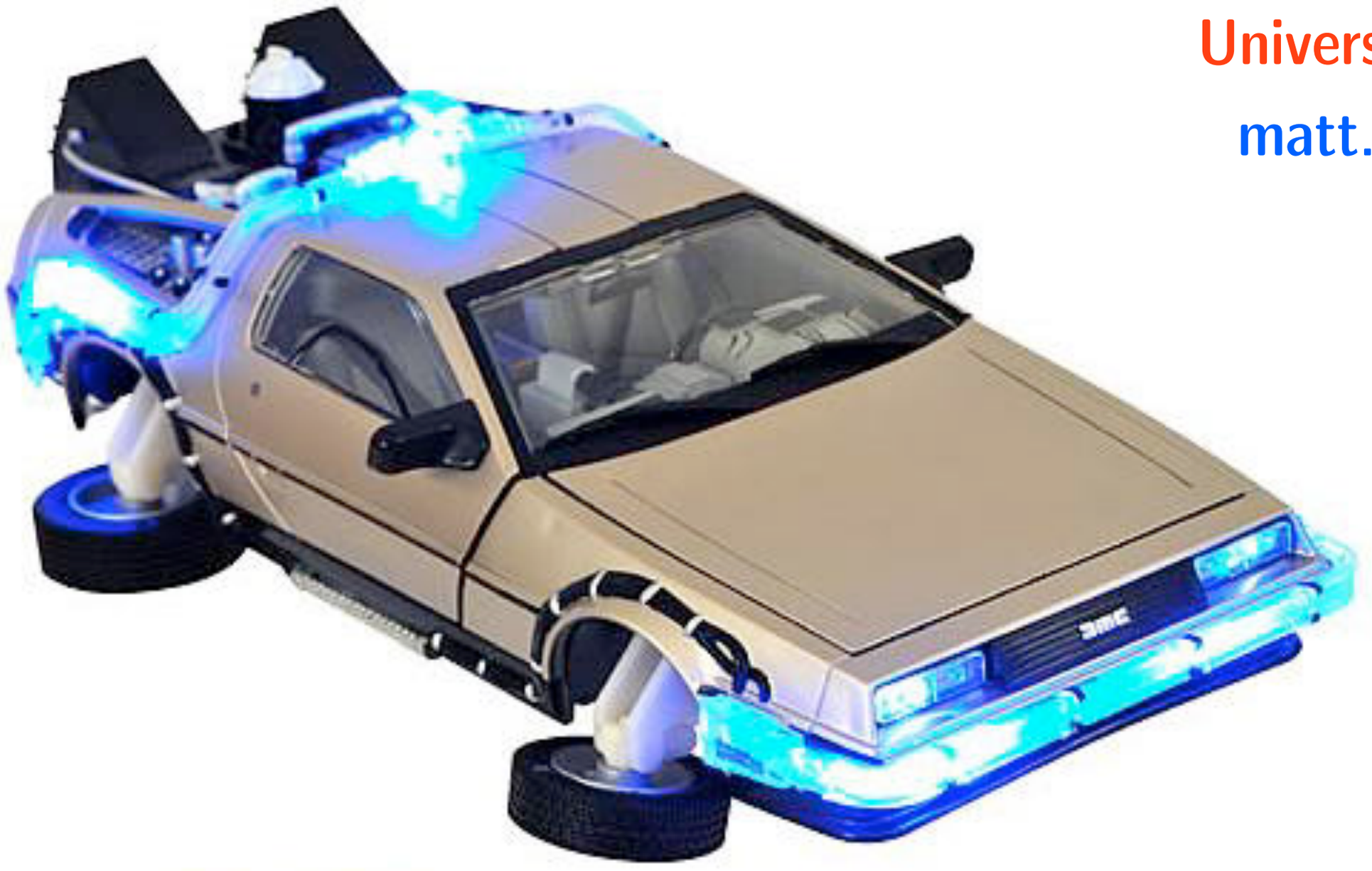
Matt Might
University of Utah
matt.might.net



Continuations

Why continuations?

- Back-tracking search
- Model exceptions
- Cooperative threads
- Preemptive threads
- Generators
- Coroutine systems
- Time-travel

What are continuations?

Continuations

A continuation is like a saved game.

Continuations

They're like time travel.

Continuations

They're like `go-when` instead of `goto`.

Continuations

The current continuation is “the rest of the computation.”

Continuations

The program stack encodes the current continuation.

Continuations

The state of a thread is a continuation.

Continuations

A continuation is a procedure
that never returns to its caller.

Continuations

Exceptions are a special case of continuations.

Continuations

A continuation is a first-class encoding of control.

Heap

Registers

Stack

Heap

Registers

Stack

Registers

Stack

Heap

Registers

Stack

Now, some magic

go-when **and** right-now

```
; An infinite loop:
(let ((the-beginning (right-now)))
  (display "Hello, world!")
  (newline)
  (go-when the-beginning))
```

The amb "function"

```
; The following prints (3 4 5)
(let ((a (amb (list 1 2 3 4 5 6 7)))
      (b (amb (list 1 2 3 4 5 6 7)))
      (c (amb (list 1 2 3 4 5 6 7))))
  (assert (= (+ (* a a) (* b b)) (* c c)))
  (display (list a b c))
  (newline))
```

SAT-solving

```
; The following evaluates to (#f #f #t)

(sat-solve (and (implies a (not b))
                (not a)
                c)
           (list a b c))
```

Cooperative threads

```
(spawn (make-thread-thunk 'a))
(spawn (make-thread-thunk 'b))
(spawn (make-thread-thunk 'c))

(start-threads)
```

```
(define counter 10)

(define (make-thread-thunk name)
  (letrec ((loop (lambda ()
                   (if (< counter 0)
                       (quit))
                   (display "in thread ")
                   (display name)
                   (display "; counter = ")
                   (display counter)
                   (newline)
                   (set! counter (- counter 1))
                   (yield)
                   (loop))))
    loop))
```

Generators

```
(for v in (tree-iterator '(3 . ((4 . 5) . 6 )))
  (display v)
  (newline))
```

Generators

```
(define (tree-iterator tree)
  (lambda (yield)

    (define (walk tree)
      (if (not (pair? tree))
          (yield tree)
          (begin
            (walk (car tree))
            (walk (cdr tree)))))

    (walk tree)))
```

call-with-current-continuation

call/cc

```
(call/cc (lambda (cc) …))
```

current-continuation

```
(define (current-continuation)
  (call/cc (lambda (cc) (cc cc))))
```

Design pattern

```
(let ((cc (current-continuation)))
  (cond
   ((procedure? cc) ...)
   ((future-val? cc) ...)
   (else (error “contract broken!”))))
```

Escape pattern

```
(lambda (...)
 (call/cc (lambda (return)
   ...)))
```

Escape pattern

```
(call/cc (lambda (break)
  (for ...)))
```

call/ec

Details