ABSTRACT LAMBDA CALCULUS MACHINES

Werner Kluge
Dept of Computer Science
University of Kiel
D-24105 Kiel/Germany
wk@informatik.uni-kiel.de

www.informatik.uni-kiel.de/inf/Kluge/index-de.html

Milestones of λ-calculus machine development

> 1974/75 proposal by Klaus Berkling at GMD in St.Augustin/Germany of a string reduction machine with full support of an applied λ-calculus;
> 1979 completion of the design of a hardware prototype of this machine at GMD in St.Augustin/Germany > the first reduction machine worldwide;
> 1983 successfull implementation at the U of Bonn/Germany of a system of cooperating reduction machines for divide-and-conquer computations based on Berkling's original λ-calculus machine concept;
> 1990 completion of an interpreting graph reducer for a full-fledged λ-calculus that faithfully performs high-level program transformations;
> 1994 completion of a compiling graph reducer for a full-fledged λ-calculus with competitive runtime performance;
> 1996/2000 distributed implementation of the compiling graph reducer on an ncube multiprocessor system, also supporting speculative evaluation.

A small SCHEME program

```
( define twice ( lambda ( f u ) ( f ( f u ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )
```

A small SCHEME program

```
( define twice ( lambda ( f u ) ( f ( f u ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( twice square 2 ) -->  16
```

A small SCHEME program

```
( define twice ( lambda ( f u ) ( f ( f u ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )



( twice square 2 ) -->  16

( twice square ) --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )
```

A small SCHEME program

```
( define twice ( lambda ( f u ) ( f ( f u ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )



( twice square 2 )  -->  16

( twice square ) --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )

( twice twice )  --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )
```

A small SCHEME program

```
( define twice ( lambda ( f u ) ( f ( f u ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )



( twice square 2 )  -->  16

( twice square ) --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )

( twice twice )  --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )

( twice twice square )
                 --> procedure twice: expects 2 args,
                     given 1 : ( lambda(a1) ... )
```

Modifying `twice`

```
( define twice ( lambda ( f )
                 ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )
```

Modifying `twice`

```
( define twice ( lambda ( f )
               ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( ( twice square ) 2 )           -->  16
```

Modifying `twice`

```
( define twice ( lambda ( f )
               ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( ( twice square ) 2 )           -->  16

( ( ( twice twice ) square ) 2 ) -->  65536
```

Modifying `twice`

```
( define twice ( lambda ( f )
               ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( ( twice square ) 2 )           -->  16

( ( ( twice twice ) square ) 2 ) -->  65536

( ( twice twice square ) 2 )
                -->  procedure twice: expects 1 arg,
                     given 2 : ( lambda(a1) ... )
```

Modifying `twice`

```
( define twice ( lambda ( f )
               ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( ( twice square ) 2 )           -->  16

( ( ( twice twice ) square ) 2 ) -->  65536

( ( twice twice square ) 2 )
                -->  procedure twice: expects 1 arg,
                     given 2 : ( lambda(a1) ... )

( twice twice )              -->  ( lambda (a1) ... )
```

Modifying `twice`

```
( define twice ( lambda ( f )
               ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( ( twice square ) 2 )           -->  16

( ( ( twice twice ) square ) 2 ) -->  65536

( ( twice twice square ) 2 )
                -->  procedure twice: expects 1 arg,
                     given 2 : ( lambda(a1) ... )

( twice twice )              -->  ( lambda (a1) ... )

( ( twice twice ) square )   -->  ( lambda (a1) ... )
```

Modifying `twice`

```
( define twice ( lambda ( f )
               ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( ( twice square ) 2 )           -->  16

( ( ( twice twice ) square ) 2 ) -->  65536

( ( twice twice square ) 2 )
               -->  procedure twice: expects 1 arg,
                    given 2 : ( lambda(a1) ... )

( twice twice )              -->  ( lambda (a1) ... )

( ( twice twice ) square )   -->  ( lambda (a1) ... )
```

One would wish / expect to get the following:

Modifying `twice`

```
( define twice ( lambda ( f )
               ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( ( twice square ) 2 )           -->  16

( ( ( twice twice ) square ) 2 ) -->  65536

( ( twice twice square ) 2 )
                -->  procedure twice: expects 1 arg,
                     given 2 : ( lambda(a1) ... )
```

One would wish / expect to get the following:

```
( twice twice ) --> ( lambda ( u') ( lambda ( u )
                      ( u'( u'( u'( u' u ) ) ) ) ) )
```

Modifying `twice`

```
( define twice ( lambda ( f )
               ( lambda ( u ) ( f ( f u ) ) ) ) )

( define square ( lambda ( v ) ( * v v ) ) )


( ( twice square ) 2 )           -->  16

( ( ( twice twice ) square ) 2 ) -->  65536

( ( twice twice square ) 2 )
                -->  procedure twice: expects 1 arg,
                     given 2 : ( lambda(a1) ... )
```

One would wish / expect to get the following:

```
( twice twice ) --> ( lambda ( u') ( lambda ( u )
                      ( u'( u'( u'( u' u ) ) ) ) ) )

( ( twice twice ) square )
     --> ( lambda ( u )
           ( * ( * ( * ( * u u ) ( * u u ) )
               ( * ( * u u ) ( * u u ) ) ) ( .... ) ) )
```

The cause of the problem

→ all functional / function-based languages are based on a **weakly normalizing λ-calculus**

→ **weak (head) normal form**

$\rightsquigarrow$ a top level abstraction which may have redices in its body

$\rightsquigarrow$ a top level application of an n-ary abstraction to fewer than n operands that are in weak normal form

→ **weak normalization** rules out **naming conflicts**

$\rightsquigarrow$ requires only a naive β-reduction (substitution)

→ **full normal form** contains no β-redices

→ **full normalization** requires full-fledged β-reductions, including the resolution of naming conflicts

→ considered too complex, not necessary ... more excuses