The Theory of Concatenative Combinators

Brent Kerby (bkerby at byu dot net).
Completed June 19, 2002. Updated February 5, 2007.

This article attempts to outline, in informal terms, a new theory of combinators, related to the theory of Combinatory Logic pioneered by Moses Schonfinkel, Haskell Curry, and others in the 1930s. Although not essential, an understanding of the classical theory of combinators may be helpful (see the links at the bottom of this article for some introductory material to combinators).

This topic is one which no doubt ought to be subjected to the rigor of modern mathematics; there are many theorems from classical combinatory logic (e.g., Church-Rosser) which we conjecture have analogues here. However, what follows is only a rough, but hopefully, friendly, introduction to the subject.

The inspiration for this theory comes from the programming language Joy, designed by Manfred von Thun. It would be very helpful if the reader is basically familiar with Joy. In Joy, data is manipulated through a stack (there are no variables); in this way, it is similar to the programming language FORTH. However, Joy goes one step further and permits (and actively encourages) pushing programs themselves onto the stack, which can then be manipulated just like ordinary data.

In fact, the theory here is basically a subset of Joy in which programs are the *only* kind of data (i.e., numbers, string literals, and other kinds of data are not part of the theory here). To someone unfamiliar with combinatory logic, it might seem that no useful computations could be done without numbers, but it will soon be seen that numeric data can be simulated using concatenative combinators, just as they could using classical combinators.

A Few Combinators: swap, dup, zap, unit, cat, cons, i, dip

Now, we'd like to introduce some fundamental combinators. Later in the article, we'll more precisely define what we mean by "combinator", but for now, roughly speaking, a combinator is a program that duplicates, destroys, rearranges, or restructures items on the stack. Perhaps the most famous combinators are "swap", "dup", and "zap" (aka "pop" or "drop"). We can express the behavior of these combinators by giving a rewrite rule for each one:

```
[B] [A] swap == [A] [B]
    [A] dup  == [A] [A]
    [A] zap  ==
```

"swap" simply swaps the top two items on the stack, reversing their order. "dup" just duplicates the top item, leaving a copy. "zap" erases the top item. Later it will be seen that "swap", "dup", "zap" have a special relation to the classic "C", "W", and "K" combinators, respectively.

Next, take a look at the combinators "cat", "cons", and "unit":

```
[B] [A] cat  == [B A]
[B] [A] cons == [[B] A]
    [A] unit == [[A]]
```

These combinators are special in that they restructure quotations. "cat" takes two quotations and concatenates them together. "cons" does the same, except that the bottom parameter is kept separately quoted inside the result; another way of thinking of "cons" is that it takes a program "A", and gives back what is still essentially the program "A", except that a parameter has been hardwired into it (later, it will be seen that "cons" has a special relation to the classic "B" combinator). "unit" takes a program "A" and leaves a program that, when executed, leaves the program "A" on the stack; also, you could think of "unit" as simply wrapping a quotation in a second layer.

Finally, take a look at the combinators "i" and "dip":

```
    [A] i   == A
[B] [A] dip == A [B]
```

These combinators are special in that they dequote stack items, removing their wrapping; in other words, these combinators execute programs on the stack. The "i" combinator simply executes the top item on the stack. The "dip" combinator executes the top item "A", but first it gets rid of the second item, which is restored after the execution of "A" is complete. The "dip" combinator will prove to be very versatile and also quite fundamental.

Interconstructability of Combinators

The eight combinators presented above are by no means all of the combinators; there are infinitely many combinators. However, eventually we'll show that from the above combinators, it is possible to construct all other combinators. In fact, it will turn out that there is a base consisting of just two combinators, from which all other combinators can be constructed.

To give an idea of how constructions can happen, it should be pointed out that the eight combinators above are not independent; many of them can be defined in terms of each other. For example, take a look at the three combinators "cat", "cons", and "unit" (these are the ones which had a restructuring effect). It is possible to construct "cat" and "unit" in terms of "cons":

```
cat  == [[i] dip i] cons cons
```

```
unit == [] cons
```

This construction of "unit" is quite simple and elegant. Notice that it employs the empty quotation "[]"; an empty quotation is in principle just like any other quotation, except that its body is empty. In constructing simpler combinators in terms of more complex ones (as we are here with "unit" in terms of "cons"), we will often find "[]" to be useful.

The construction of "cat" above needs a bit of comment. Essentially, "[i] dip i" is a program that runs the top two programs, in turn; the bottom one is run first with "[i] dip", and then the top is run with "i". Of course, "[i] dip i" itself is not a valid construction of "cat", since "cat" should not run the top two programs, but should instead leave a program on the stack that would run those two programs. This is where "cons" comes in. In all, this is how the construction works:

```
   [B] [A] [[i] dip i] cons cons
== [B] [[A] [i] dip i] cons
== [[B] [A] [i] dip i]
== [[B] i [A] i]
== [B A]
```

This construction has a bit of a caveat, which should be pointed out: in order for this construction to work out, we had apply rewrite rules inside of a quotation. If a particular formalism considers this to be valid, then we say that it has **transparent quotation**, whereas if this is not valid, then it has **opaque quotation**. The standard Joy interpreter actually has opaque quotation in that "[[B] [A] [i] dip i]" is considered distinct from "[B A]", even though both quotations would do the same thing when executed (they are distinct because, for example, "size" would give "5" in the first case and perhaps "2" in the second). However, even though the two above quotations can be distinguished, in many contexts they would be considered interchangable.

So, we just showed how "cons" could be used to construct "unit" and "cat". However, the reverse is also possible:

```
cons == [unit] dip cat
```

This constructs "cons" as basically "cat", except that the bottom parameter is wrapped up first by "unit" (the "dip" is used so that the "unit" applies to the bottom parameter rather than the top one).

Now, moving on to another construction, notice that "dip" and "swap" are quite similar, in a way; the only difference is that "dip" executes the top program, whereas "swap" leaves it quoted. This suggests the construction of swap:

```
swap == unit dip
```

It is also possible to construct "dip" in terms of "swap":

```
dip == swap unit cat i
```

It works like this:

```
   [B] [A] swap unit cat i
== [A] [B] unit cat i
== [A] [[B]] cat i
== [A [B]] i
== A [B]
```

Thus, "swap unit cat i", given two stack items, does the same thing that "dip" does with them.

Finally, we'll give a curious way in which "i" can be constructed from "dip":

```
i == dup dip zap
```

Here we are again constructing a simple primitive using more complex ones; usually this type of construction is considered unelegant, and this one is particularly bad in it relies on making an unnecessary copy (i.e., "dup" is used to make a copy that is subsequently thrown away, unused, by "zap"). The construction can perhaps be improved in this way:

```
i == [[]] dip dip zap
```

Here we insert a "[]" under the top item, using "[[]] dip" (equivalently, we could have done "[] swap"). The "[]" serves as the useless item that we dip under, and is subsequently destroyed by "zap". In some contexts we could even use this construction:

```
i == [[]] dip dip dip
```

This is the same as the last construction except that at the very end we use "dip" to get rid of our useless "[]"; this works because "dip" just executes "[]", which has no effect except to get rid of the "[]". The caveat of this construction is that the final "dip" relies on there being an item to dip under; it doesn't matter what the item is, but there needs to be one there, otherwise an error may be generated. To put things in classical terms, a formalism like the "eta-theory" (one with a principle of extensionality) would consider "[] dip" to always be a valid program that does nothing, whereas the "beta-theory" would not. In essense, the principle of extensionality corresponds to the assumption that we have an infinite number of stack items at our disposal, if we need them (which, in interpretation, may not necessarily valid).

Lambdas

Now we'd like to introduce a construct that will turn out to be analagous to the lambda of the classical lambda calculus. We'll use the expression

```
A\
```

to mean "pop the top item off the stack and store it in the 'A' variable". Then, subsequent use of the letter "A" will result in the execution of the program that was stored in it. For example, one could do

```
A\ B\ A [B]
```

This would pop the top two items off the stack and call them "A" and "B"; then, "A" (the used-to-be top item of the stack) is executed, and then "B" is pushed back on the stack. Note that this is what "dip" does, and in the fact the above expression is a construction of "dip" using lambdas.

At this point, one might object that the above expression is not quite a proper construction of "dip", because it has the side effect of leaving the variables "A" and "B" with values still stored in them. This could be remedied by adding a construct to end the scope of a variable (i.e., to erase the value in it). For example, suppose we defined "A$" to end the scope of "A"; then the proper construction of dip would be:

```
A\ B\ A [B] B$ A$
```

However, since there a variety of ways to formalize scoping, and since the lack of scoping won't pose much of a problem in this article, we won't really bother with it.

Now that we've defined the lambda construct, we can give a more precise definition of what a combinator is. First of all, a **proper combinator** is a program that can be expressed as a series of lambdas, followed by a concatenative expression purely in terms of the variables used in those lambdas; for example, these are all

proper combinators:

```
A\ B\ C\ C [[B C] C] A
A\ A [] A
A\ B\
[]
```

The second combinator may seem a little exotic, since it uses an empty quotation (the last combinator listed, "[]", is even more so exotic, since it takes no parameters); however, we see no reason to exclude it. Even the empty program itself, "", qualifies as a proper combinator (although it's a rather trivial one). Note, the third combinator uses neither of its parameters, which is acceptable; at least, it is acceptable in the sense that it still qualifies as being a proper combinator; however, there is interest in a theory which, like the "I-calculus" of classical combinators, excludes combinators that ignore one or more of their parameters; we will consider this in the section on "Conservative Completeness".

Then, a **combinator** is any closed expression in terms of lambdas, concatenation, quotations, and variables; by "closed", we mean that all variables must be bound by a lambda, although we are not requiring all lambdas to occur at the head. (Equivalently, we could define a combinator as anything which can be written as a combination of proper combinators, i.e., an expression built from proper combinators by concatenation and quotation alone.) Here are some examples of combinators which are not proper combinators (so, we will call them **improper combinators**):

```
A\ B\ A B [C\ A]
A\ B\ [[A] dip] B
[i]
```

The second example contains, not just variables, but also "dip"; this is fine, since "dip" itself can be expressed in terms of lambdas. Similarly, in the third example, "i" can be expressed in terms of lambdas.

An Abstraction Algorithm

An interesting fact is that it is possible to eliminate lambdas using combinators. More specifically, using just the combinators "i", "dip", "cons", "dup", and "zap", it is possible to rewrite any expression containing lambdas to an expression not containing lambdas. A consequence of this is that all combinators can be constructed from just "i", "dip", "cons", "dup", "zap"; in other words, we'll say that {i, dip, cons, dup, zap} is a complete base.

The process is somewhat similar to the analagous one in classical combinatory logic. Suppose we want to eliminate the lambda in this program:

```
A\ B A [C A]
```

This program begins by popping "A" off, and then executing "B" (which does not use "A"). We could postpone the lambda by not immediately popping "A" off the stack, but by instead using "dip" to make "B" look under it; here's a reduction of the above program:

```
[B] dip A\ A [C A]
```

Note that this is progress toward eliminating the lambda, since we have reduced its scope. At this point, we should note that "A", after it is popped off, is used twice. Using "dup", we can reduce this lambda to two lambdas that only use their variable once:

```
[B] dip dup A1\ A2\ A2 [C A1]
```

And now, "A2\ A2" can be replaced by just "i":

```
[B] dip dup A1\ i [C A1]
```

Next, we use "dip" again:

```
[B] dip dup [i] dip A1\ [C A1]
```

Now, we have the situation where "A1" is popped off and then a quotation is pushed which uses it; we can reduce this using "cons":

```
[B] dip dup [i] dip [A1\ C A1] cons
```

Now the lambda is easily eliminated:

```
[B] dip dup [i] dip [[C] dip A1\ A1] cons
[B] dip dup [i] dip [[C] dip i] cons
```

We didn't need "zap" in this example, but it would have been neccessary if we had been trying to eliminate a **vacuous** lambda (i.e., a lambda in which the variable is never used).

To state things more precisely:

- First, every lambda x\ where there is no occurence of x in scope, is replaced by "zap". Scope extends all the way to the right until we hit a "]", or the end of the program (but if we allow shadowing of variables, this definition of scope will need some refinement -- or we can simply change the names to be unique before we begin)
- Then, for every lambda "x\" that has two or more references to x in scope, replace "x\" with "dup x1\ x\" (where x1 is a name which is not used elsewhere) and change the first occurence of x to x1; continue this until every lambda has exactly one occurence of its variable.
- Consider the last (furthest right) lambda "x\". It will be immediately followed by an atom or quotation which will either (1) be equal to x, (2) be a quotation containing x, or (3) neither be or contain an occurence of x. Apply these three rules respectively

```
(1)   x\ x   => i
(2)   x\ [Z] => [x\ Z] cons
(3)   x\ C   => [C] dip x\
```

We should note that this is by no means the only or the best possible abstraction algorithm; in fact, this one tends to yield very unwieldly results even for expressions of only modest size. Several improvements are possible, but this will not be discussed in this article. The important thing for our purpose here is simply that the algorithm works.

The sip Combinator

The algorithm above demonstrates that {i, dip, cons, dup, zap} is a complete base. Now, those five combinators are by no means the only ones that can be chosen to form a base. Much of the remainder of this article will be exploring the other possibilities for bases. Right now, we'd like to introduce a very versatile combinator called "sip" and show how it can be used in a base:

```
[B] [A] sip == [B] A [B]
```

You can think of "sip" as executing the top program "A" on the stack, except that it first saves a copy of the

next item "B" and restores it when "A" is done.

In a way, "sip" resembles "dip" in the way that it runs a program and then restores data; but also, "sip" has a bit of "dup" in it, in that it uses an item multiple times. In fact, "sip" can play the role of both "dip" and "dup" in the sense that {i, cons, sip, zap} is a complete base. To show this, we only have to demonstrate that "dip" and "dup" can be constructed from it; "dup" is rather simple:

```
dup == [] sip
```

To construct "dip" is a bit trickier. First, we'll try to construct "[A] dip" (a particular instance of "dip", so to speak), which is an easier task. Well, "[A] sip" does essentially the same thing as "[A] dip" except that it leaves an undesired copy of the next stack item ("B") for the program "A"; but we can easily get rid of this extra copy by using "zap", in this way:

```
[A] dip == [zap A] sip
```

So, this gives us a way of constructing an instance of "dip", but it remains to construct "dip" itself. We'll do by using a lambda:

```
dip == A\ [zap A] sip
```

Of course, this is not really a satisfactory construction of "dip"; we'd like to get rid of the lambda. Fortunately, it is not difficult:

```
dip == A\ [zap A] sip
    == [A\ zap A] cons sip
    == [[zap] dip A\ A] cons sip
    == [[zap] dip i] cons sip
```

Now, this construction still has a problem in that "dip" occurs on the right hand side, whereas we would like the right hand side to only contain members of the base {i, cons, sip, zap}. Fortunately, in this case, the "dip" occurs as a particular instance (i.e., a quotation immediately precedes the use of "dip"), and we can get rid of it by using the fact demonstrated above that

```
[A] dip == [zap A] sip
```

or in this case, that

```
[zap] dip == [zap zap] sip
```

Using this, finally we get the construction of "dip":

```
dip == [[zap zap] sip i] cons sip
```

Constructing combinators from a restricted base is something of an art. One might wonder whether the above construction of "dip" is a good one; that is, is there a shorter or more elegant one? Well, a computer search reveals that in fact there is no shorter one, and only one of the same length:

```
dip == [[cons zap] sip i] cons sip
```

This is essentially the same construction except that "cons zap" is used as a synonym for "zap zap", since both just destroy the top two stack items. More information on the computer search tool will be given later in the article.

Schemes of Combinators: dign, buryn

Now we'd like to introduce some schemes of combinators (i.e., sequences of combinators that serve a similar function). First, we'd like to introduce "dign"; this is a combinator that reaches under "n" items and "digs" up the next item all the way to the top. For example:

```
[C] [B] [A] dig2 == [B] [A] [C]
```

Note that "dig1" is the same as "swap".

Next we'll look at a complementary scheme of combinators, "buryn". This simply takes the top item and buries it under "n" items:

```
[C] [B] [A] bury2 == [A] [C] [B]
```

Note that "bury1" is also "swap".

Now, we'll look at how to construct these combinators. The most straightforward scheme for constructing "dign" is thus:

```
dig1 == [] cons dip
dig2 == [] cons cons dip
dig3 == [] cons cons cons dip
(...)
```

In essence, how this works is that we wrap the top few stack items up with "[] cons cons ...", until all items that we are dealing with are wrapped up, except one (the item we what to be dug up); then, using "dip", we simultaneously bury and unwrap the whole package. For example, here is how "dig3" works:

```
   [D] [C] [B] [A] [] cons cons cons dip
== [D] [C] [B] [[A]] cons cons dip
== [D] [C] [[B] [A]] cons dip
== [D] [[C] [B] [A]] dip
== [C] [B] [A] [D]
```

Now we'll look at how to construct "buryn". Interestingly, "buryn" is more difficult to construct than "dign". The simplest general way, perhaps, is thus:

```
bury1 == [[] cons] dip swap i
bury2 == [[] cons cons] dip swap i
bury3 == [[] cons cons cons] dip swap i
```

Like our construction of "dign", this works by wrapping up all the items that we want to get out of the way, and then dealing with them all at once, in their bundled form. Here is how "bury3" works:

```
   [D] [C] [B] [A] [[] cons cons cons] dip swap i
== [D] [C] [B] [] cons cons cons [A] swap i
== [D] [C] [[B]] cons cons [A] swap i
== [D] [[C] [B]] cons [A] swap i
== [[D] [C] [B]] [A] swap i
== [A] [[D] [C] [B]] i
== [A] [D] [C] [B]
```

Of course, these constructions of "dign" and "buryn" are not the only ones; here are some alternatives:

```
bury4 == swap [swap [swap [swap] dip] dip] dip
      == [[[swap] cons dip] cons dip] cons dip
      == [swap] cons [dip] cons [dip] cons dip
```

```
dig4 == [[[swap] dip swap] dip swap] dip swap
```

Another Combinator Scheme: flipn

Now we'd like to introduce a very special scheme of combinators, "flipn", which simply reverses the order of the top "n" items; for example,

```
[C] [B] [A] flip3 == [A] [B] [C]
```

When we try to construct "flipn", we find that it is very useful to first define the following combinator:

```
[B] [A] take == [A [B]]
```

You can look at "take" as taking in a parameter into a quotation, similar to "cons", except that "take" takes the parameter into the end of the quotation, instead of the beginning. Another way to look at "take" is that it is kind of a postponed "dip"; it is like "dip", except that the results are left enclosed in a quotation. In fact, by using "i" to unwrap this quotation, we can use "take" to construct "dip":

```
dip == take i
```

Similarly, we could define "take" in terms of "dip":

```
take == [dip] cons cons
```

As it turns out, "take" is a very versatile combinator in its own right, but it is especially useful in constructing "flipn":

```
flip2 == [] take take i
flip3 == [] take take take i
flip4 == [] take take take take i
(...)
```

It works like this:

```
[C] [B] [A] [] take take take i
[C] [B] [[A]] take take i
[C] [[A] [B]] take i
[[A] [B] [C]] i
[A] [B] [C]
```

Essentially we're wrapping the whole thing up, in reverse order, using "take", and then we unwrap the whole thing at the end using "i". By the way, we can compact our construction of "flipn" a bit by using the fact that "[] take" is the same as "unit" and "take i" is the same as "dip":

```
flip4 == [] take take take take i
      == unit take take dip
```

An interesting connection exists among the three schemes "dign", "buryn", and "flipn": Any one of them can be defined elegantly in terms of one of the others:

```
 dig4 == bury4 bury4 bury4 bury4
      == flip4 flip5

bury4 == dig4 dig4 dig4 dig4
      == flip5 flip4
```

```
flip4 == bury3 bury2 bury1
      == dig1 dig2 dig3
```

Clearly, any of these three schemes suffice to generate any desired permutation of the stack.

Two More Schemes: dipn, sipn

We'd like to introduce extensions to the "dip" and "sip" combinators which turn out to be useful. First, "dipn":

```
             [A] dip0 == A
         [B] [A] dip1 == A [B]
     [C] [B] [A] dip2 == A [C] [B]
[D] [C] [B] [A] dip3 == A [D] [C] [B]
```

These extensions are like "dip", except that they can dip under more than item; for example, "dip2" dips under two items. Of course, it is always the top item "A" which is dequoted.

The construction of "dipn" is very similar to the construction for "buryn"; here are several possible ways:

```
dip4 == [unit cons cons cons] dip dip i
     == swap [swap [swap [dip] dip] dip] dip
     == [[[dip] cons dip] cons dip] cons dip
     == [dip] cons [dip] cons [dip] cons dip
```

Next we'll introduce "sipn":

```
             [A] sip0 == A
         [B] [A] sip1 == [B] A [B]
     [C] [B] [A] sip2 == [C] [B] A [C] [B]
[D] [C] [B] [A] sip3 == [D] [C] [B] A [D] [C] [B]
```

These extensions are like "sip" except that they save more than one item, and restore them after "A" is executed. Here is a straightforward way of constructing "sipn".

```
sip4 == [unit cons cons cons] dip [i] swat sip i
```

Here we're using a combinator "swat" as a contraction of "swap cat"; think of "swat" as prepending something to a program. And, here is a curious alternative construction of "sipn":

```
sip4 == [cons cons cons sip] dup dup dup i
```

Applicative Combinators: w, k, b, c, s

We'd now like to define several new combinators related to the classical theory of combinators:

```
    [B] [A] w == [B] [B] A
    [B] [A] k == A
[C] [B] [A] b == [[C] B] A
[C] [B] [A] c == [B] [C] A
[C] [B] [A] s == [[C] B] [C] A
```

The "w" combinator is similar to "dup", except that "w" uses an extra parameter "A" which is dequoted after the duplication. A similar relation exists between "k"/"zap", "b"/"cons", and "c"/swap. Here's a way to state it precisely:

```
w == [dup] dip i
k == [zap] dip i
b == [cons] dip i
c == [swap] dip i
```

This expresses "w", for example, as doing a dipped "dup", and then dequoting the extra "A" parameter with "i". The constructions can easily go the other way, as well, though:

```
 dup == [] w
 zap == [] k
cons == [] b
swap == [] c
```

From here, it's not too hard to see that all the classical combinators can be elegantly nestled within the concatenative theory. For example, recall the classical construction:

```
B == S(KS)K
```

In the concatenative system, this becomes:

```
b == [k] [[s] k] s
```

Perhaps the similarity is easier to see if we write the original classical construction using a more strict scheme of parentheses, where every application is written in the form "F(X)":

```
B == S(K(S))(K)
```

Now the concatenative construction is exactly identical, except the entire expression is reversed, and "()"s are replaced by "[]"s.

We say that "i", "b", "c", "w", "s", and "k" are **applicative combinators**, meaning that the right-hand-side of each one's rewrite rule is an **applicative expression**, where we recursively define an applicative expression to be either be an atom ("A", "B", etc.) or an expression of the form "[X] Y" where both "X" and "Y" are applicative expressions.

Most combinators we've discussed, of course, such as "dup", "zap", "cons", or "dip", are not applicative, since they do not contain the required single dequoted atom at the end of their right-hand-side.

Finally, we would like to introduce one more applicative combinator:

```
[B] [A] t == [A] B
```

We use this combinator to make the point that an applicative combinator need not always end by dequoting the top stack item, as "i", "w", "k", "b", "c", and "s" do; in this case, it ends by dequoting "B" (the second stack item). However, "t" is an interesting combinator in its own right. In particular, it is useful because we can map the concatenative theory of combinators into the classical one by sending "[x]" to "Tx" and "x y" (x concatenated with y) to "Bxy" (i.e., x composed with y).

Towards a Minimal Base: {cons, sip, k}

The smallest complete base we've mentioned so far is {i, cons, sip, zap}. It is well known that in classical combinatory logic there is a complete two-combinator base {S, K}. It might be wondered if there is a similar two-combinator base in the concatenative theory. Eventualy, we'll see that there is.

But first, we'd like to examine the base {cons, sip, k}. This is a natural reduction of {i, cons, sip, zap}, relying on the "k" combinator to perform both the dequoting job of "i" and the destructive job of "zap". Recall that

```
[B] [A] k == A
```

Thus, "k" executes the top program, but before it does that, it destroys the stack item beneath. "k" will be useful in our search for a minimal base.

To show that {cons, sip, k} is complete, it is sufficient to show that "i" and "zap" can be constructed from it. This can be done without much difficulty:

```
zap == [] k
  i == [] sip k
```

This "zap" works by pushing an empty quotation, which allow "k" to destroy the original first item (because "k" can only destroy the second item down); then the empty quotation is executed, which does nothing.

The "i" works by making a copy of the top item with "[] sip", and then simultaneously executing one copy and discarding the other one, using "k".

A Smaller Base: {s', k}

Now we'd like to introduce a complete two-combinator base and give some explanation for how it came about. First, we know that the two combinators "s" and "k" are very powerful:

```
    [B] [A] k == A
[C] [B] [A] s == [[C] B] [C] A
```

From them it is possible to form all applicative combinators, including "i", "b", "c", and "w". And from those we know it is easy to form "cons", "swap", and "dup" (also, from "k" we can easily form "zap"). This almost gives us completeness; however, there is no way to form "dip" or "sip" using just "s" and "k" because, roughly speaking, they provide no way to dequote items buried in the stack.

However, we can make progress if we use a slightly modified "s":

```
[C] [B] [A] [X] s' == [[C] B] X [C] A
```

This "s'" is like "s", except that it takes an extra parameter "X" on top and dequotes it in the middle of the action (which, hopefully, should allow us to form "dip"). We picked the design of "s'" so that we could easily get "s" back out of it:

```
s == [] s'
```

Now, here is a way to construct "dip":

```
dip == [] swap [] swap [k] cons s'
```

It works like this:

```
   [B] [A] [] swap [] swap [k] cons s'
== [B] [] [A] [] swap [k] cons s'
== [B] [] [] [A] [k] cons s'
== [B] [] [] [[A] k] s'
```

```
== [[B]] [A] k [B]
== A [B]
```

Alternatively, here is how to construct several of the basic combinators (including dip) straight from the primitives "s'" and "k":

```
   i == [] [k] [] s'
cons == [[] k] [] s'
 dip == [k] [[] [[]]] [] s' s' s'
 sip == [k] [] [[] [[]] s' k] s'
 dup == [[] [k] [] s'] [] [] s'
 zap == [] k
```

An Elegant Base: {cake, k}

Now we will introduce a simpler combinator, "cake", and show how it, with "k", forms a complete base:

```
[B] [A] cake == [[B] A] [A [B]]
```

The name "cake" is a contraction of "cons" and "take", which describes what this combinator does; we could express it this way:

```
cake == [cons] sip2 take
```

"cake" takes only two parameters, has a symmetrical right-hand-side, and yields quite nice constructions of the basic combinators:

```
 zap == [] k
 dip == cake k
cons == cake [] k
   i == [[]] dip k
 dup == [] cake dip dip
```

Conservative Completeness: {j', i}

In the classical theory of combinatory logic, there was some interest in systems where destructions were not allowed. In the concatenative theory, that would be analagous to a system based on, for example, the combinators {i, cons, dip, dup}, there being an absense of "zap". We will refer to this type of system as **conservative**.

The question arises, is there a more simple conservative base than {i, cons, dip, dup}. Well, applying the same idea as we did above with s', we will start with the well known conservative base {j, i} from classical combinatory logic and extend it:

```
[D] [C] [B] [A] j      == [[C] [D] A] [B] A
[D] [C] [B] [A] [X] j' == [[C] X [D] A] [B] A
```

The original "J" combinator of the classical theory is then available simply as "[] j'".

Now, it is known that in the classical theory, "J" and "I" suffice to give conservative completeness; thus, from "j" and "i" it is possible to construct "w" and "b" and thus "dup" and "cons". That would give us "i", "dup", and "cons". The only other thing we need is "dip":

```
dip == [] [] [i] j' i i
```

Explicitly, we can construct the other basic combinators thus:

```
  swap == [] [] j i i
  cons == swap [] [i] j
     t == [i] [i] j
   dup == [[] [] []] dip [t t] cons j
```

There is also a nicer two-combinator conservative base, {coup, sap}, where

```
  [C] [B] [A] coup == [[C] B] [A] [A]
      [B] [A] sap  == A B
```

"coup" being a contraction of "cons" and "dup", and "sap" being a nice combinator which executes two programs in reverse order. We can construct the elementary combinators like so:

```
i    == [] sap
cons == [] coup sap
dip  == [] cons [[] cons] sap sap
dup  == [[] []] dip coup [] cons cons sap i
```

Linear Completeness: {take, cat, i} and {cons, sap}

Now we'll examine what happens if we take the conservative restriction one step further by excluding combinators that have a duplicative effect (as well as those that a destructive effect). A system that has both these restrictions we'll call **linear**.

The most obvious linear base is {i, cons, dip}; however, one interesting alternative is {take, cat, i}. This base may be interesting because with "take" and "i" alone it is possible to reshuffle the stack in any way (remember that "flipn", and thus "buryn" and "dign" are constructible from "take" and "i"); moreover, this can be done without using any quotations except for the empty quotation "[]". For example:

```
  bury3 == [] take take take take i [] take take take i
   dig3 == [] take take take i [] take take take take i
```

Some other basic constructions in {take, cat, i} are as follows:

```
  dip == take i
 unit == [] take
 cons == [unit] dip cat
 swap == [] take take i
```

Now, we'll turn our attention to a two-combinator linear base, {cons, sap}. We can construct "i" and "dip" in this way:

```
    i == [] sap
  dip == [] cons [[] cons] sap sap
```

This gives us "i", "dip", and "cons", which is sufficient for linear completeness.

Iterators: repn, zn

Finally, we'd like to introduce two more schemes of combinators; first, "repn", which simply takes a program and executes it a certain number of times.

```
[A] rep0 ==
[A] rep1 == A
[A] rep2 == A A
[A] rep3 == A A A
(...)
```

To construct these, it is handy to define a combinator "run" as follows:

```
[A] run == A [A]
```

Then, "repn" can be constructed thus:

```
rep0 == zap
rep1 == run zap
rep2 == run run zap
rep3 == run run run zap
(...)
```

To compact things, we can use the fact that "run zap == i":

```
rep0 == zap
rep1 == i
rep2 == run i
rep3 == run run i
rep4 == run run run i
(...)
```

Finally, we'd like to introduce "zn", a series of applicative combinators:

```
[B] [A] z0 == B
[B] [A] z1 == [B] A
[B] [A] z2 == [[B] A] A
[B] [A] z3 == [[[B] A] A] A
(...)
```

The interesting thing about these combinators is that they can be used to represent numbers, by defining the operations of arithmetic (multiplication, addition, and exponentiation) in this way:

```
* == b
+ == [cons b] cons s
^ == i
```

For example, it will be found that:

```
z5 == [z3] [z2] +
z6 == [z3] [z2] *
z9 == [z3] [z2] ^
```

These combinators are called the **Church numerals**, in honor of Alonzo Church, who first put them forth.

If a similar attempt is made to try to use "repn" to represent numbers, we'll find that the appropriate arithmetic operators are as follows:

```
* == b
+ == [sip] sap
^ == [[cons] cons] sap i
```

However, in a way, this is less elegant because the above "^" operator is an improper combinator, whereas all three operators for "zn" were proper combinators.

A Brute-force Automated Construction Finder

There is a small program available at http://tunes.org/~iepos/joys.zip which will search for the smallest constructions of a particular combinator, given a particular base. There is a README enclosed which explains the usage of the program.

Unfortunately, the program relies on a poor algorithm, which limits its use to finding constructions of only very small size. Essentially, the program works by making a particular program of size 1 (i.e., only one word long, where a quotation counts as a single word), and then giving it a test run to see if it generates the desired effect on the stack; then, the program moves onto the next program of size 1, and tests it, and so forth, until they all programs of size 1 been exhausted, at which point the program moves on to size 2.

A dramatic improvement could result if the program would incrementally try the candidate program, and backtrack when an error occured. An error would occur, for example, when executing "dip" and only one stack item is available; also, it is safe to backtrack if a primitive is dequoted that does not match up with the goal; e.g., if the goal is "0[2][1]", and somewhere along the way, the candidate program runs "1", then it is not possible that this candidate program could ever match the goal, since the goal requires the program to begin by executing "0"; thus, by backtracking, huge masses of programs may be cut out of the search space.

A further suggestion is to postpone filling in quotations until their bodies are needed by the backtracking algorithm.

Appendix: Combinators

```
          [A] zap  ==
          [A] i    == A
          [A] unit == [[A]]
          [A] rep  == A A
          [A] m    == [A] A
          [A] run  == A [A]
          [A] dup  == [A] [A]
       [B] [A] k    == A
       [B] [A] z    == B
       [B] [A] nip  == [A]
       [B] [A] sap  == A B
       [B] [A] t    == [A] B
       [B] [A] dip  == A [B]
       [B] [A] cat  == [B A]
       [B] [A] swat == [A B]
       [B] [A] swap == [A] [B]
       [B] [A] cons == [[B] A]
       [B] [A] take == [A [B]]
       [B] [A] tack == [B [A]]
       [B] [A] sip  == [B] A [B]
       [B] [A] w    == [B] [B] A
       [B] [A] peek == [B] [A] [B]
       [B] [A] cake == [[B] A] [A [B]]
    [C] [B] [A] poke == [A] [B]
    [C] [B] [A] b    == [[C] B] A
    [C] [B] [A] c    == [B] [C] A
    [C] [B] [A] dig  == [B] [A] [C]
    [C] [B] [A] bury == [A] [C] [B]
    [C] [B] [A] flip == [A] [B] [C]
    [C] [B] [A] s    == [[C] B] [C] A
[D] [C] [B] [A] s'   == [[D] C] A [D] B
[D] [C] [B] [A] j    == [[C] [D] A] [B] A
```

```
[E] [D] [C] [B] [A] j'   == [[D] A [E] B] [C] B
```

References

Baker, Henry. **Linear Logic and Permutation Stacks -- The Forth Shall Be First**. http://home.pipeline.com/~hbaker1/ForthStack.html.

Barker, Chris. **Iota and Jot: the simplest languages?** http://ling.ucsd.edu/~barker/Iota/.

Curry, Haskell and Robert Feys. **Combinatory Logic**. North-Holland: Amsterdam, 1958.

Hindley, J. Roger, and Jonathan P. Seldin. **Introduction to Combinators and Lambda-calculus**. New York: Cambridge University Press, 1986.

Keenan, David. **To Dissect a Mockingbird**. http://uq.net.au/~zzdkeena/Lambda/index.htm.

Smullyan, Raymond. **To Mock a Mockingbird**. Alfred A. Knopf: New York, 1985.

Thun, Manfred. **The main page for the programming language Joy**. http://www.latrobe.edu.au/philosophy/phimvt/joy.html.