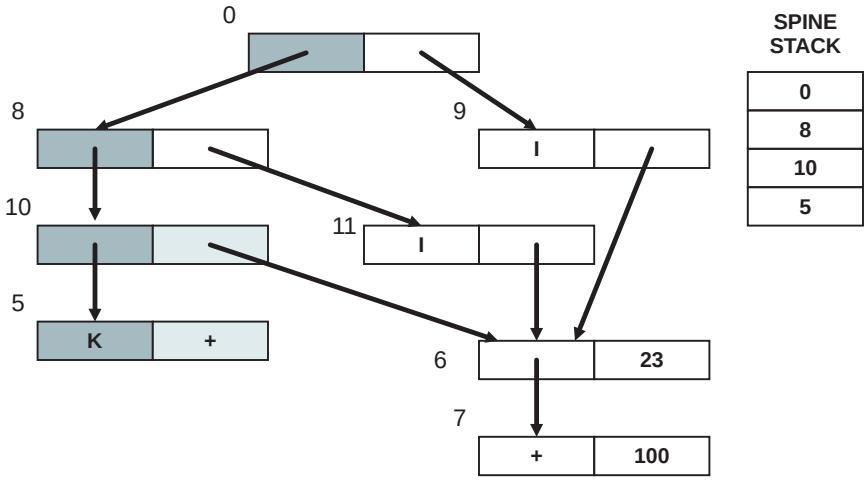

Figure A-13. Reduction step 3.

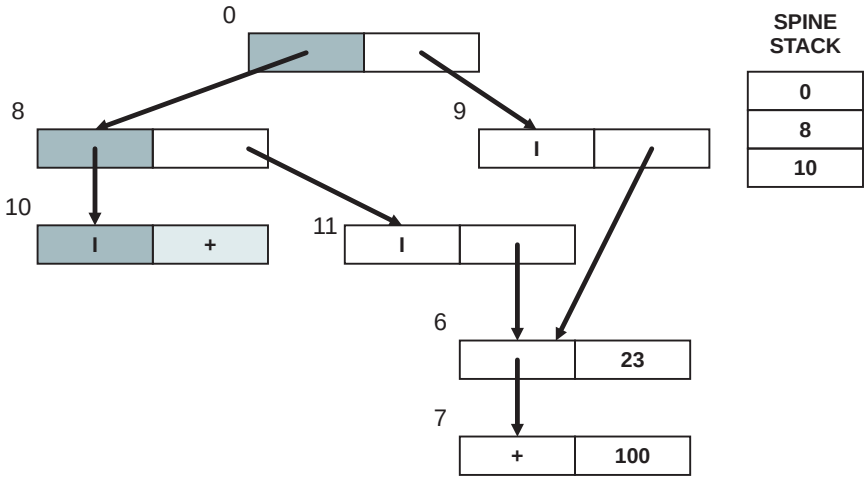

Figure A-14. Reduction step 4.

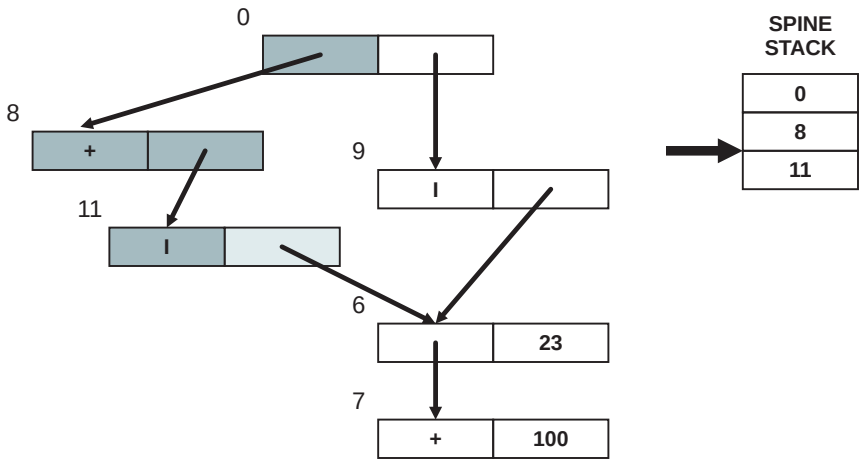

Figure A-15. Reduction step 5.

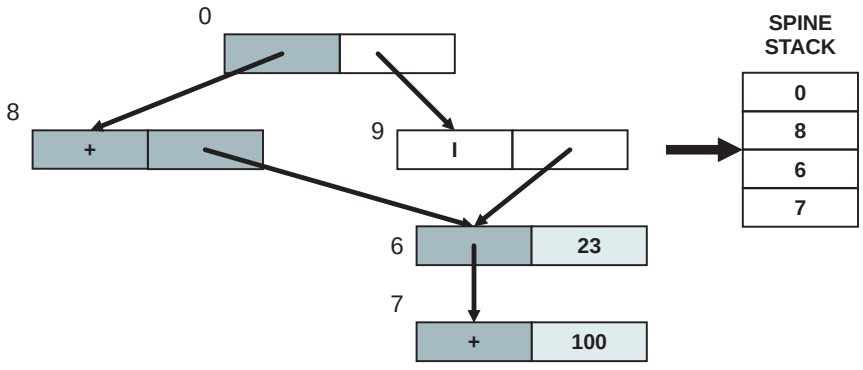

Figure A-16. Reduction step 6.

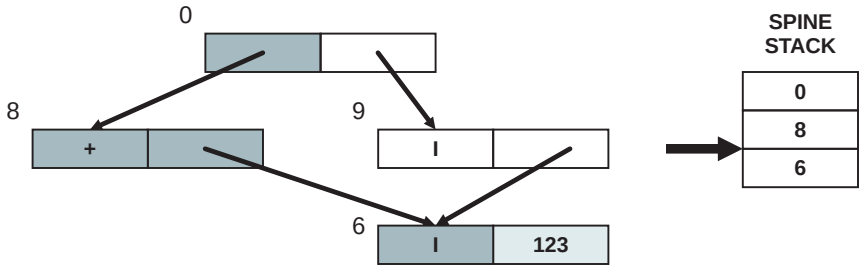

Figure A-17. Reduction step 7.

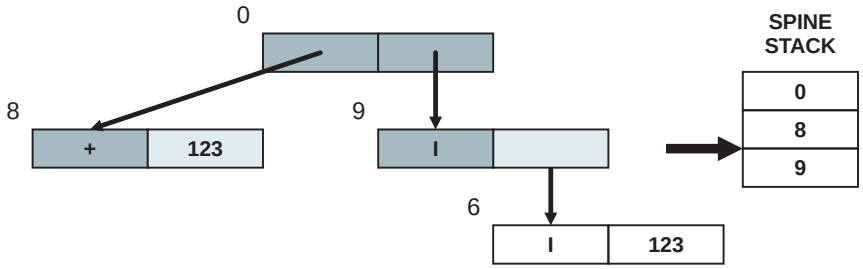

Figure A-18. Reduction step 8.

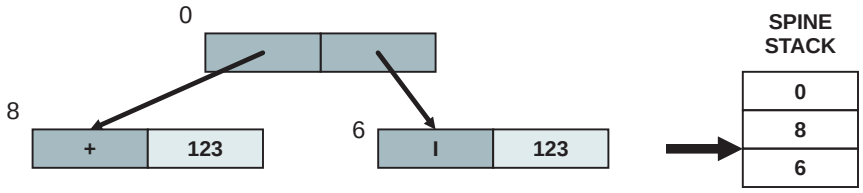

Figure A-19. Reduction step 9.

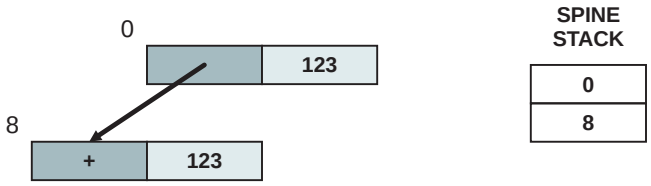

Figure A-20. Reduction step 10.

Figure A-21. Reduction step 11.

reduction. Two new nodes, 8 and 9, have been allocated from the heap. Nodes 1 and 2 have been abandoned and become garbage cells in the heap.

After the **S** combinator from Figure A-11 has been executed to yield the graph shown in Figure A-12, traversal down the left spine continues until the **S** combinator at node 4 is reached. In this case, nodes 4, 3, and 8 are used as inputs for the **S** combinator, resulting in the graph shown in Figure A-13.

Execution continues with reduction of the **K** combinator and nodes 5 and 10 from Figure A-13 to modify node 10 into an **I** node as shown in Figure A-14. This graph is further reduced to that shown in Figure A-15.

In Figure A-15, we have a slight problem. The combinator to be reduced is **+**, but that combinator is strict in both arguments (i.e. it requires both arguments to be evaluated before the addition may be performed), and the arguments are subtrees instead of constants. To solve this problem, we temporarily suspend evaluation of the **+** combinator, and recurse with our evaluation process to evaluate the subtree pointed to by the right-hand side of node 8. The marker in the spine stack shows that we may use the same spine stack for this evaluation, but must remember to return control to the **+** combinator when the marker is on top of the spine stack. Figures A.15, A.16, A.17, and A.18

show the evaluation of the first argument of the + combinator in node 8. Note that in Figure A-17 the input subexpression ((+ 100) 23) has been reduced to a single value, which is shared by both nodes 8 and nodes 9.

Once the first argument for the addition is evaluated, the process is repeated for the second argument in the right-hand side of node 0, as illustrated by Figures A.18, A.19, and A.20.

Finally, in Figure A-20, the addition is ready to be performed, with the result placed in node 0. When node 0 is evaluated, it produces the correct result of 246 shown in Figure A-21.

A.5. THE TURNER SET OF COMBINATORS

There are some problems with the SKI set of combinators. While they are sufficient to do any job, they have certain inherent inefficiencies. The most obvious inefficiency is that the **S** combinator is forced to pass copies of its third argument to both the right and the left, when what is often desired is passing the argument to either the left or right, but not both.

In the graph of Figure A-13, node 5 contained a **K** node whose purpose was to discard the copy of a pointer to node 6 which was in the right-hand side of node 10. In other words, the creation of node 10 was a waste of effort, since its effects were undone by the reductions in Figures A.14 and A.15. Node 10 was created because node 11 was needed to make a copy of the input for the addition, and **S** must create two nodes at a time.

Combinator input	Result
I x	x
K c x	(I c)
S f g x	((f x) (g x))
B f g x	(f (g x))
C f g x	((f x) g)
S' c f g x	((c (f x)) (g x))
B* c f g x	(c (f (g x)))
C' c f g x	((c (f x)) g)
U f P x y	((f x) y)

Table A-1. Non-strict members of the Turner combinator set.

The Turner Set contains combinators that pass input parameters to only the left and right sides, and perform other useful graph rewrites of a similar nature. The full Turner Set is shown in Table A-1 (Peyton Jones 1987).

Figure A-22 shows the effects of reducing a subgraph with the **B** combinator. The results are similar to an **S** combinator reduction, but the third argument x is passed only to g, not to f. The **C** combinator, shown in Figure A-23, is a converse operation, which only passes x to f but not g. The addition of the **B** and **C** combinators can cut garbage production nearly in half, and eliminate a large number of **K** and **I**

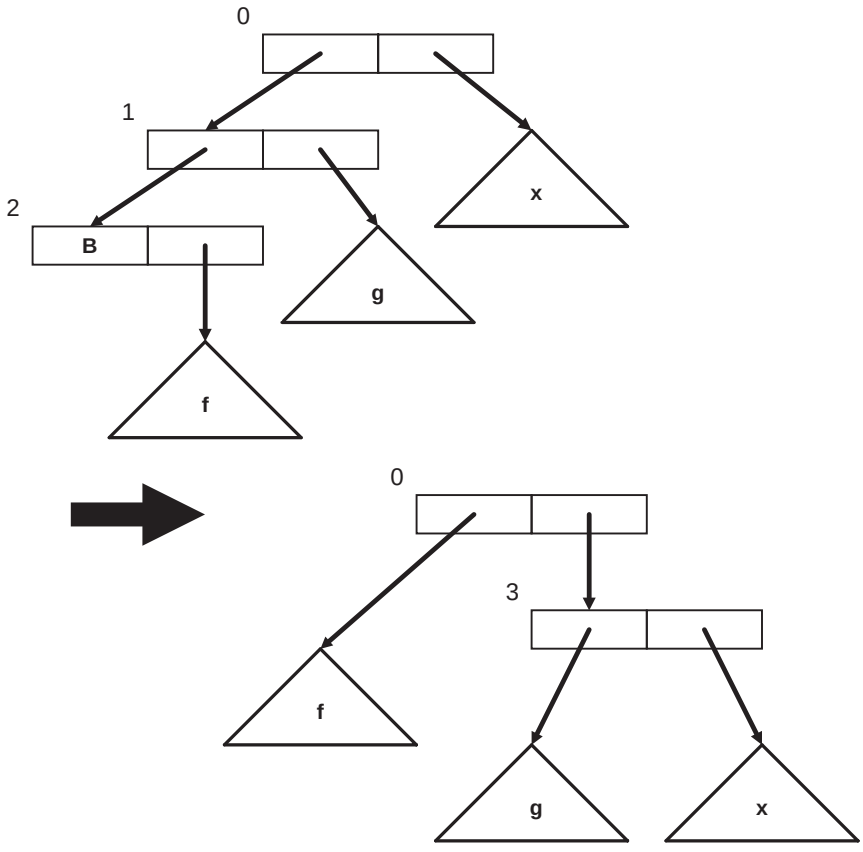

Figure A-22. Operation of the **B** combinator. $(((B\ f)\ g)\ x) \rightarrow (f\ (g\ x))$

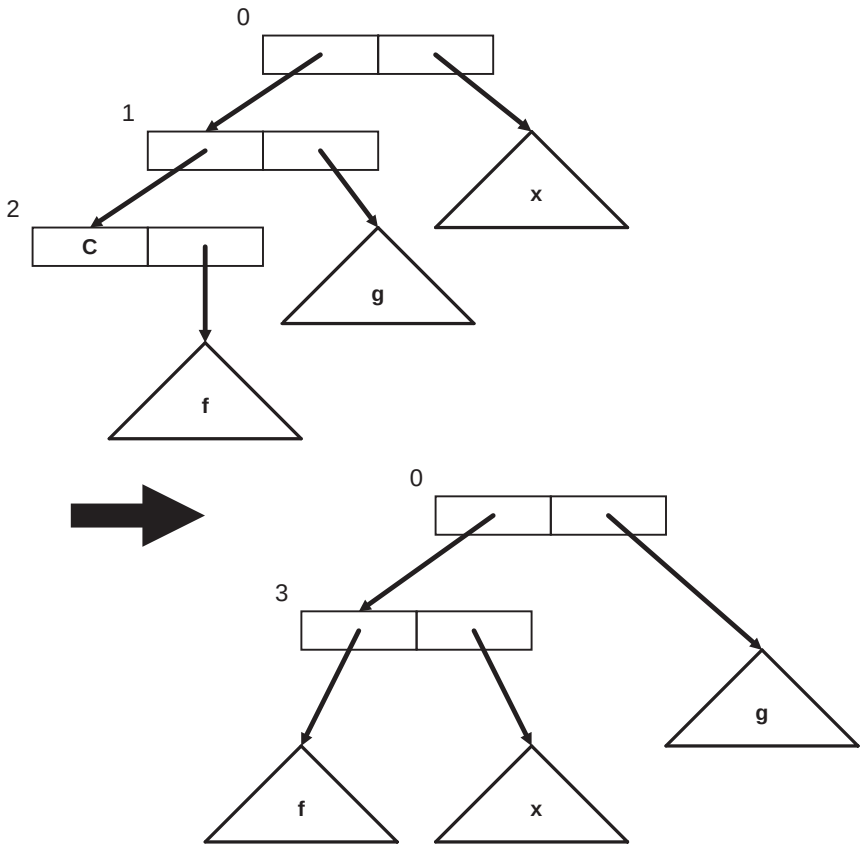

Figure A-23. Operation of the **C** combinator. $(((C\ f)\ g)\ x) \rightarrow ((f\ x)g\ )$

Starting expression		Optimized
1. ((S (K p))(K q))	→	(K (p q))
2. ((S (K p)) I)	→	p
3. ((S (K p))(B q r))	→	(((B* p) q) r)
4. ((S (K p)) q)	→	((B p)q)
5. ((S ((B p) q))(K r)	→	(((C' p)q)r)
6. (S (p (K q)))	→	((C p) q)
7. ((S (B p) q)) r)	→	(((S' p)q)r)

Table A-2. Turner Set optimizations.

reductions that would be needed to dispose of unwanted copies of pointers to parameters.

Using the full Turner Set of combinators can be accomplished by first compiling to SKI combinators, then applying the optimization rules shown in Table A-2 (Peyton Jones 1987). At each step, the smallest numbered optimization rule that can be applied is used to ensure the highest quality code. The first two optimizations do not actually involve the introduction of new combinators, but rather make use of mathematical identities. For our example of the doubling function, these optimizations would result in the modification:

```
((S ((S (K +)) I) ) I)
        ((S + ) I)
```

by applying rule 2, which is certainly a big simplification! In fact, in this case, **B** and **C** were not needed at all. However, an inferior quality optimization could be done using rule 4, which would produce:

```
((S ((S (K +)) I) ) I)
        ((S ((B +) I) I)
```

illustrating the use of **B** to eliminate the creation of node 10 as discussed earlier.

The astute reader will notice that all these optimizations begin with the **S** combinator, and that a peephole optimizer is well suited to performing the optimizations on-the-fly during code generation. The power of using these extra combinators should not be underestimated. For example, the doubly recursive Fibonacci function can be simplified from:

```
((S ((S ((S (K IF)) ((S ((S (K )) I)) (K 3)))) (K 1)))
        ((S ((S (K +)) ((S (K CYCLE)) ((S ((S (K -)) I)) (K 1)))))
        ((S (K CYCLE)) ((S ((S (K -)) I)) (K 2)))))
```

to:

```
((S (((S' IF) ((C ) 3)) (K 1))) (((S' +) ((B CYCLE)
        ((C -)1))) ((B CYCLE) ((C -) 2))))
```

by using the full Turner Set. The entity **CYCLE** is not a combinator, but rather a compiler directive that compiles a pointer to the root of the graph for recursion.

A.6. SUPERCOMBINATORS

One method for creating large combinators is using supercombinator compilation (Hughes 1982). The idea is that there is unnecessary overhead associated with invoking combinators and allocating heap cells which are immediately discarded. To reduce these overheads, supercombinators can be created by compressing suitable strings of combinators into special-purpose code sequences to rewrite large subgraphs. For example, the optimized expression for doubling a number is:

((S +) I)

This expression can be made into a combinator itself by creating a new function that takes a single input and produces an output which is doubled. Let us call this new combinator **$DOUBLE**. Using **$DOUBLE**, the example of doubling the sum of 100 and 23 can be written as:

($DOUBLE ((+ 100) 23))

instead of:

(((S +) I) ((+ 100) 23))

Since a function that doubles its input is quite likely to be faster than a sequence generated with **S**, **I**, and **+** combinators, the supercombinator version using **$DOUBLE** will run faster. In fact, the supercombinator technique can be extended to the point of creating a unique set of supercombinators for each program with only a little SKI glue used to hold the program together. The supercombinators must be created by the compiler for a specific program, since the number of possible supercombinators is unbounded.

The advantages of using supercombinators are clear. Supercombinators can reduce the manipulations of the graph as well as the graph size by providing customized combinator functions. Improvements of a factor of ten in execution speed are believed possible (Wray 1988).

From a hardware construction viewpoint, special-purpose graph reduction systems are often limited to the SKI or Turner Set of combinators because there is limited microcode memory available to support the combinator instruction set. With the TIGRE architecture introduced by this book, this is not a problem, since TIGRE uses a small set of (potentially microcoded) primitive operations described by TIGRE assembly language to synthesize combinators which are defined as routines in program memory. Other abstract machines, notably the G-Machine and TIM discussed in Chapter 2, also synthesize combinators from a small set of primitive functions.

A.7. INHERENT PARALLELISM IN COMBINATOR GRAPHS

One of the things that makes functional programming languages exciting is the lure of "free" parallelism. Since functional programming languages are referentially transparent, the evaluation order is unimportant (except when dealing with infinite-length data structures, but with reasonable care that situation can be handled).

Because of this insensitivity to evaluation order, the full available parallelism of a program may be exploited simply by spinning off a new process down the right-hand side of each tree node while traversing the leftmost nodes. Issues of controlling parallelism and allocating resources to ensure best use of the hardware exist, but the point is that the combinator graphs are programs that have implicit parallelism information inherent in their structure.

Much work needs to be done in the area of parallelism, but a good start will be to create a very efficient graph reduction engine, and to identify the architectural features necessary to support fast uniprocessor graph reduction.