0.1 \((Y \ Y)\) Works!

Copyright © 1991 by Matthias Felleisen and Daniel P. Friedman

DRAFT: September 26, 1991
A Lecture on the *Why* of *Y*

by Matthias Felleisen
Is this the function \texttt{length}?

\begin{Verbatim}
(define length
  (lambda (l)
    (if (null? l) 0
      (add1 (length (cdr l))))))
\end{Verbatim}

It sure is.

Suppose (\texttt{define \ldots \ldots}) no longer works. Can you describe in your own words what \texttt{length} does?

For one, the body of \texttt{length} cannot refer to \texttt{length}.

Then we might as well write something like this.

\begin{Verbatim}
(define length
  (lambda (l)
    (if (null? l) 0
      (add1 (hukairs (cdr l))))))
\end{Verbatim}

Yes, except that (\texttt{define \ldots \ldots}) doesn't work anymore.

So perhaps something more like this?

\begin{Verbatim}
(lambda (l)
  (if (null? l) 0
    (add1 (hukairs (cdr l)))))
\end{Verbatim}

Yes, that's better.

But what happened to the function?

It is no longer recursive.

And what does it do?

It measures the length of the empty list and nothing else.

And what does \texttt{hukairs} do?

Who cares. The function doesn't work for non-empty lists in any case.

Suppose we could name this new function. What would be a good name?

We think \texttt{length\textsubscript{0}} is great because the function only measures lists of length 0.
How would you write a function that measures the length of lists that contain one item?

Well, we could try the following.

```
(lambda (l)
  (if (null? l) 0
    (add1 (length0 (cdr l)))))
```

Almost, but (define ....) doesn’t work for length0.

So? Replace length0 by its definition.

```
(lambda (l)
  (if (null? l) 0
    (add1
     ((lambda (l)
        (if (null? l) 0
          (add1
           ((lambda (l)
              (if (null? l) 0
                (add1
                 (hukairs (cdr l))))
              (cdr l))))))))
```

And what’s a good name for this function?

That’s easy: length1.

Is this the function that would measure the length of lists that contain two items?

Yes, this is length2. We just expand the call to hukairs to get the next version of length.

```
(lambda (l)
  (if (null? l) 0
    (add1
     ((lambda (l)
        (if (null? l) 0
          (add1
           (hukairs
            (cdr l))))
        (cdr l)))))))
```

Now, what do you think recursion is?

What do you mean?
Well, we have seen how to measure the list with no items, with one item, with two, and so on. How could we get the function length back?

If we could write an infinite function, we could write \( \text{length}_\infty \).

But we can't write an infinite function. And we still have all these repetitions and patterns in these functions.

All these programs contain a function that looks like length, and that's not right. No, let's abstract out these patterns.

Is this the right way to rewrite length, so that length reappears? It's worth a try.

\[
((\lambda (\text{length})
     (\lambda (l)
       (if (null? l) 0
        (add1 (\text{length} (\text{cdr} l))))))
\text{hukairs})
\]

Rewrite length in the same style.

\[
((\lambda (\text{length})
     (\lambda (l)
       (if (null? l) 0
        (add1 (\text{length} (\text{cdr} l))))))
((\lambda (\text{length})
     (\lambda (l)
       (if (null? l) 0
        (add1 (\text{length} (\text{cdr} l))))))
\text{hukairs})
\]
And \textit{length}_2.

\begin{verbatim}
((lambda (length)
  (lambda (l)
    (if (null? l) 0
     (add1 (length (cdr l))))))
((lambda (length)
  (lambda (l)
    (if (null? l) 0
     (add1 (length (cdr l))))))
((lambda (length)
  (lambda (l)
    (if (null? l) 0
     (add1 (length (cdr l)))))))
  hukairs))
\end{verbatim}

Close, but there are still repetitions. True. Let’s get rid of them.

Where should we start? Name the function that takes \textit{length} as an argument and that returns a function that looks like \textit{length}.

What’s a good name for this function? What about \textit{mk-length} for “make \textit{length}”? No problem.

\begin{verbatim}
)((lambda (mk-length)
  (mk-length hukairs))
((lambda (length)
  (lambda (l)
    (if (null? l) 0
     (add1 (length (cdr l)))))))
\end{verbatim}
0.1. \((Y \ Y)\) WORKS!

<table>
<thead>
<tr>
<th>Is this (length_1)?</th>
<th>It sure is. And this is (length_2).</th>
</tr>
</thead>
</table>
| \[
\lambda \text{mk-length} \\
  \lambda \text{length} \\
  \lambda \text{l} \\
  \begin{cases} 
    \text{null? l } 0 \\
    \text{add1 (length (cdr l)))}
  \end{cases}
\] | \[
\lambda \text{mk-length} \\
  \lambda \text{length} \\
  \lambda \text{l} \\
  \begin{cases} 
    \text{null? l } 0 \\
    \text{add1 (length (cdr l)))}
  \end{cases}
\] |

<table>
<thead>
<tr>
<th>Can you do (length_3)?</th>
<th>Here we go.</th>
</tr>
</thead>
</table>
| \[
\lambda \text{mk-length} \\
  \lambda \text{length} \\
  \lambda \text{l} \\
  \begin{cases} 
    \text{null? l } 0 \\
    \text{add1 (length (cdr l)))}
  \end{cases}
\] | \[
\lambda \text{mk-length} \\
  \lambda \text{length} \\
  \lambda \text{l} \\
  \begin{cases} 
    \text{null? l } 0 \\
    \text{add1 (length (cdr l)))}
  \end{cases}
\] |

<table>
<thead>
<tr>
<th>So what is recursion?</th>
<th>It is like an infinite tower of applications of (mk\text{-length}) to an arbitrary function.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Do we really need an infinite tower?</th>
<th>Not really of course. Everytime we use (length) we only need a finite number, but we never know how many.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Could we guess how many we need?</th>
<th>Sure, but we may not guess a large enough number.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>When do we find out that we didn't guess a large enough number?</th>
<th>When we apply the function (hukairs) that is passed to the first (mk\text{-length}).</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>What if we could create another application of (mk\text{-length}) to (hukairs) at this point?</th>
<th>That would postpone the problem by one, and besides, how could we do that?</th>
</tr>
</thead>
</table>
Well, since nobody cares what function we pass to \texttt{mk-length}, we could pass it \texttt{mk-length} initially. That’s the right idea. And then we invoke \texttt{mk-length} on \texttt{hukairs} and the result of this on the \texttt{cdr} so that we get one more piece of the tower.

Then this is still \texttt{length}\textsubscript{0}? Yes. And when we apply \texttt{mk-length} once, we get \texttt{length}\textsubscript{1}.

\begin{verbatim}
(((lambda (mk-length)
  (mk-length mk-length))
  (lambda (mk-length)
    (lambda (l)
      (if (null? l) 0
       (add1 (mk-length (cdr l)))))))))
\end{verbatim}

Could we do this more than once? Yes, just keep passing \texttt{mk-length} to itself, and we can do this as often as we need to!

What would you call \texttt{this} function? It is \texttt{length}, of course.

\begin{verbatim}
(((lambda (mk-length)
  (mk-length mk-length))
  (lambda (mk-length)
    (lambda (l)
      (if (null? l) 0
       (add1 ((mk-length (mk-length mk-length)
                (cdr l))))))))
\end{verbatim}

Except that it no longer contains the function that looks like \texttt{length}. Can we fix that? We could extract the self-application of \texttt{mk-length} and call it \texttt{length}.

Why? Because it really makes the function \texttt{length}.

But \texttt{(mk-length mk-length)} is not a function. So what do we do?

It only returns a function.
0.1. \((Y Y)\) WORKS!

Make the self-application of \(mk\text{-}length\) into a function.

No problem, we just use the old trick of wrapping a \texttt{lambda} around the application. After all, the self-application does return a function!

Which function?

The function \textit{length}. Remember?

Ok, do it!

\begin{verbatim}
((lambda (mk-length)
  (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (if (null? l) 0
       (add1
        ((lambda (z)
           ((mk-length mk-length) z))
        (cdr l)))))))))
\end{verbatim}

Move out the new function so that we get \textit{length} back.

\begin{verbatim}
((lambda (mk-length)
  (mk-length mk-length))
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (if (null? l) 0
          (add1 (length (cdr l)))))
      (lambda (z)
        ((mk-length mk-length) z)))))
\end{verbatim}

Is this ok to do?

Yes. Think about it. We always did the reverse: When we knew what the argument to a function was, we proceeded with the function body and used the argument value whenever we saw the parameter name.

Can we extract the function that looks like \textit{length} and give it a name?

Yes, it does not depend on \textit{mk\text{-}length} at all!
Is this the right function?  

```
((lambda (le)
   ((lambda (mk-length)
      (mk-length mk-length))
    (lambda (mk-length)
      (le (lambda (z)
           ((mk-length mk-length) z))))))
   (lambda (length)
     (lambda (l)
       (if (null? l) 0
        (add1 (length (cdr l))))))
```

Yes.

What did we actually get back?  

We extracted the old function `mk-length`!

Let’s separate the function that makes `length` from the function that looks like `length`.

That’s easy.

```
(lambda (le)
   ((lambda (mk-length)
      (mk-length mk-length))
    (lambda (mk-length)
      (le (lambda (z)
           ((mk-length mk-length) z))))))
```

Does this function have a name?  

Yes, it is called the applicative-order \( Y \) combinator. At least, when we rewrite it a little bit.

```
(define Y
  (lambda (le)
    ((lambda (f)
       (le (lambda (z) ((f f) z)))
    (lambda (f)
      (le (lambda (z) ((f f) z))))))
```

Does \((define \ldots \ldots)\) work again?  

Sure, now that we know what recursion is.
<table>
<thead>
<tr>
<th>Do you now know why $Y$ works?</th>
<th>Read this chapter one more time and you will.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does $(Y Y)$ work, too?</td>
<td>And how it works!!!</td>
</tr>
</tbody>
</table>