THEORETICAL PEARLS

Flattening Combinators: Surviving Without Parentheses

CHRIS OKASAKI
United States Military Academy∗
West Point, New York
(*e-mail:* `Christopher.Okasaki@usma.edu`)

Abstract

A combinator expression is *flat* if it can be written without parentheses, that is, if all applications nest to the left, never to the right. This note explores a simple method for flattening combinator expressions involving arbitrary combinators.

You should avoid using too many parenthetical remarks...
— Lyn Dupré, *BUGS in Writing*

1 Introduction

I recently received the following email:

> *Sir,*
>
> *I just spilled soda on my keyboard and now the 9 and 0 keys don't work. I can still use the 9 and 0 on the numeric keypad, but what should I do about the parentheses? I can pick up a new keyboard over the weekend, but the combinator homework is due tomorrow, and it uses parentheses all over the place!*
>
> *Respectfully,*
> *Cadet K*

Many possible solutions spring to mind—use brackets instead of parentheses, remap the keyboard, download a hex editor—but let us take Cadet K's problem at face value. Because application is typically treated as left associative, combinator expressions need parentheses only when an application occurs in the argument position of another application. Suppose we forbid such applications and require all expressions to be written in the form $c_1 \ldots c_n$. Call such an expression *flat*. Can an arbitrary combinator expression always be rewritten in this form? Yes.

∗ The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Military Academy, the Department of the Army, the Department of Defense, or the U.S. Government.

2 A Simple Transformation

Postfix notations such as RPN are famous for avoiding parentheses. For example, the arithmetic expression $(1+2)*(3+4)$ is rewritten in RPN as

```
1 2 + 3 4 + *
```

Postfix languages are nearly always based on a stack model. In RPN, numbers mean "push this number onto the stack" and binary operators mean "pop the top two numbers from the stack, perform the operation, and push the result". Perhaps we can use the same approach for combinator expressions, where combinators replace numbers and applications replace arithmetic operators.

We introduce four primitive commands:

- $\mathsf{Push}[c]$: Push the combinator c onto the stack.
- Apply: Pop the top two values from the stack, apply one to the other, and push the result.
- Begin: Initialize the stack to empty. *Used only once, at the beginning of the program.*
- End: Pop the final answer off the stack. *Used only once, at the end of the program.*

Begin and End are not strictly necessary, but will be convenient later. Programs are sequences of commands separated by semicolons.

In this notation, the expression $(c_1c_2)(c_3c_4)$ can be rewritten

$$\mathsf{Begin}; \mathsf{Push}[c_1]; \mathsf{Push}[c_2]; \mathsf{Apply}; \mathsf{Push}[c_3]; \mathsf{Push}[c_4]; \mathsf{Apply}; \mathsf{Apply}; \mathsf{End}$$

Execution of this program would proceed as follows (the stack grows to the left):

$$\begin{array}{ll}
\mathsf{Begin}; & // \text{ stack} = [\,] \\
\mathsf{Push}[c_1]; & // \text{ stack} = [c_1] \\
\mathsf{Push}[c_2]; & // \text{ stack} = [c_2, c_1] \\
\mathsf{Apply}; & // \text{ stack} = [(c_1c_2)] \\
\mathsf{Push}[c_3]; & // \text{ stack} = [c_3, (c_1c_2)] \\
\mathsf{Push}[c_4]; & // \text{ stack} = [c_4, c_3, (c_1c_2)] \\
\mathsf{Apply}; & // \text{ stack} = [(c_3c_4), (c_1c_1)] \\
\mathsf{Apply}; & // \text{ stack} = [((c_1c_2)(c_3c_4))] \\
\mathsf{End} & // \text{ final answer} = (c_1c_2)(c_3c_4)
\end{array}$$

The orginal expression can be compiled into this code via a simple post-order traversal of the expression tree. Let $\lceil\cdot\rceil$ be the compilation function.

$$\lceil e \rceil = \mathsf{Begin}; \lfloor e \rfloor ; \mathsf{End}$$

Most of the work is done by the auxiliary compilation function $\lfloor\cdot\rfloor$.

$$\begin{array}{lcl}
\lfloor c \rfloor & = & \mathsf{Push}[c] \\
\lfloor e_1 e_2 \rfloor & = & \lfloor e_1 \rfloor ; \lfloor e_2 \rfloor ; \mathsf{Apply}
\end{array}$$

3 From Postfix Commands to Combinators

Next, we want to turn the four postfix commands into combinators. How we do this depends on how we interpret semicolons.

3.1 Left-Associative Reverse Function Application

Eventually, we want to treat semicolons as left-associative function application, but we begin by interpreting semicolons as left-associative reverse function application (i.e., $x; f = fx$ and $x; f; g = (x; f); g = g(fx)$).

Under this interpretation, the postfix program

$$\mathsf{Begin}; \mathsf{Push}[c_1]; \mathsf{Push}[c_2]; \mathsf{Apply}; \mathsf{End}$$

becomes

$$\mathsf{End}\ (\mathsf{Apply}\ (\mathsf{Push}[c_2]\ (\mathsf{Push}[c_1]\ \mathsf{Begin})))$$

This expression makes perfect sense if we interpret Begin as a stack, $\mathsf{Push}[c]$ and Apply as functions from stacks to stacks, and End as a function from stacks to values. We implement stacks as nested pairs, using the usual Church definitions of pairs:

$$\begin{array}{lcl} \mathsf{Pair} & = & \lambda x.\lambda y.(\lambda f.f\ x\ y) \\ \mathsf{Pcase} & = & \lambda p.\lambda f.p\ f \end{array}$$

The Pcase combinator takes a pair and a function, and passes both elements of the pair to the function. We also need a value for the empty stack, although the details of this definition are irrelevant because the empty stack will never be inspected.

$$\mathsf{Empty} = \lambda x.x$$

The definitions of the four commands are then

$$\begin{array}{lcl} \mathsf{Push}[c] & = & \lambda s.\mathsf{Pair}\ c\ s \\ \mathsf{Apply} & = & \lambda s.\mathsf{Pcase}\ s\ (\lambda x.\lambda s'.\mathsf{Pcase}\ s'\ (\lambda y.\lambda s''.\mathsf{Pair}\ (y\ x)\ s'')) \\ \mathsf{Begin} & = & \mathsf{Empty} \\ \mathsf{End} & = & \lambda s.\mathsf{Pcase}\ s\ (\lambda x.\lambda s'.x) \end{array}$$

If we allow pattern matching on the Pair constructor as syntactic sugar for Pcase, these definitions can be written more clearly as

$$\begin{array}{lcl} \mathsf{Push}[c]\ s & = & \mathsf{Pair}\ c\ s \\ \mathsf{Apply}\ (\mathsf{Pair}\ x\ (\mathsf{Pair}\ y\ s)) & = & \mathsf{Pair}\ (y\ x)\ s \\ \mathsf{Begin} & = & \mathsf{Empty} \\ \mathsf{End}\ (\mathsf{Pair}\ x\ s) & = & x \end{array}$$

3.2 Right-Associative Function Application

We proceed by reinterpreting semicolons as right-associative function application rather than left-associative reverse function application. Under this interpretation, the postfix program

$$\mathsf{Begin}; \mathsf{Push}[c_1]; \mathsf{Push}[c_2]; \mathsf{Apply}; \mathsf{End}$$

becomes

$$\mathsf{Begin}\ (\mathsf{Push}[c_1]\ (\mathsf{Push}[c_2]\ (\mathsf{Apply}\ \mathsf{End})))$$

Each command now takes all subsequent commands as an argument, rather than all previous commands. This suggests a continuation-passing approach. Take continuations to be functions from stacks to the final answer. Then End is a continuation, Push$[c]$ and Apply are functions from continuations to continuations, and Begin is a function that takes a continuation and produces the final answer. The definitions do not change much, except for the introduction of continuations:

$$\begin{array}{lcl}
\mathsf{Push}[c]\ k\ s & = & k\ (\mathsf{Pair}\ c\ s) \\
\mathsf{Apply}\ k\ (\mathsf{Pair}\ x\ (\mathsf{Pair}\ y\ s)) & = & k\ (\mathsf{Pair}\ (y\ x)\ s) \\
\mathsf{Begin}\ k & = & k\ \mathsf{Empty} \\
\mathsf{End}\ (\mathsf{Pair}\ x\ s) & = & x
\end{array}$$

3.3 Left-Associative Function Application

Finally, we reinterpret semicolons as left-associative function application. The postfix program

$$\mathsf{Begin}; \mathsf{Push}[c_1]; \mathsf{Push}[c_2]; \mathsf{Apply}; \mathsf{End}$$

becomes

$$\mathsf{Begin}\ \mathsf{Push}[c_1]\ \mathsf{Push}[c_2]\ \mathsf{Apply}\ \mathsf{End}$$

No parentheses, at last!

The necessary changes to the combinators are surprisingly minor—merely reverse the arguments to Push$[c]$ and Apply. Begin and End are unchanged.

$$\begin{array}{lcl}
\mathsf{Push}[c]\ s\ k & = & k\ (\mathsf{Pair}\ c\ s) \\
\mathsf{Apply}\ (\mathsf{Pair}\ x\ (\mathsf{Pair}\ y\ s))\ k & = & k\ (\mathsf{Pair}\ (y\ x)\ s) \\
\mathsf{Begin}\ k & = & k\ \mathsf{Empty} \\
\mathsf{End}\ (\mathsf{Pair}\ x\ s) & = & x
\end{array}$$

In changing the order of arguments, we also change how we use continuations. Rather than encapsulating the rest of the program, a continuation now represents only the very next command. It is helpful to consider an example reduction in which I have underlined the redex at each step.

$$\begin{array}{ll}
 & \underline{\mathsf{Begin}\ \mathsf{Push}[c_1]}\ \mathsf{Push}[c_2]\ \mathsf{Apply}\ \mathsf{End} \\
\Rightarrow & \underline{\mathsf{Push}[c_1]\ \mathsf{Empty}\ \mathsf{Push}[c_2]}\ \mathsf{Apply}\ \mathsf{End} \\
\Rightarrow & \underline{\mathsf{Push}[c_2]\ (\mathsf{Pair}\ c_1\ \mathsf{Empty})\ \mathsf{Apply}}\ \mathsf{End} \\
\Rightarrow & \underline{\mathsf{Apply}\ (\mathsf{Pair}\ c_2\ (\mathsf{Pair}\ c_1\ \mathsf{Empty}))\ \mathsf{End}} \\
\Rightarrow & \underline{\mathsf{End}\ (\mathsf{Pair}\ (c_1\ c_2)\ \mathsf{Empty})} \\
\Rightarrow & c_1\ c_2
\end{array}$$

We now have all the machinery necessary to convert an arbitrary combinator expression into flat form. The resulting flat expression has size $2n + 1$ (where n is the size of the original expression, not counting applications) and uses only $k + 3$ distinct combinators (where k is the number of distinct combinators in the original expression).

3.4 A Minor Variation on Push

The Push[c] command is actually a family of combinators, one for each distinct combinator in the original expression. If desired we can replace this family of combinators with a single Push combinator that takes the combinator to be pushed as an argument. However, to avoid parentheses, the combinator to be pushed must be the *second* argument, not the first.

$$\mathsf{Push}\ s\ c\ k = k\ (\mathsf{Pair}\ c\ s)$$

We can then write the expression $c_1 c_2$ as

$$\mathsf{Begin}\ \mathsf{Push}\ c_1\ \mathsf{Push}\ c_2\ \mathsf{Apply}\ \mathsf{End}$$

Although this version of Push is probably preferable in practice, it does suffer from the theoretical disadvantages of increasing the size of the resulting expressions from $2n + 1$ to $3n + 1$ and the number of distinct combinators from $k + 3$ to $k + 4$.

4 Reducing the Number of Combinators

A few hours after responding to Cadet K's email, I received the following voice-mail:

> *Sir, this is Cadet K again. My roommate thought he could fix my keyboard, so I let him try, but he made it worse! Now it's making a funny burning smell, and only two keys work, the P and the A. Is there any way I can still finish the homework?*

It is well known (Schönfinkel, 1924) that any combinator expression—indeed, any closed λ-term—can be rewritten using only the combinators S, K, and I, or even just S and K, since $\mathsf{I} = \mathsf{S}\ \mathsf{K}\ \mathsf{K}$. In fact, Fokker (1992) and others (Barendregt, 1984) have shown that one combinator suffices. For example, Fokker defines a combinator

$$\mathsf{X} = \lambda h.h(\lambda f.\lambda g.\lambda x.(fx)(gx))(\lambda x.\lambda y.\lambda z.x)$$

such that $\mathsf{K} = \mathsf{XX}$ and $\mathsf{S} = \mathsf{X(XX)}$.

Taking an arbitrary combinator expression involving only applications and X, we can use the techniques of the previous section to obtain a flat expression involving the combinators Push[X], Apply, Begin, and End. Unfortunately, even if we abbreviate each combinator to a single letter, we still have two combinators too many for Cadet K. Is it possible to eliminate the Begin and End combinators? Yes.

4.1 Eliminating End

Suppose we modify Apply to detect when there is only a single element in the stack, and to return that element. Then we can replace End with an extra Apply. We first need a more sophisticated implementation of stacks, one that allows us to detect when the stack is empty. We implement stacks as lists rather than nested pairs, using the usual Church definitions of lists:

$$\begin{array}{lcl} \mathsf{Nil} & = & \lambda n.\lambda c.n \\ \mathsf{Cons} & = & \lambda h.\lambda t.(\lambda n.\lambda c.c\ h\ t) \\ \mathsf{Lcase} & = & \lambda l.\lambda n.\lambda c.l\ n\ c \end{array}$$

The Lcase combinator takes a list and two values. If the list is empty, the first value is returned; if the list is non-empty, the second value is a function that is applied to the head and tail of the list. Rather than using Lcase directly, we will allow pattern matching on the Cons and Nil constructors as syntactic sugar for the appropriate calls to Lcase.

The last complication is that Apply currently takes two arguments, a stack and the next command. When the stack contains only a single element, we do not want Apply to take the other argument. Abbreviating each to a single letter (and specializing Push to X), the main combinators can now be defined as

$$\begin{array}{lcl} \mathsf{B}\ k & = & k\ \mathsf{Nil} \\ \mathsf{P}\ s\ k & = & k\ (\mathsf{Cons}\ \mathsf{X}\ s) \\ \mathsf{A}\ (\mathsf{Cons}\ x\ \mathsf{Nil}) & = & x \\ \mathsf{A}\ (\mathsf{Cons}\ x\ (\mathsf{Cons}\ y\ s))\ k & = & k\ (\mathsf{Cons}\ (y\ x)\ s) \end{array}$$

An expression such as X (X X) then becomes

$$\mathsf{B\ P\ P\ P\ A\ A\ A}$$

4.2 Eliminating **Begin**

The first two combinators in a flattened expression will always be Begin Push[X]. We can easily eliminate Begin by replacing Begin Push[X] with Push[X] Nil. But doing so will not reduce the number of combinators required because it exposes Nil! The Apply combinator is already serving double-duty by replacing the End combinator. Can we make it serve triple-duty by replacing Nil as well?

We modify the program to supply an extra context argument anywhere Apply might appear. This context argument is a Church boolean indicating whether this particular instance of Apply is being used as a command or a stack. Apply takes this context argument and behaves accordingly. Once we have made this change, other commands (Push[X]) and stacks (Cons) might accidentally receive this context argument, so we must modify them as well to take and discard the context. For stack contexts we use the context argument Stk = True = $\lambda x.\lambda y.x$ and for command contexts we use the context argument Cmd = False = $\lambda x.\lambda y.y$. We use the variable b for contexts.

Keeping Begin for the moment and allowing pattern matching on the context arguments, the main combinators become

$$\begin{array}{lcl} \mathsf{B}\ k & = & k\ \mathsf{Cmd}\ \mathsf{A} \\ \mathsf{P}\ b\ s\ k & = & k\ \mathsf{Cmd}\ (\mathsf{Cons}\ \mathsf{X}\ s) \\ \mathsf{A}\ \mathsf{Stk} & = & \mathsf{Nil} \\ \mathsf{A}\ \mathsf{Cmd}\ (\mathsf{Cons}\ x\ \mathsf{Nil}) & = & x \\ \mathsf{A}\ \mathsf{Cmd}\ (\mathsf{Cons}\ x\ (\mathsf{Cons}\ y\ s))\ k & = & k\ \mathsf{Cmd}\ (\mathsf{Cons}\ (y\ x)\ s) \end{array}$$

where

$$\begin{array}{lcl} \mathsf{Nil} & = & \lambda n.\lambda c.n \\ \mathsf{Cons} & = & \lambda h.\lambda t.(\lambda b.\lambda n.\lambda c.c\ h\ t) \\ \mathsf{Lcase} & = & \lambda l.\lambda n.\lambda c.l\ \mathsf{Stk}\ n\ c \end{array}$$

Notice how the Lcase combinator passes Stk to the list l, which might be Apply. To make the calls to Lcase explicit, we desugar the pattern matching in Apply, yielding

$$\begin{array}{lcl} \mathsf{A} & = & \lambda b.b\ \mathsf{Nil}\ (\lambda s.\mathsf{Lcase}\ s\ \mathsf{Dummy}\ (\lambda x.\lambda s'. \\ & & \qquad \mathsf{Lcase}\ s'\ x\ (\lambda y.\lambda s''.\lambda k.k\ \mathsf{Cmd}\ (\mathsf{Cons}\ (y\ x)\ s'')))) \end{array}$$

where the definition of Dummy is irrelevant because the stack will never be empty at that point.

Now, when we inline B to replace B P, we get P Cmd A. We have successfully replaced Nil with A, but yet another new combinator, Cmd, has been exposed. Fortunately, P ignores its first argument so we can substitute anything we like for Cmd, such as P itself.

With all these changes, we can now flatten the expression X (X X) to

$$\mathsf{P\ P\ A\ P\ P\ A\ A\ A}$$

Cadet K can finally complete his homework!

4.3 Optimality

We can now convert any combinator expression into a flat expression involving only two distinct combinators, P and A. Although there exist other pairs of combinators that would work as well, no further progress is possible. In other words, there is no single combinator U such that every combinator expression can be rewritten in the form U^n, where $\mathsf{U}^1 = \mathsf{U}$ and $\mathsf{U}^n = \mathsf{U}^{n-1}\mathsf{U}$.

Suppose such a U exists. Then, for any combinator expression E there exists an n such that $E = \mathsf{U}^n$. Consider the combinator expression F where

$$\mathsf{F} = \mathsf{Y}\ (\lambda f.\lambda x.f)$$

and Y is the usual fixpoint combinator. Note that

$$\mathsf{F}\ x = \mathsf{F}$$

If there exists an n such that $\mathsf{F} = \mathsf{U}^n$, then $\mathsf{U}^{n+k} = \mathsf{U}^n = \mathsf{F}$ for any $k \geq 0$. But this implies that there are only a finite number of distinct combinator expressions, which is nonsense. Therefore, the original assumption that U exists must be incorrect.

5 Conclusion

Has this note been an exercise in determining how many angels can dance on the keys of a broken keyboard? Yes, but there are several less frivolous implications. First, the techniques in Section 3 could be used to adapt nearly any stack-based postfix notation to combinator form. Second, the techniques of Section 4 suggest a novel approach to Gödel numbering—convert an arbitrary expression into a string of Ps and As, and read the result as a binary number!

A few hours later, as I was leaving for home, I noticed a note taped to my door, with suspiciously familiar handwriting. Fearing the worst, I locked my door and walked away without reading the note.

6 Acknowledgments

My thanks to Jeroen Fokker for many useful comments on an earlier draft. Also, since writing this note, I have learned that Mayer Goldberg invented a similar technique several years ago, but never published it.

References

Barendregt, H. P. (1984). *The lambda calculus: Its syntax and semantics.* North-Holland.

Fokker, Jeroen. (1992). The systematic construction of a one-combinator basis for lambda-terms. *Formal aspects of computing*, **4**(6A), 776–780.

Schönfinkel, M. (1924). Über die bausteine der mathematischen logik. *Mathematische annalen*, **92**, 307–316.