Combinators

Combinators are an alternative to pure functions. They do not use variables and are, therefore, immune to the scoping problems caused by conflicts of names of formal parameters. We present an introduction to combinatory algebras and show how to convert pure functions into combinators. The implementation of these ideas requires control of the order of substitutions in rewrite rules. We discuss the techniques needed to achieve this control.

Roman E. Maeder

My article "Higher-Order Functions" in the previous issue of this journal [Maeder 1995] introduced pure functions, which look like `Function[`x, *body*`]` in *Mathematica*. As we saw, the fact that a name has to be chosen for the formal parameter can lead to problems.

Here is a variant of the Curried addition from the previous article.

```
In[1]:= add = Function[y, Function[x, x + y]]
Out[1]= Function[y, Function[x, x + y]]
```

When the definition is evaluated, the formal parameter `x` is renamed `x$`.

```
In[2]:= add[y]
Out[2]= Function[x$, x$ + y]
```

Here we see why the renaming is necessary. This function adds `x` to its argument, as we expect.

```
In[3]:= add[x]
Out[3]= Function[x$, x$ + x]
```

If no renaming had taken place, we would have gotten `Function[x, x + x]`, a function that doubles its argument.

There is an alternative way to define functions. The main idea is not to use any variables at all.

Combinatory Algebras

Functional languages modeled after λ-calculus provide two basic operations: application (of functions to arguments), and abstraction. Abstraction takes a term t and a variable x and forms the function $\lambda x.t$ or `Function[`x, t`]`.

In a combinatory algebra, there is just one operation: application. Because there is no way to define functions by abstraction, some basic operations need to be built in. These are the constants or basic combinators. Combinatory terms are built up from the constants and variables by repeated application. In the literature, the application of term t_1 to term t_2 is written $t_1 t_2$, omitting the operator (similar to the way multiplication is often written without the dot). Application is taken to be left associative; that is, xyz is interpreted as $(xy)z$, not as $x(yz)$.

Remarkably, two basic combinators suffice to express all possible functions. Let us see how this works. With abstraction we can turn any term t (involving the variable x) into a function `Function[`x, t`]`. If we want to define the same function as a combinatory term we must find a term T (not containing x) such that $Tx = t$. In this case, T describes the same function as `Function[`x, t`]`. The term t is either an atom (a variable or a constant) or it is composite, $t = t_1 t_2$. For each of these possibilities, we can easily see what kind of combinators are necessary to find the corresponding T:

- If t is the variable x, the term T must be the identity combinator I, with the property

$$\mathrm{I}X = X \qquad \text{for all } X \tag{1}$$

  Therefore, $Tx = \mathrm{I}x = x$, as required.

- If t is a constant or variable c, other than x, then the function `Function[`x, c`]` is constant, returning c no matter what its argument is. Therefore, we need a constructor for constant functions, the combinator K with the property

$$\mathrm{K}XY = X \qquad \text{for all } X, Y \tag{2}$$

  We can set $T = \mathrm{K}c$. Then $Tx = \mathrm{K}cx = c$, as required.

- If t is a composite term $t_1 t_2$, we can recursively find combinatory terms T_1 and T_2 with the properties $T_1 x = t_1$ and $T_2 x = t_2$. To build T we need a combinator S that distributes an argument onto two terms:

$$\mathrm{S}XYZ = XZ(YZ) \qquad \text{for all } X, Y, Z \tag{3}$$

  We can set $T = \mathrm{S}T_1T_2$. Then $Tx = \mathrm{S}T_1T_2x = T_1x(T_2x) = t_1t_2$, as required.

Roman Maeder is one of the designers of Mathematica and the author of three books on Mathematica-related topics. He is now a professor of computer science at the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland.

The combinator I is even redundant. We can use SKK instead, because $SKKX = KX(KX) = X$, according to equations 3 and 2. However, we will keep working with all three combinators, to make our sometimes rather complicated terms more readable.

Formally, a *combinatory algebra* is a structure with one binary operation (application) that contains two constants S and K with the desired properties stated in equations 3 and 2. Combinatory terms are defined inductively:

- A variable is a combinatory term.
- A constant is a combinatory term.
- If t_1 and t_2 are combinatory terms, then $t_1 t_2$ (the application of t_1 to t_2) is a combinatory term.

A *combinator* is a term that does not contain any variables. If the combinatory algebra doesn't contain any constants besides S and K, combinators will be terms containing only S and K.

Combinatory algebras were developed by Schönfinkel and Curry in the 1920s [Schönfinkel 1924; Curry 1930]. They studied systems of combinatory logic (logic without variables), including the S and K combinators. These systems were later found to be paradoxical and thus unsuitable for the description of logic. (We shall explain the paradoxical nature later, in the section on fixed points.) Combinators have an obvious operational nature, however: they transform terms in various ways. As such it is no surprise that interest in them was renewed in theoretical computer science. Programs, too, have an operational aspect; it seems natural to use λ-calculus or combinatory algebras to describe the semantics of programming languages.

In the formal treatment of combinatory logic, the combinators S and K define a reduction (term-rewriting) relation according to the rules

$$\begin{aligned} KXY &\rightarrow X \\ SXYZ &\rightarrow XZ(YZ) \end{aligned} \tag{4}$$

A term that does not contain any reducible subterms is said to be in normal form.

It is, of course, trivial to implement these rules in *Mathematica*. We can use the familiar form of application of a function to an argument $t_1[t_2]$ to realize the operation of application $t_1 t_2$ in a combinatory algebra. This representation is also left associative: $t_1[t_2][t_3]$ stands for $t_1 t_2 t_3 = (t_1 t_2) t_3$, rather than for $t_1(t_2 t_3)$ (the latter is $t_1[t_2[t_3]]$). As a consequence, you will encounter expressions that are deeply nested in their head position, something that is rarely seen in other applications.

The symbols S, K, and I are so standard in the literature that we do not want to use different names. To protect them from interference with built-in objects (I being the main problem), we define them in their own context. The reduction rules for S, K, and I (equation 4) are easily programmed. Here is part of the package `Combinators.m`:

```
BeginPackage["Combinators`"]

`S::usage = "S is the distributive combinator."
`K::usage = "K is the constant combinator."
`I::usage = "I is the identity combinator."

Begin["`Private`"]

I[x_] := x
K[x_][y_] := x
S[x_][y_][z_] := x[z][y[z]]

End[]

Protect[S, K, I]

EndPackage[]
```

Note that we introduce the three symbols `S`, `K`, and `I` in the usage messages in the form `` `S``, `` `K``, and `` `I``. The context marks (backquotes) force them to be created in the current context ``Combinators` ``, even if they exist already in the system context. Without the initial context mark, `I` would refer to the existing symbol in the system context. Once defined, the symbol lookup rules (which search the system context last) guarantee that the unadorned form `I` refers to the combinator, rather than to the complex number.

```
In[4]:= Needs[ "Combinators`" ]
        I::shdw: Warning: Symbol I appears in multiple contexts
           {Combinators`, System`}; definitions in context
            Combinators` may shadow or be shadowed by other
            definitions.
```

The warning message alerts us to our nonstandard practice of declaring symbols that shadow system symbols. The I combinator hides the complex number `I`, which is what we want here.

```
In[5]:= ?I
        I is the identity combinator.
```

The rule takes effect as intended.

```
In[6]:= I[x]
Out[6]= x
```

We can see that I could indeed be replaced by SKK. The two combinators behave in the same way.

```
In[7]:= S[K][K][x]
Out[7]= x
```

Combinatory Abstraction

In the previous section we gave the rules for converting a term t, involving the variable x, into a combinatory term T such that $Tx = t$. The operation of converting t to T is written $\lambda^* x.t$. According to equations 1–3 it can be defined inductively as follows:

$$\begin{aligned}\lambda^* x.x &= \mathrm{I} \\ \lambda^* x.c &= \mathrm{K}c \qquad \text{for atoms } c \neq x \\ \lambda^* x.t_1\, t_2 &= \mathrm{S}(\lambda^* x.t_1)\,(\lambda^* x.t_2)\end{aligned} \tag{5}$$

These rules are sufficient, but they can give unnecessarily complicated results. There are two obvious improvements we can make. First, we can treat composite terms not containing x as if they were constant atoms. For constants c and d we get

$$\lambda^* x.cd = \mathrm{S}\,(\lambda^* x.c)(\lambda^* x.d) = \mathrm{S}(\mathrm{K}c)\,(\mathrm{K}d)$$

However, the simpler term $\mathrm{K}(cd)$ works as well, because $\mathrm{K}(cd)X = cd = \mathrm{S}(\mathrm{K}c)\,(\mathrm{K}d)X$. We will, therefore, use the rule $\lambda^* x.t = \mathrm{K}t$ for *any* terms not containing x, not only for atoms. A second important special case is $\lambda^* x.t_1 x$, where t_1 does not contain x. Instead of $\mathrm{S}(\lambda^* x.t_1)\,(\lambda^* x.x) = \mathrm{S}(\mathrm{K}t_1)\mathrm{I}$, we can simply use t_1, since obviously $t_1 x = t_1 x$. Additionally, we define a form of λ^* with several arguments according the recursion

$$\lambda^* x_1 x_2 \ldots x_n.t = \lambda^* x_1.\lambda^* x_2. \ldots \lambda^* x_n.t$$

Here are the definitions for `LambdaStar[`x`][`t`]`, our implementation of $\lambda^* x.t$:

```
LambdaStar[x_Symbol][x_] := I

LambdaStar[x_Symbol][c_] /; FreeQ[c, x] := K[c]

LambdaStar[x_Symbol][f_[x_]] /; FreeQ[f, x] := f

LambdaStar[x_Symbol][a_[b_]] :=
   S[LambdaStar[x][a]][LambdaStar[x][b]]

LambdaStar[x_Symbol, y__][t_] :=
   LambdaStar[x][ LambdaStar[y][Unevaluated[t]] ]
```

The extra `Unevaluated` in the last rule preserves `t` in unevaluated form throughout the recursion, if `LambdaStar` was originally called in the form `LambdaStar[`x_1`, …, `x_n`][Unevaluated[`t`]]`. (We will need this form in the next section.) If no `Unevaluated` is present in the original call, nothing bad happens because `t` is evaluated in the original call and is then merely passed along.

These two expressions demonstrate the identity and constants, respectively.

```
In[8]:= {LambdaStar[x][x], LambdaStar[x][c]}
Out[8]= {I, K[c]}
```

We can always check the results by applying them to `x`. We should get the original expression back. (Recall that `Through[{`f_1`, `f_2`}[`*args*`]]` gives `{`f_1`[`*args*`], `f_2`[`*args*`]}`.)

```
In[9]:= Through[ %[x] ]
Out[9]= {x, c}
```

This result is the equivalent of the η rule in λ-calculus, which says that `Function[x, f[x]]` is the same as simply `f`.

```
In[10]:= LambdaStar[x][f[x]]
Out[10]= f
```

Here is an example where the function itself depends on `x`.

```
In[11]:= LambdaStar[x][x[e]]
Out[11]= S[I][K[e]]

In[12]:= %[x]
Out[12]= x[e]
```

Self-application leads to this term.

```
In[13]:= LambdaStar[x][x[x]]
Out[13]= S[I][I]
```

When this term is applied to itself we get an infinite iteration. The term SII(SII) does not have a normal form. It has a single infinite reduction: SII(SII) $\to$ I(SII)(I(SII)) $\to$ SII(SII).

```
In[14]:= %[%]
         $IterationLimit::itlim: Iteration limit of 4096 exceeded.
Out[14]= Hold[I[S[I][I]][I[S[I][I]]]]
```

Here is an example with two variables.

```
In[15]:= LambdaStar[x, y][y[x][y]]
Out[15]= S[S[K[S]][S[K[S[I]]][K]]][K[I]]
```

Interfacing with Built-In Functions

Unary functions in *Mathematica* can simply be treated as constants in our combinatory algebra. The treatment of binary functions is not as straightforward. In combinatory algebras, functions have only one argument, as they do in λ-calculus. Functions of several variables can be Curried: Instead of `f[`x`, `y`]`, we use `f[`x`][`y`]`. To interface with binary functions built into *Mathematica*, we can simply define a combinator that represents the Curried form of a function. We call it **c**. It has the property that

$$\mathbf{c}FXY = F(X, Y) \qquad \text{for all } F, X, Y \tag{6}$$

The corresponding rule for λ^* is

$$\lambda^* x.f(a, b) = \lambda^* x.\mathbf{c} f a b \tag{7}$$

We cannot implement this rule naively. The term $\mathbf{c} f a b$ would turn back into $f(x, y)$ by equation 6. Therefore, we apply the rule for S once to get $\mathrm{S}(\lambda^* x.\mathbf{c} f a)(\lambda^* x.b)$. Here are the necessary additions to our package:

```
`c::usage =
   "c[f] represents the binary function f in Curried form."

c[f_][x_][y_] := f[x, y]

LambdaStar[x_][f_[a_, b_]] :=
   S[LambdaStar[x][c[f][a]]][LambdaStar[x][b]]
```

Here is the term representing a binary sum as a combinatory term.

```
In[16]:= LambdaStar[x, y][x+y]
Out[16]= S[S[K[S]][S[K[K]][c[Plus]]]][K[I]]
```

It checks out as expected.

```
In[17]:= %[a][b]
Out[17]= a + b
```

Abstracting over y first gives a different term. We have not expressed the fact that addition is commutative.

```
In[18]:= LambdaStar[y, x][x+y]
Out[18]= S[K[S[c[Plus]]]][K]
```

Here is a more complicated example involving one binary function (multiplication) and two unary ones.

```
In[19]:= LambdaStar[x][Sin[x] Cos[x]]
Out[19]= S[S[K[c[Times]]][Cos]][Sin]
```

Converting Functions to Combinators

Now we have all the tools needed to convert pure functions into combinators. We can simply turn the function Function[x, t] into LambdaStar[x, t]. The result of applying either to an argument a, that is, LambdaStar[x, t][a] or Function[x, t][a], is t /. x -> a. Therefore, we can use the combinator in place of the pure function.

Because the syntax for pure functions allows some variations, we need several rules. We can also deal with functions of several variables by turning Function[{*vars...*}, *body*] into LambdaStar[*vars...*][*body*]. Here is the definition of the conversion operation toCombinators[*expr*]:

```
toCombinators::usage = "toCombinators[expr] converts all
   pure functions in expr into combinators."

SetAttributes[{leafQ, unevQ}, HoldFirst]
leafQ[expr_] := FreeQ[Hold[expr], Literal[Function[_, _]]]
unevQ[expr_] := leafQ[expr] &&
              FreeQ[Hold[expr], Literal[LambdaStar[__][_]]]

toCombinators[ expr_ ] :=
    expr //. {
        Literal[Function[{}, body_]] :> body,
        Literal[Function[{x__}, body_?unevQ]] :>
            LambdaStar[x][Unevaluated[body]],
        Literal[Function[x_, body_?unevQ]] :>
            LambdaStar[x][Unevaluated[body]],
        Literal[Function[{x__}, body_?leafQ]] :>
            LambdaStar[x][body],
        Literal[Function[x_, body_?leafQ]] :>
            LambdaStar[x][body]
    }
```

The rules contain a few subtleties, which we want to explain here. The predicates leafQ and unevQ in the patterns guarantee that innermost pure functions are converted first, before the functions containing them. The predicate unevQ[*expr*] is true if *expr* contains no functions or instances of LambdaStar; leafQ[*expr*] is true if *expr* contains no functions. Only the innermost functions contain no other functions; they are the only ones for which these predicates are true the first time the substitution operation //. applies the rules.

We give the predicates the attribute HoldFirst, because we want to apply them to parts of our expressions that are not evaluated, such as bodies of pure functions. A predicate in a pattern, such as Function[x_, body_?*pred*], is tested by first matching the pattern variable body with an expression *expr* and then evaluating *pred*[*expr*]. If *pred* didn't have the attribute HoldFirst, *expr* would be evaluated in the normal way as the argument of *pred*.

This example shows the effect just described. Our "predicate" prints out its argument, so we can see exactly what expression it receives as argument.

```
In[20]:= Hold[x+x] /. Hold[expr_?Print] :> nothing
         2 x
Out[20]= Hold[x + x]
```

The argument of the predicate is 2x, rather than x+x.

To further preserve the argument of the predicate in unevaluated form, it is wrapped in Hold when it is used as the argument of FreeQ. Observe also that we wrap all patterns involving Function or LambdaStar in Literal to prevent their evaluation.

If we omit Literal around the pattern Function[_, _] we get an error message, because Function[_, _] is evaluated (as the argument of FreeQ). Treated as a pure function, it is syntactically wrong, as the error message (correctly) points out.

```
In[21]:= FreeQ[Hold[Function[x, x+x]], Function[_, _]]
         Function::flpar:
            Parameter specification _ in Function[_, _]
              should be a symbol or a list of symbols.
Out[21]= False
```

There is no need to evaluate the pattern, so it is best wrapped in Literal.

```
In[22]:= FreeQ[Hold[Function[x, x+x]], Literal[Function[_, _]]]
Out[22]= False
```

The order of conversion of the pure functions is important: If an outer pure function containing another one in its body were converted first, the Unevaluated would prevent the code of the inner LambdaStar from ever being called.

During conversion, inner pure functions are turned into (unevaluated) expressions involving LambdaStar. Therefore, if the body of a function contains an instance of LambdaStar, we *must* evaluate it to trigger the code of LambdaStar. (Normally, we do not want to evaluate bodies of functions.) Because the rules turn pure functions into something else, every pure function in our expression will eventually be a leaf; therefore, the conditions do not prevent that all functions are converted.

Let us now see the combinators obtained from some important pure functions. Our rules work nicely to convert the pure functions corresponding to the three basic combinators S, K, and I into their simplest forms.

```
In[23]:= toCombinators[ Function[x, x] ]
Out[23]= I

In[24]:= toCombinators[ Function[{x, y}, x] ]
Out[24]= K

In[25]:= toCombinators[ Function[ {x, y, z}, x[z][y[z]]] ]
Out[25]= S
```

Some of the resulting combinators have standard names. We shall need them in the next section.

This a constant function that always returns `c`.

```
In[26]:= toCombinators[ Function[x, c] ]
Out[26]= K[c]
```

Our inside-out conversion order makes sure that scoping rules are observed. This nested function returns its *second* argument, not its first one.

```
In[27]:= toCombinators[ Function[x, Function[x, x]] ]
Out[27]= K[I]
```

We can easily check our statement.

```
In[28]:= %[x][y]
Out[28]= y
```

The **B** combinator represents function composition.

```
In[29]:= B = toCombinators[ Function[{f, g, x}, f[g[x]]] ]
Out[29]= S[K[S]][K]
```

Verification is trivial, since we constructed it as a pure function with exactly this body.

```
In[30]:= B[f][g][x]
Out[30]= f[g[x]]
```

The **C** combinator exchanges the two arguments of a Curried function. The symbol `C` also appears in the system context; therefore, we force it to be created in the current (global) context in the same way as we did in the package.

```
In[31]:= `C = S[B[B][S]][K[K]]
         C::shdw: Warning: Symbol C appears in multiple contexts
            {Global`, System`}; definitions in context Global`
             may shadow or be shadowed by other definitions.
Out[31]= S[S[K[S[K[S]][K]]][S]][K[K]]

In[32]:= C[f][x][y]
Out[32]= f[y][x]
```

The **W** combinator doubles its second argument.

```
In[33]:= W = LambdaStar[x, y][x[y][y]]
Out[33]= S[S][K[I]]

In[34]:= W[f][x]
Out[34]= f[x][x]
```

Here is the Curried form of addition `add` with which we started this article. No variables are left; therefore, no renaming problems can occur.

```
In[35]:= toCombinators[ Function[y, Function[x, x+y]] ]
Out[35]= S[K[S[c[Plus]]]][K]
```

This expression corresponds to `add[x]` in the input `In[3]` above. It is a function that adds `x` to its argument.

```
In[36]:= %[x]
Out[36]= S[c[Plus]][K[x]]
```

Indeed, that is what it does.

```
In[37]:= %[a]
Out[37]= a + x
```

The use of `Unevaluated` prevents the evaluation of the body of the pure functions during the conversion.

```
In[38]:= toCombinators[ Function[x, x+x] ]
Out[38]= S[c[Plus]][I]
```

If we hadn't used `Unevaluated`, the body `x+x` would have turned into `2x` and we would have gotten this combinator.

```
In[39]:= toCombinators[ Function[x, 2x] ]
Out[39]= S[K[c[Times][2]]][I]
```

(In this simple case, the two resulting functions behave the same way, however.)

Fixed Points

Our article on higher-order functions in the last issue of this journal discussed fixed points of functions. We constructed an operator Y such that $Y(f) = f(Y(f))$ The same construction works also in combinatory algebras.

We can define **Y** in terms of the combinators introduced in the previous section. **Y** has the property that $\mathbf{Y}f = f(\mathbf{Y}f)$.

```
In[40]:= Y = W[S][B[W][B]];
```

Fixed points are infinite terms. Here we can see how they are built up. The fixed point is an infinite sequence of nested `f`s: `f[f[f[...[]...]]]`.

```
In[41]:= Block[{$RecursionLimit = 20}, Y[f] ]
         $RecursionLimit::reclim: Recursion depth of 20 exceeded.
Out[41]= f[f[f[f[f[f[f[f[f[f[f[Hold[S[K[f]]][Hold[S[S[K[f]]][I]]]
              [Hold[S[S[K[f]]][I]]]]]]]]]]]]]
```

Although an attempt to reduce $\mathbf{Y}f$ leads to an infinite reduction, we can still prove that $\mathbf{Y}f = f(\mathbf{Y}f)$ by transforming the term according to the leftmost combinators B, W, and S as follows:

$$\begin{aligned}
\mathbf{Y}f &= \mathbf{WS}(\mathbf{BWB})f \\
&= \mathbf{S}(\mathbf{BWB})(\mathbf{BWB})f \\
&= \mathbf{BWB}f(\mathbf{BWB}f) && (8)\\
&= \mathbf{W}(\mathbf{B}f)(\mathbf{BWB}f) \\
&= \mathbf{B}f(\mathbf{BWB}f)(\mathbf{BWB}f) \\
&= f(\mathbf{BWB}f(\mathbf{BWB}f)) && (9)\\
&= f(\mathbf{Y}f)
\end{aligned}$$

where the last step follows from noting that the second term in Line 9 is equal to Line 8 and, therefore, to $\mathbf{Y}f$.

The existence of fixed points for all terms means that a negation combinator would also have a fixed point. But a logic in which a term is equal to its negation is inconsistent. It follows that this system cannot form the basis for ordinary logic.

Computations in Combinatory Algebras

Let us show that we can perform computations with integers in a combinatory algebra. We show how to realize the natural numbers in this formalism. It can be shown that all computable functions can be implemented as combinators. The definitions from this section are in the file `Numerals.m`.

```
In[42]:= << Numerals.m
```

The truth values **true** and **false** can be given as $\mathbf{T} = \mathbf{K}$ and $\mathbf{F} = \mathbf{KI}$, respectively. With this definition, the conditional **if** B **then** M **else** N is simply BMN. If B is true, we get M:

```
In[43]:= T[M][N]
Out[43]= M
```

If B is false, we get N:

```
In[44]:= F[M][N]
Out[44]= N
```

Next, we need a pairing construct. With $\mathbf{P} = \lambda^* mnz.zmn$, the term $\mathbf{P}MN$ is an encoding of the ordered pair $<M, N>$. The components can be extracted by applying it to T and F, respectively. This term represents the pair $<M, N>$.

```
In[45]:= pair = P[M][N]
Out[45]= S[S[I][K[M]]][K[N]]
```

The two components of the pair are extracted easily:

```
In[46]:= {pair[T], pair[F]}
Out[46]= {M, N}
```

Now, we can represent numbers. The integer 0 is represented by $\overline{0} = \mathbf{I}$, and an integer $n > 0$ is represented by $\overline{n} = \mathbf{PF}\overline{n-1}$. The function `num[n]` performs this conversion.

```
num[0] := I
num[n_Integer?Positive] := P[F][num[n-1]]
```

Here is the term that represents the number 5.

```
In[47]:= num[5]
Out[47]= S[S[I][K[K[I]]]][K[S[S[I][K[K[I]]]][K[S[S[I][K[K[I]]]]
          [K[S[S[I][K[K[I]]]][K[S[S[I][K[K[I]]]][K[I]]]]]]]]]
```

These definitions are useful only if we can actually compute with these numbers. Everything has been set up in such a way that the successor function `succ` and the predecessor function `pred` (with `pred[0] = 0`) are easy to implement. We need also a zero test `zero[n]` that returns true if $n = 0$, and false otherwise. Here they are:

```
succ = LambdaStar[x][P[F][x]]
pred = LambdaStar[x][x[F]]
zero = LambdaStar[x][x[T]]
```

Here is the number $\overline{2}$, defined as the two-fold successor of $\overline{0}$.

```
In[48]:= Nest[succ, num[0], 2]
Out[48]= S[S[I][K[K[I]]]][K[S[S[I][K[K[I]]]][K[I]]]]
```

Applying the predecessor to it twice gets us back to $\overline{0} = \mathbf{I}$.

```
In[49]:= Nest[pred, %, 2]
Out[49]= I
```

How can we define functions on numbers, such as addition? The standard recursive definition of *plus* is

$$plus = \lambda xy.\ \texttt{if}\ x = 0\ \texttt{then}\ y\ \texttt{else}\ plus(x-1)(y+1)$$

We convert this definition into a combinatory term p whose smallest fixed point is the desired function *plus*:

$$p = \lambda^* fxy.\texttt{zero}\,xy(f(\texttt{pred}\,x)(\texttt{succ}\,y))$$

Now, we can set $plus = \mathbf{Y}p$. Because $\mathbf{Y}$ is involved, this term does not have a normal form; evaluation would not terminate. (There are other ways to represent numbers as combinatory terms and to convert recursive definitions into combinators that do not have this problem.)

Conclusions

This example shows once more that *Mathematica* is well uited to implementing formal systems and to experimenting vith them. Here, we used expressions that have deeply ested heads. We showed how we can specify the order of valuation using replacement rules with conditions. More nformation about combinatory algebras can be found in the extbook [Hindley and Seldin 1986] or the standard refernce [Barendregt 1984].

eferences

arendregt, H.P. 1984. *The Lambda Calculus*, revised ed. Studies in Logic 103. North Holland, Amsterdam.

urry, H.B. 1930. Grundlagen der kombinatorischen Logik. *Amer. J. Math.* 52:509–536, 789–834.

Hindley, J. Roger, and Jonathan P. Seldin. 1986. *Introduction to Combinators and λ-Calculus*. London Math. Soc. Student Texts 1. Cambridge Univ. Press, London.

Maeder, Roman E. 1995. Higher-order functions. *The Mathematica Journal* 5(3): 61–67.

Schönfinkel, M. 1924. Ueber die Bausteine der mathematischen Logik. *Math. Annalen* 92:305–316.

Roman E. Maeder
ETH Zurich, Institute of Theoretical Computer Science, ETH Zentrum IFW, 8092 Zurich, Switzerland
`maeder@inf.ethz.ch`

The electronic supplement contains the package `Combinators.m` and `Numerals.m`, as well as the notebook `Combinators.ma` with the examples from this article.