miniAdapton
A Minimal Implementation of Incremental Computation in Scheme

Dakota Fisher, Matthew Hammer, William E. Byrd, Matthew Might

September 17, 2016

This material is partially based on research sponsored by DARPA under agreement number AFRL FA8750-15-2-0092 and by NSF under CAREER grant 1350344. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.
Memoization

- Remember (i.e. “make a memo of”) previous results
- Classic example: fibonacci

\[
\text{fib}(0) = 1; \quad \text{fib}(1) = 1; \quad \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)
\]

- Naively-implemented fibonacci is exponential
- Using only memoization, fibonacci can be made linear
- Memoization can yield algorithmic speedups
- Memoization forbids mutation
A Memoized Function

(define-memo (max-tree t)
  (cond
    ((pair? t)
     (max (max-tree (car t))
          (max-tree (cdr t)))))
    (else t)))
A Memoized Function

> (max 1 2 3 4)
4
> (max-tree '((1 . 2) . (3 . 4)))
4
User Session with Memoization

> (define some-tree '((1 . 2) . (3 . 4)))
> (max 1 2 3 4)
4
> (max-tree some-tree)
4

> (set-cdr! some-tree 5)
> some-tree
((1 . 2) . 5)
> (max 1 2 5)
5
> (max-tree some-tree)
4
What is incremental computation?

- Reuse previous results/computations (like memoization)
- ... specifically for changing inputs
What is Adapton?

- a general, language-based approach to incremental computation
- “memoization supporting mutation”
- How: remember not just the result of a computation, but also keep track of dependencies between computations
- Specifically, Adapton creates a dependency graph called the DCG (or demanded computation graph).
What is Adapton?

By analogy to thunks (zero-argument procedures) and promises (memoized thunks)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Thunk</th>
<th>Promise</th>
<th>Adapton “Promise”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stored</td>
<td>closure</td>
<td>+ result</td>
<td>+ dependencies</td>
</tr>
<tr>
<td>Avoids Recomputation</td>
<td>no</td>
<td>yes</td>
<td>when correct</td>
</tr>
<tr>
<td>Supports Mutation</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Aside: Why Mutation?

We live in a temporal world full of mutation, some programs dealing with mutation:
Aside: Why Mutation?

We live in a temporal world full of mutation, some programs dealing with mutation:

- Make
Aside: Why Mutation?

We live in a temporal world full of mutation, some programs dealing with mutation:

- Make
- Spreadsheets
Aside: Why Mutation?

We live in a temporal world full of mutation, some programs dealing with mutation:

- Make
- Spreadsheets
- Databases
Aside: Why Mutation?

We live in a temporal world full of mutation, some programs dealing with mutation:

- Make
- Spreadsheets
- Databases
- Interpreters
What is miniAdapton?

- a minimal version of Adapton
- try to be readable
- try to be portable
- try to be small
What is miniAdapton?

- a minimal version of Adapton
- try to be readable
- try to be portable
- try to be small
- try to be used
What is miniAdapton?

- a minimal version of Adapton
- try to be readable
- try to be portable
- try to be small
- try to be used
- try to be abused
Visualization of max-tree in Adapton
Visualization of max-tree in Adapton

(max-tree some-tree)
Visualization of max-tree in Adapton

```
(max-tree some-tree)
```

- some-tree
Visualization of max-tree in Adapton

some-tree

(max-tree some-tree)

(max-tree '((1 . 2) . (3 . 4)))
Visualization of max-tree in Adapton

(max-tree some-tree)

some-tree

(max-tree '((1 . 2) . (3 . 4)))

(max-tree '(1 . 2))
Visualization of max-tree in Adapton

some-tree

(max-tree some-tree)

(max-tree '((1 . 2) . (3 . 4)))

(max-tree '(1 . 2))

(max-tree 1)
Visualization of max-tree in Adapton

```
(max-tree some-tree)

some-tree

(max-tree '((1 . 2) . (3 . 4)))

(max-tree '(1 . 2))

(max-tree 1) (max-tree 2)
```
Visualization of max-tree in Adapton

```
(max-tree some-tree)

some-tree

(max-tree '((1 . 2) . (3 . 4)))

(max-tree '(1 . 2))
(max-tree 1) (max-tree 2) (max-tree 4) (max-tree 3)
(max-tree '(3 . 4))
```
Visualization of max-tree in Adapton

```
(max-tree some-tree)
(max-tree '((1 . 2) . (3 . 4)))
(max-tree '(1 . 2))
(max-tree 1) (max-tree 2) (max-tree 4) (max-tree 3)
(max-tree '(3 . 4))
```
Visualization of max-tree in Adapton

(max-tree some-tree)

some-tree

(max-tree '((1 . 2) . (3 . 4)))

(max-tree '(1 . 2))

(max-tree 1) (max-tree 2) (max-tree 4) (max-tree 3)

(max-tree '(3 . 4))

(max-tree 3) (max-tree 4)
Visualization of max-tree in Adapton

```
(max-tree some-tree)
(max-tree '((1 . 2) . (3 . 4)))
(max-tree '(1 . 2))
(max-tree 1) (max-tree 2) (max-tree 4)(max-tree 3)
(max-tree '(3 . 4))
```
Visualization of max-tree in Adapton

(max-tree some-tree)

some-tree

(max-tree '((1 . 2) . (3 . 4)))

(max-tree '(1 . 2))
(max-tree 1) (max-tree 2) (max-tree 4)
(max-tree (3 . 4))
(max-tree 3)
(max-tree 4)
Visualization of max-tree in Adapton

```
(max-tree some-tree)

some-tree

(max-tree '((1 . 2) . (3 . 4)))

(max-tree '((1 . 2) . 5))

(max-tree '(1 . 2))

(max-tree '3 . 4))

(max-tree 1)

(max-tree 2)

(max-tree 3)

(max-tree 4)
```
Visualization of max-tree in Adapton

(max-tree some-tree)

some-tree

(max-tree '((1 . 2) . (3 . 4)))

(max-tree '1 . 2)

(max-tree 1) (max-tree 2) (max-tree 4) (max-tree 3)

(max-tree '(3 . 4))

(max-tree '((1 . 2) . 5))

(max-tree '(1 . 2))

(max-tree '3 . 4)

(max-tree 3) (max-tree 4)
Visualization of max-tree in Adapton

\[
\text{some-tree} \quad \text{(max-tree some-tree)}
\]

\[
\text{some-tree} \quad \text{(max-tree '((1 . 2) . (3 . 4)))}
\]

\[
\text{some-tree} \quad \text{(max-tree '(1 . 2))}
\]

\[
\text{some-tree} \quad \text{(max-tree 1)}
\]

\[
\text{some-tree} \quad \text{(max-tree 2)}
\]

\[
\text{some-tree} \quad \text{(max-tree 4)}
\]

\[
\text{some-tree} \quad \text{(max-tree 3)}
\]

\[
\text{some-tree} \quad \text{(max-tree 5)}
\]

\[
\text{some-tree} \quad \text{(max-tree 5)}
\]

\[
\text{some-tree} \quad \text{(max-tree 4)}
\]
What’s in a node?

(define-record-type
  (adapton adapton-cons adapton?)
  (fields
What’s in a node?

(define-record-type
  (adapton adapton-cons adapton?)
(fields
  thunk)
What's in a node?

(define-record-type
  (adapton adapton-cons adapton?)
(fields
  thunk
  (mutable result))
What's in a node?

(define-record-type
  (adapton adapton-cons adapton?)
(fields
  thunk
  (mutable result)
  (mutable sub)
  (mutable super))
What’s in a node?

(define-record-type
  (adapton adapton-cons adapton?)
(fields
  thunk
  (mutable result)
  (mutable sub)
  (mutable super)
  (mutable clean?)))
Nodes

\begin{itemize}
\item \texttt{(max-tree some-tree)}
\item \texttt{(max-tree '((1 . 2) . (3 . 4))})
\item \texttt{(max-tree '(1 . 2))}
\item \texttt{(max-tree 1)}
\item \texttt{(max-tree 2)}
\item \texttt{(max-tree 4)}
\item \texttt{(max-tree 3)}
\item \texttt{(max-tree '(3 . 4))}
\end{itemize}
miniAdapton Interfaces

▶ Adapton thunks ("athunks") and Adapton references ("arefs")
  ▶ adapton-ref
  ▶ adapton-ref-set!
  ▶ adapt
  ▶ adapton-force

▶ Adapton memoization ("amemo")
  ▶ adapton-memoize, adapton-memoize-l
  ▶ define-amemo, define-amemo-l

▶ Adapton variables ("avar")
  ▶ define-avar
  ▶ avar-get
  ▶ avar-set!
miniAdapton Interfaces

- Adapton thunks ("athunks") and Adapton references ("arefs")
  - adapton-ref
  - adapton-ref-set!
  - adapt
  - adapton-force
- Adapton memoization ("amemo")
  - adapton-memoize, adapton-memoize-l
  - define-amemo, define-amemo-l
miniAdapton Interfaces

▶ Adapton thunks ("athunks") and Adapton references ("arefs")
  ▶ adapton-ref
  ▶ adapton-ref-set!
  ▶ adapt
  ▶ adapton-force
▶ Adapton memoization ("amemo")
  ▶ adapton-memoize, adapton-memoize-l
  ▶ define-amemo, define-amemo-l
▶ Adapton variables ("avar")
  ▶ define-avar
  ▶ avar-get
  ▶ avar-set!
Interface Demo

```lisp
> (add-to-load-path "miniAdaption")
```
Interface Demo

```scala
import (minioadapt)```

Interface Demo

```scheme
(import (minilanguage))
(define rl (adapton-ref 1))
```
Interface Demo

```scheme
(import (miniladaption))
(define r1 (adaption-ref 1))
r1
```
Interface Demo

```scheme
(define r1 (adapt-ref 1))
r1
```

```scheme
(import (miniladaptor))
```

```scheme
#:import r6rs:record:adaptation
```
Interface Demo

```
  (import (mini-adopton))
  (define r1 (adopton-ref 1))
  r1
  #r6rs-record:adopton
  (adopton-force r1)
```
Interface Demo

```scheme
(define r1 (adaption-ref 1))
(export r1)
(siloc:memoize r1)
```

```
> (import (miniadaption))
> (define r1 (adaption-ref 1))
> r1
#<r6rs:record:adaption>
> (adaption-force r1)
1
```
(import (mini-adapton))
(define r1 (adapton-ref 1))
r1
#<r6rs:record:adapton>
(adapton-force r1)
(adapton-ref-set! r1 5)
(import (miniladaption))
(define r1 (adaption-ref 1))
r1
(r6rs:record:adaption)
(adaption-force r1)
(adaption-ref-set! r1 5)
(adaption-force r1)
Interface Demo

```lisp
(import (miniladaptor))
(define r1 (adaption-ref 1))
(r1)
;;(r#rs:record:adaption>
(adiaption-force r1)
(adiaption-ref-set! r1 5)
(adiaption-force r1)
```
```
(r Grs:record:adapt:)

(define r1 (adapt-ref 1))
(r1)
(adapt-ref-set! r1 5)
(adapt-force r1)

(define t1 (adapt (+ 2 (adapt-force r1))))
```
(import (minialdaption))
(define r1 (aldaption-ref 1))
r1
#<r6rs:record:aldaption>
(aldaption-force r1)
(adaption-ref-set! r1 5)
(adaption-force r1)

(define t1 (adapt (+ 2 (aldaption-force r1))))
(aldaption-force t1)
Interface Demo

1. (import (minialpation))
2. (define r1 (adapton-ref 1))
3. r1
4. (adapton-force r1)
5. (adapton-ref-set! r1 5)
6. (adapton-force r1)
7. (define t1 (adapton (+ 2 (adapton-force r1))))
8. (adapton-force t1)
(import miniadaption)
(define r1 (adaption-ref 1))
r1
(define t1 (adaption-force r1))
(adaption-ref-set! r1 5)
(adaption-force r1)
(define t1 (adaption-force t1))
(adaption-ref-set! r1 1-5)
Interface Demo

```
(Import (mini-adaptor))
(define r1 (adaptor-ref 1))
r1
(rGrs:record:adaptor>
(adaptor-force r1)
(adaptor-ref-set! r1 5)
(adaptor-force r1)
(define t1 (adapt (+ 2 (adaptor-force r1))))
(adaptor-force t1)
(adaptor-ref-set! r1 -5)
(adaptor-force t1)
```
(import (mini-adapton))
(define r1 (adapton-ref 1))
r1
(r6rs:record.adapton)
(adapton-force r1)
(adapton-ref-set! r1 5)
(adapton-force r1)
(define t1 (adapt (+ 2 (adapton-force r1))))
(adapton-force t1)
(adapton-ref-set! r1 -5)
(adapton-force t1)
3
(import (minigo:adapt))
(define r1 (adapt-ref r1))
r1
(define t1 (adapt-ref-set! r1 5))
(adapt-ref r1)

(define t1 (adapt-ref-set! r1 -5))
(adapt-ref r1)

(define-memo (max-tree t) (cond ((adapt-ref? t) (max-tree (adapt-ref r1))) ((pair? t) (max (max-tree (car t)) (max-tree (cdr t)))) (else t)))
Interface Demo

```lisp
(import (mini-adapton))
(define r1 (adapton-ref 1))
r1
(r6rs:record-adapton)
(adapton-force r1)
(adapton-ref-set! r1 5)
(adapton-force r1)

(define t1 (adapt (+ 2 (adapton-force r1))))
(adapton-force t1)
(adapton-ref-set! r1 -5)
(adapton-force t1)

(define-memo (max-tree t) (cond ((adapton? t) (max-tree (adapton-force t))) ((pair? t) (max (max-tree (car t)) (max-tree (cdr t)))) (else t)))
(define t2 (adapton-ref (cons 4 t1)))
```
(import (minidapton))
(define r1 (adapton-ref 1))
r1
(adapton-force r1)
(adapton-ref-set! r1 5)
(adapton-force r1)
(define t1 (adapt (+ 2 (adapton-force r1))))
(adapton-force t1)
(adapton-ref-set! r1 -5)
(adapton-force t1)
(define-memo (max-tree t1) (cond ((adapton? t) (max-tree (adapton-force t))) ((pair? t) (max (max-tree (car t)) (max-tree (cdr t)))) (else t)))
(define t2 (adapton-ref (cons 4 t1)))
(max-tree t2)
Interface Demo
Interface Demo

```
(adapton-ref-set! t2 100)
```

Interface Demo

20  21  (adapton-ref-set! t2 100)
22  (max-tree t2)
Interface Demo

```scheme
10  (adapt-ref-set! t2 100)
20  (max-tree t2)
23  100
```
Interface Demo

20  (adapton-ref-set! t2 100)
21  (max-tree t2)
22  100
23  (define r2 (adapton-ref (* (adapton-force r1) -1)))
Interface Demo

```
20> (adapton-ref-set! t2 100)
21> (max-tree t2)
22> 100
23> (define r2 (adapton-ref (* (adapton-force r1) -1)))
25> (adapton-ref-set! r1 0)
```
Interface Demo

(adapton-ref-set! t2 100)
(max-tree t2)
100
(define r2 (adapton-ref (* (adapton-force r1) -1)))
(adapton-ref-set! r1 0)
(+ (adapton-force r1) (adapton-force r2))
Interface Demo

(adapton-ref-set! t2 100)
(max-tree t2)
100
(define r2 (adapton-ref (* (adapton-force r1) -1)))
(adapton-ref-set! r1 0)
(+ (adapton-force r1) (adapton-force r2))
Interface Demo

```
(adapton-ref-set! t2 100)
(mag-tree t2)
100
(define r2 (adapton-ref (* (adapton-force r1) -1)))
(mag-tree-ref-set! r1 0)
(+ (adapton-force r1) (adapton-force r2))
(define v1 (cons 1 2))
```
Interface Demo

```
(define-ref-set! t2 100)
(define-tree t2)
100
(define r2 (define-ref (* (define-force r1) -1)))
(define-ref-set! r1 0)
(* (define-force r1) (define-force r2))
(define:avar v1 (cons 1 2))
(define:avar v2 (cons 3 4))
```
(define r2 (adapton-ref (* (adapton-force r1) -1)))
(define v1 (cons 1 2))
(define v2 (cons 3 4))
(define v3 (cons (avar-get v1) (avar-get v2)))
Interface Demo

```scheme
20> (adapton-ref-set! t2 100)
21> (max-tree t2)
22> 100
23> (define r2 (adapton-ref (* (adapton-force r1) -1)))
24> (adapton-ref-set! r1 0)
25> (* (adapton-force r1) (adapton-force r2))
26> 5
27> (define-avar v1 (cons 1 2))
28> (define-avar v2 (cons 3 4))
29> (define-avar v3 (cons (avar-get v1) (avar-get v2)))
30> v3
```
Interface Demo

```
20 4 (adapton-ref-set! t2 100)
21 20 (max-tree t2)
25 5 (define r2 (adapton-ref (* (adapton-force r1) -1)))
26 5 (adapton-ref-set! r1 0)
27 5 (* (adapton-force r1) (adapton-force r2))
29 5 (define:avar v1 (cons 1 2))
29 5 (define:avar v2 (cons 3 4))
30 5 (define:avar v3 (cons (avar-get v1) (avar-get v2)))
31 5 v3
32 5 (srs:record:adapton)
```
(adapt-on-ref-set! t2 100)
(max-tree t2)

(define r2 (adapt-on-ref (* (adapt-on-force r1) -1)))
(adapt-on-ref-set! r1 0)
(+ (adapt-on-force r1) (adapt-on-force r2))

(define-avar v1 (cons 1 2))
(define-avar v2 (cons 3 4))
(define-avar v3 (cons (avar-get v1) (avar-get v2)))

(v3)
(<r6rs:record>adapt-on>)
(avar-get v3)
Interface Demo

```
20  (adapt-on-ref-set! t2 100)
21  (max-tree t2)
22  100
23  1
24  (define r2 (adapt-on-ref (* (adapt-on-force r1) -1)))
25  (adapt-on-ref-set! r1 0)
26  (* (adapt-on-force r1) (adapt-on-force r2))
27  5
28  (define v1 (cons 1 2))
29  (define v2 (cons 3 4))
30  (define v3 (cons (vivar-get v1) (vivar-get v2)))
31  v3
32  #<vfs:record adapt-on>
33  (vivar-get v3)
34  ((1 . 2) 3 . 4)
```
(adapton-ref-set! t2 100)

100

(define r2 (adapton-ref (* (adapton-force r1) -1)))

(define-r1 (adapton-ref s1 0)

(+ (adapton-force r1) (adapton-force r2))

(define v1 (cons 1 2))

(define v2 (cons 3 4))

(define v3 (cons (avar-get v1) (avar-get v2)))

(v3)

(define-record adapton

(avar-get v3)

((1 2) 3 4)

(max-tree v3)
Interface Demo

```scheme
(adaptive-ref-set! t2 100)
(max-tree t2)
100
(define r2 (adaptive-ref (* (adaptive-force r1) -1)))
(adaptive-ref-set! r1 0)
(* (adaptive-force r1) (adaptive-force r2))
(define v1 (cons 1 2))
(define v2 (cons 3 4))
(define v3 (cons (avar-get v1) (avar-get v2)))
v3
#<Gc:record adaptive>
(avar-get v3)
((1 2) (3 4))
(max-tree v3)
```
(adapton-ref-set! t2 100)
(max-tree t2)
100
(define r2 (adapton-ref (* (adapton-force r1) -1)))
(adapton-ref-set! r1 0)
(+ (adapton-force r1) (adapton-force r2))
(define v1 (cons 1 2))
(define v2 (cons 3 4))
(define v3 (cons (avar-get v1) (avar-get v2)))
(avar-get v3)
(avar-set! v2 5)
 Interface Demo

```
; (adapt-on-ref-set! t2 100)
; (max-tree t2)
; 100
; (define r2 (adapt-on-ref (* (adapt-on-force r1) -1)))
; (adapt-on-ref-set! r1 0)
; (* (adapt-on-force r1) (adapt-on-force r2))
; v3
; (define-avar v1 (cons 1 2))
; (define-avar v2 (cons 3 4))
; (define-avar v3 (cons (avar-get v1) (avar-get v2)))
; v3
; (avar-get v3)
; ((1 . 2) 3 . 4)
; (max-tree v3)
; (avar-set! v2 5)
; (max-tree v3)
```
Interface Demo

```
20  (adapton-ref-set! t2 100)
21  (max-tree t2)
22  100
23  (define r2 (adapton-ref (adapton-force r1) -1)))
24  (adapton-ref-set! r1 0)
25  (* (adapton-force r1) (adapton-force r2))
26  v3
27  (define-avar v1 (cons 1 2))
28  (define-avar v2 (cons 3 4))
29  (define-avar v3 (cons (avar-get v1) (avar-get v2)))
30  (avar-get v3)
31  (avar-set! v2 5)
32  (max-tree v3)
```
microAdapton: the core of miniAdapton

- inspired by microKanren for implementing miniKanren
- implements core operations for miniAdapton
- avoids implicit DCG construction
- miniAdapton builds implicit DCG construction on top of microAdapton
Implementation - microAdapton

```
(define-record-type
  (adaption adaption-cons adaption?)
  (fields
   thunk
   (mutable result)
   (mutable sub)
   (mutable super)
   (mutable clean?)))

(make-a-thunk thunk)
(adaption-cons thunk
  "empty"
  empty-set
  empty-set
  $))

(define (adaption-odd-deg-edge! a-super a-sub)
  (adaption-sub-set! a-super
    (set-cons a-sub (adaption-sub a-super)))
  (adaption-super-set! a-sub
    (set-cons a-super (adaption-super a-sub)))))

(define (adaption-del-deg-edge! a-super a-sub)
  (adaption-super-set! a-super
    (set-rem a-super (adaption-sub a-super)))
  (adaption-sub-set! a-sub
    (set-rem a-sub (adaption-super a-sub)))))

(define (adaption-compute a)
  (if (adaption-clean? a)
      (adaption-result a)
      (begin
        (set-for-each
          (lambda (x)
            (adaption-del-deg-edge! a x))
          (adaption-sub a))
        (adaption-super-set! a #t)
        (adaption-result-set! a))
      record-type)

(Lisp Interaction Prelude)
```

Beginning of buffer
Implementation - microAdapton

```
(adaptive-clean? set! a #t)
(adaptive-result-set! a)
((adaptive-thunk a)))
(adaptive-compute a))))

(define (adaptive-dirty! a)
  (when (adaptive-clean? a)
    (adaptive-clean?-set! a #f)
    (set-for-each adaptive-dirty!
      (adaptive-super a))))

(define (adaptive-ref-val)
  (letrec ((a adaptive-cons
    (lambda () (adaptive-result a))
    val
    empty-set
    empty-set
    #t))
    a))

(define (adaptive-ref-set! a val)
  (adaptive-result-set! a val)
  (adaptive-dirty! a))
```
Implementation - miniAdapton

```
(defineadapton-force
  (let((currently-adapting #f))
    (lambda(a)
      (let((prev-adapting
        currently-adapting)
        (set! currently-adapting a)
        (let((result (adapton-compute a)))
          (set! currently-adapting
              prev-adapting)
          (when currently-adapting
            (adapton-add-dcg-edge
currently-adapting
  a))
          result))))))

(define-syntax adopt
  (syntax-rules ()
    ([_ expr]
      (make-ast-node (lambda () expr)))))

(define (adapton-memoize-1 f)
  (memoize (lambda x (adapt (apply f x))))))

(define (adapton-memoize f)
  (let ((f* (adapton-memoize-1 f)))
    (lambda x (adapton-force (apply f* x))))))

(define-syntax lambda-memo-1
  (syntax-rules ()
    ([_ (args ...) body ...]
      (let ((f* (adapton-memoize-1
        (lambda (args ...)
          body ...))))
        (lambda (args ...) (f* args ...))))))

(define-syntax lambda-memo
  (syntax-rules ()
    ([_ (args ...) body ...]
      (lambda-memo-1...)))
```

Implementation - miniAdapton

```scheme
(define-syntax define-memo
  (syntax-rules ()
    ((__ (args ...) body ...)
     (let ((#* (adapt-memoize)
                 (lambda (args ...)
                 body ...))))
     (lambda (args ...) (#* args ...))))

(define-syntax define-memo-1
  (syntax-rules ()
    ((__ (args ...) body ...)
     (define f (lambda-memo-1 (args ...)
                        body ...))))

(define-syntax define-memo
  (syntax-rules ()
    ((__ (args ...) body ...)
     (define f (lambda-memo (args ...)
                           body ...))))

(define-syntax define-avar
  (syntax-rules ()
    ((__ name expr)
     (define name
      (adapt-ref (adapt expr))))

(define (avar-get v)
  (adapt-force (adapt-force v)))

(define-syntax avar-set!
  (syntax-rules ()
    ((__ v expr)
     (adapt-ref-set! v (adapt expr))))
```

Conclusion

- Adapton implemented in a more minimal form
- A minimal implementation encourages hackability
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
- We want you to use this as soon as possible
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
- We want you to use this as soon as possible
- Play with this
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
- We want you to use this as soon as possible
- Play with this
- This toy we made is neat and everyone should play with it:
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
- We want you to use this as soon as possible
- Play with this
- This toy we made is neat and everyone should play with it:
- `git clone 'https://github.com/fisherdj/miniAdapton'`
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
- We want you to use this as soon as possible
- Play with this
- This toy we made is neat and everyone should play with it:
  - git clone 'https://github.com/fisherdj/miniAdapton'
  - ...

...
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
- We want you to use this as soon as possible
- Play with this
- This toy we made is neat and everyone should play with it:
  - git clone 'https://github.com/fisherdj/miniAdapton'
  - ...
  - git clone 'https://github.com/fisherdj/miniAdapton'
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
- We want you to use this as soon as possible
- Play with this
- This toy we made is neat and everyone should play with it:
  - git clone 'https://github.com/fisherdj/miniAdapton'
  - ...
  - git clone 'https://github.com/fisherdj/miniAdapton'
- More details about the code are in the paper
Conclusion - Play with it

- Incremental computation that you can play with RIGHT NOW
- We want you to use this as soon as possible
- Play with this
- This toy we made is neat and everyone should play with it:
  - git clone
    'https://github.com/fisherdj/miniAdapton'
  - ...
  - git clone
    'https://github.com/fisherdj/miniAdapton'
- More details about the code are in the paper
Challenges

Modifying miniAdapton:
- Avoid recomputation when answers to subcomputations don’t change (full Adapton)
- Add debugging information and/or visualization
- miniAdapton in other languages

Using miniAdapton:
- Adapton data structures
- Adapton for interactive applications
Acknowledgements and Related Work

- Thanks to Jason Hemann and Dan Friedman for microKanren, a huge inspiration and motivation for miniAdapton
- Incremental Computing via Function Caching, Pugh and Teitelbaum POPL 1986 (still a good inspiration for data structures using Adapton)
- The Adapton Project, Hammer et al OOPSLA 2015 and PLDI 2014 (http://adapton.org)
- Self-Adjusting Computation, Acar et al; (http://www.umut-acar.org/self-adjusting-computation)