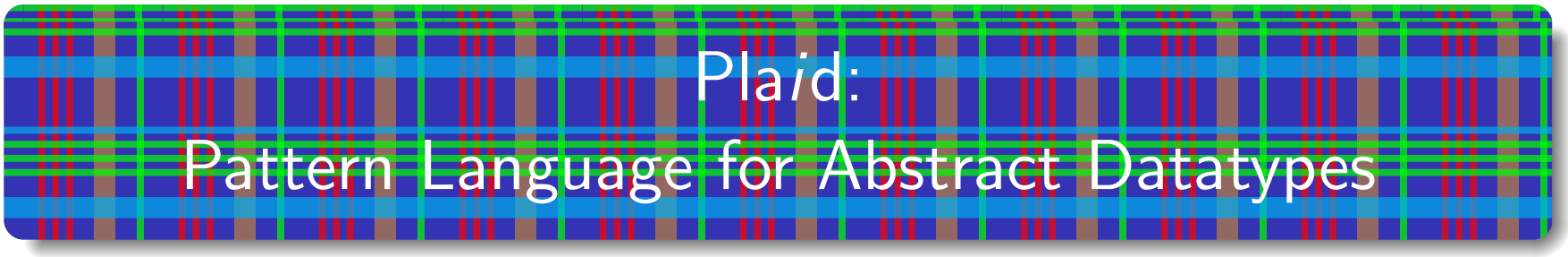


Austin Clements Irene Zhang Dan Ports

Friday, May 11, 2007

Syntax Patterns and ADT's

```
(define (make-computer model os)
  (list '*computer* os model))
```

Syntax Patterns and ADT's

```
(define (make-computer model os)
  (list '*computer* os model))
```

```
(define (crash-computer comp)
  (cond ((match comp '(*computer* windows _))
         'done)
        ((match comp '(*computer* linux _))
         'failed)))
```

Syntax Patterns and ADT's

```
(define (make-computer model os)
  (list '*computer* os model))
```

```
(define (crash-computer comp)
  (cond ((match comp '(*computer* windows _))
         'done)
        ((match comp '(*computer* linux _))
         'failed)))
```

Syntax Patterns and ADT's

```
(define (make-computer model os)
  (list '*computer* os model))
```

```
(define (crash-computer comp)
  (cond ((match comp '(*computer* windows _))
         'done)
        ((match comp '(*computer* linux _))
         'failed)))
```

Syntax patterns are limited to built-in types.

Syntax Patterns and ADT's

```
(define (make-computer model os)
  (list '*computer* os model))
```

```
(define (crash-computer comp)
  (cond ((match comp '(make-computer _ 'windows))
         'done)
        ((match comp '(make-computer _ 'linux))
         'failed)))
```

Syntax patterns are limited to built-in types.

Semantic Patterns and ADT's

```
(define (make-computer model os)
  (list '*computer* os model))
```

```
(define (computer? datum)
  (and (pair? datum)
       (eq? (car datum) '*computer*)))
(define computer-model third)
(define computer-os second)
```

```
(define (match:make-computer? pattern)
  (and (pair? pattern)
       (eq? (car pattern) 'make-computer)))
(define match:make-computer-model second)
(define match:make-computer-os third)
```

```
(define (match:make-computer model-comb os-comb)
  (define (match data dictionary succeed)
    (and (computer? data)
         (model-comb
          (computer-model data) dictionary
          (lambda (new-dict n)
            (os-comb
             (computer-os data) new-dict
             succeed)))))
  match)
```

```
(define (compile-match-make-computer pat use-env loop)
  `(match:make-computer ,(loop (match:make-computer-model pat))
                        ,(loop (match:make-computer-os pat))))
(eq-put! 'make-computer 'pattern-keyword compile-match-make-computer)
```

(match comp '(make−computer _ 'windows))

We want to be able to write this

```
(define (computer? datum)
  (and (pair? datum)
       (eq? (car datum) '*computer*)))
(define computer-model third)
(define computer-os second)

(define (match:make-computer model-comb os-comb)
  (define (match data dictionary succeed)
    (and (computer? data)
         (model-comb
          (computer-model data) dictionary
          (lambda (new-dict n)
            (os-comb
             (computer-os data) new-dict
             succeed)))))
  match)

(define (compile-match-make-computer pat use-env loop)
  `(match:make-computer ,(loop (match:make-computer-model pat))
                        ,(loop (match:make-computer-os pat))))
(eq-put! 'make-computer 'pattern-keyword compile-match-make-computer)
```

```
(define (match:make-computer? pattern)
  (and (pair? pattern)
       (eq? (car pattern) 'make-computer)))
(define match:make-computer-model second)
(define match:make-computer-os third)
```

Without having to write this

Constructor-Oriented Pattern Syntax

Data-oriented

Resembles data

(1 (? x))

Constructor-oriented

Resembles code

(cons 1 (cons x ()))

Constructor-Oriented Pattern Syntax

Data-oriented

Resembles data

```
(1 . ((? x) . ()))
```

Constructor-oriented

Resembles code

```
(cons 1 (cons x ()))
```

Constructor-Oriented Pattern Syntax

Data-oriented

Resembles data

(1 . ((? x) . ()))

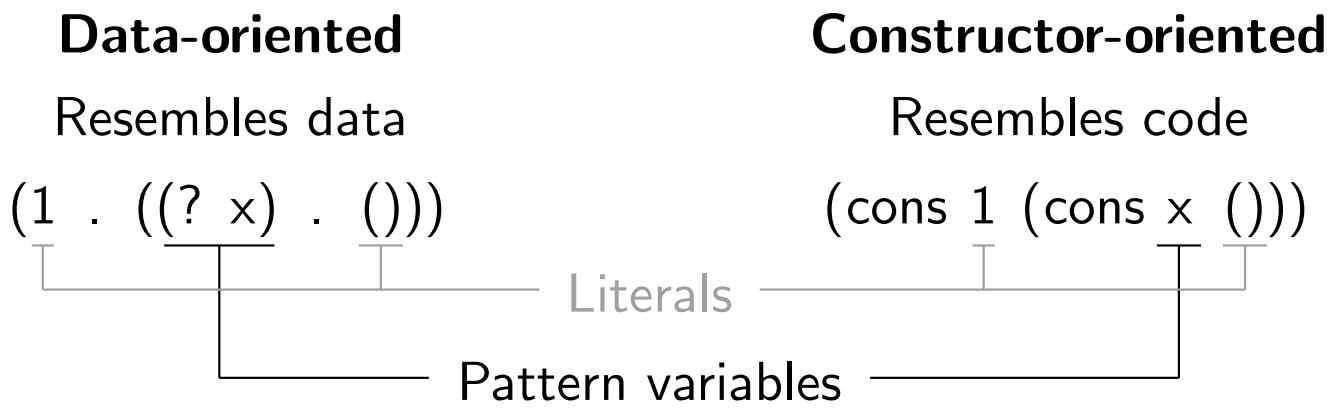
Constructor-oriented

Resembles code

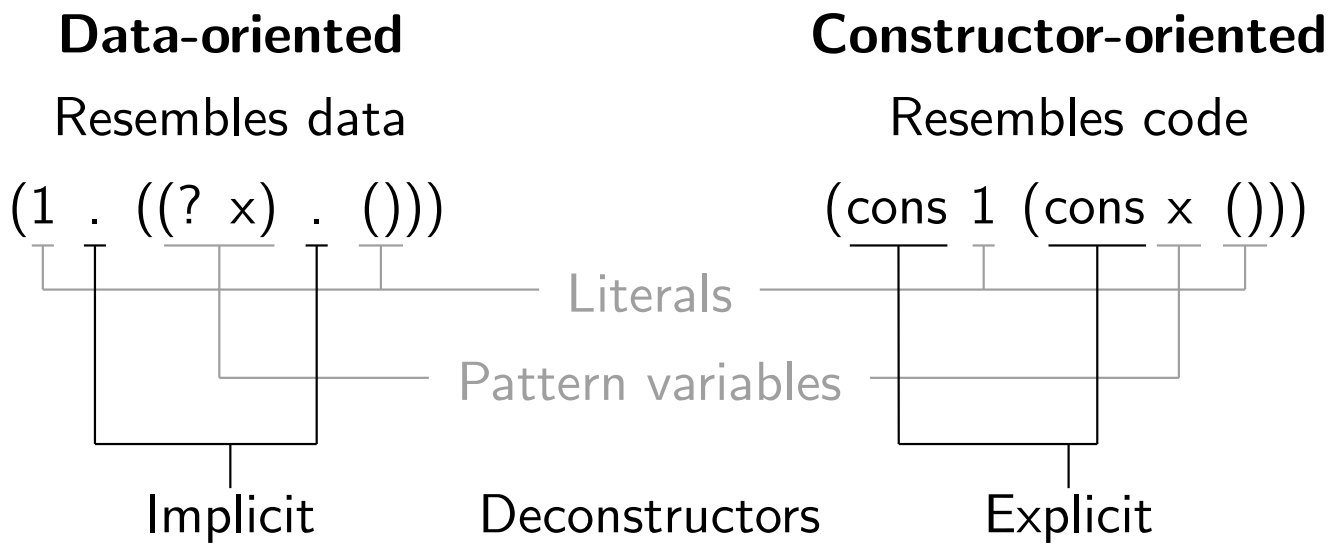
(cons 1 (cons x ()))

Literals

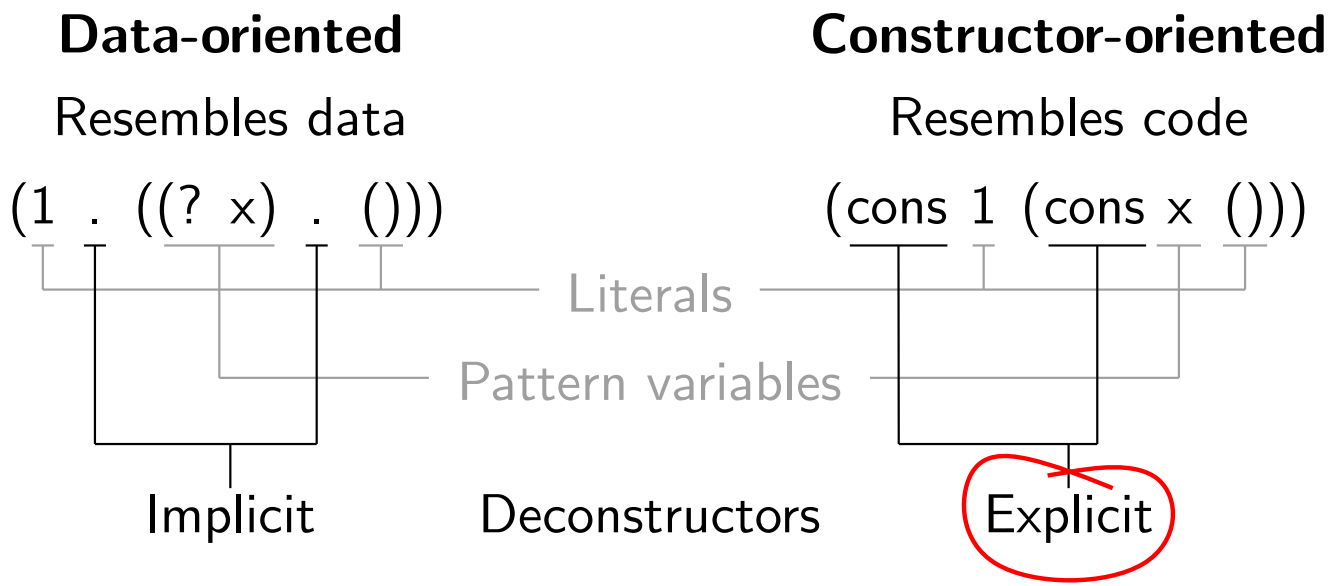
Constructor-Oriented Pattern Syntax



Constructor-Oriented Pattern Syntax



Constructor-Oriented Pattern Syntax



Abstraction, Abstraction, Abstraction

```
(define (make-computer model os)
  (list '*computer* os model))
```

Abstraction, Abstraction, Abstraction

```
(define (make-computer model os)
  (list '*computer* os model))
```

Pattern	Datum
(make-computer _ 'linux)	(*computer* linux pc)

Abstraction, Abstraction, Abstraction

```
(define (make-computer model os)
  (list '*computer* os model))
```

Pattern	Datum
(make-computer _ 'linux)	(*computer* linux pc)
(list '*computer* 'linux _)	(*computer* linux pc)

Abstraction, Abstraction, Abstraction

```
(define (make-computer model os)
  (list '*computer* os model))
```

Pattern	Datum
(make-computer _ 'linux)	(*computer* linux pc)
(list '*computer* 'linux _)	(*computer* linux pc)
(cons '*computer* (list ' linux _))	(*computer* linux pc)

Abstraction, Abstraction, Abstraction

```
(define (make-computer model os)
  (list '*computer* os model))
```

Pattern	Datum
(make-computer _ 'linux)	(*computer* linux pc)
(list '*computer* 'linux _)	(*computer* linux pc)
(cons '*computer* (list 'linux _))	(*computer* linux pc)
(list 'linux _)	(linux pc)

Abstraction, Abstraction, Abstraction

```
(define (make-computer model os)
  (list '*computer* os model))
```

Pattern	Datum
(make-computer _ 'linux)	(*computer* linux pc)
(list '*computer* 'linux _)	(*computer* linux pc)
(cons '*computer* (list 'linux _))	(*computer* linux pc)
(list 'linux _)	(linux pc)
(cons 'linux (list _))	(linux pc)
(list _)	(pc)
(cons _ (list))	(pc)
(list)	()
()	()

Success

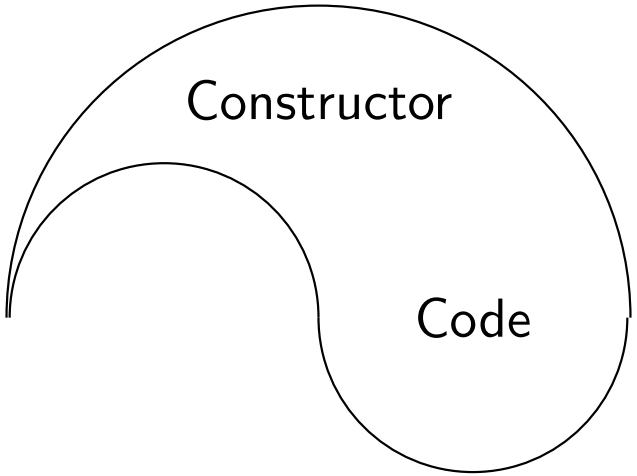
Plambda the Ultimate Deconstructor

```
(define make-computer
  (plambda (model os)
    (list '*computer* os model)))
```

Bridges Scheme abstraction and pattern abstraction

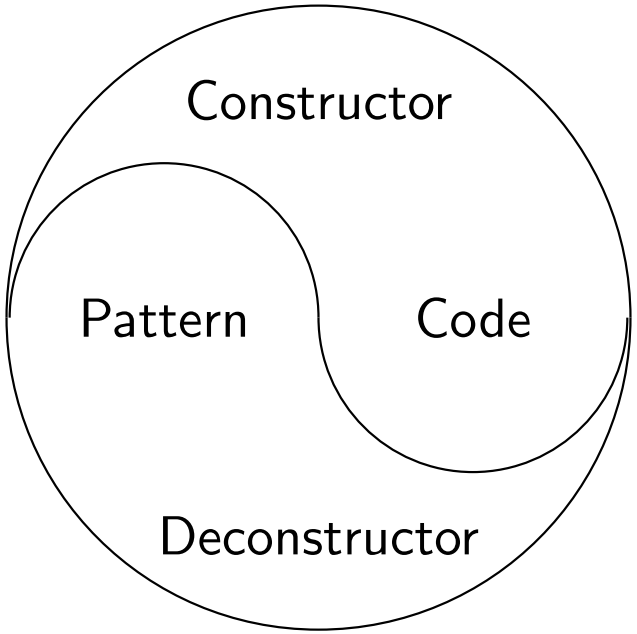
Plambda the Ultimate Deconstructor

```
(define make-computer
  (plambda (model os)
    (list '*computer* os model)))
```



Plambda the Ultimate Deconstructor

```
(define make-computer
  (plambda (model os)
    (list '*computer* os model)))
```



Syntactic Sugar

```
(define-constructor (make-computer model os)
  (list '*computer* os model))
```

desugars into

```
(define make-computer
  (plambda (model os)
    (list 'computer os model)))
```

Can often simply change **define** to **define-constructor** in existing code!

Pattern Matching in Scheme

```
(define (crash-computer comp)
  (pcase comp
    ((make-computer _ 'windows) 'done)
    ((make-computer _ 'linux) 'failed)))
```

pcase combines pattern matching, failure

Pattern Matching in Scheme

```
(define (crash-computer comp)
  (pcase comp
    ((make-computer model 'windows)
     (if (expensive? model)
         'this-is-not-the-crash-you-are-looking-for
         'done))
    ((make-computer _ 'linux) 'failed)))
```

pcase combines pattern matching, failure, and binding

Pro-Choice Patterns

pcase allows us to add *decision* to the pattern language

Pro-Choice Patterns

pcase allows us to add *decision* to the pattern language

```
(define-constructor (append a b)
  (pcase a
    (() b)
    ((cons this rest) (cons this (append rest b)))))
```

Pro-Choice Patterns

pcase allows us to add *decision* to the pattern language

```
(define-constructor (append a b)
  (pcase a
    (() b)
    ((cons this rest) (cons this (append rest b)))))

(pcase '(1 2 3 4)
  ((append x (list 3 4)) (pp x)))
```

$\Longrightarrow$ x $\leftarrow$ (1 2)

Constructors in Plaid

```
(define-constructor (make-computer model os)
  (list '*computer* os model))

(define bills-computer (make-computer 'desktop 'windows))

(define (crash-computer comp)
  (pcase comp
    ((make-computer _ 'windows) 'done)
    ((make-computer _ 'linux) 'failed)))
```

(crash-computer bills-computer) $\Longrightarrow$ done

With the power of Plaid, patterns are no longer limited to built-in types.

Deconstructors in Plaid

Pattern matching can be used to pick data out of an abstract datatype...

```
(define (computer:os comp)
  (pcase comp
    ((make-computer _ os) os)
    (_ (error ``Object must be of type computer''))))

(define (computer:model comp)
  (pcase comp
    ((make-computer model _) model)
    (_ (error ``Object must be of type computer''))))
```

Deconstructors in Plaid

... or to imagine different views of a data structure.

```
(define-constructor (lambda* args body)
  (cons 'lambda (cons args body)))

(define-constructor (define* name expr)
  (pcase expr
    ((lambda* args . body)
     (cons 'define (cons (cons name args) body)))
    (_ (list 'define name expr))))
```

Imagination in Plaid

```
(define-constructor (** x y)
  (pcase x
    (1 y)
    (_ (pcase y
         (1 x)
         (_ (list '* x y))))))

(pcase '(* 2 3)
  ((** a b) (pp (list a b))
            (if (or (eqv? a 1) (eqv? b 1))
                (next))))
```

Pattern Compiler

Pattern syntax is represented by an abstract structure

A compiler translates the syntax

```
(cons 1 (cons a ()))
```

to

```
(pattern:deconstructor
 (lambda ()
   (constructor->deconstructor cons))
 (list
  (pattern:literal 1)
  (pattern:deconstructor
   (lambda ()
     (constructor->deconstructor cons))
   (list (pattern:variable a)
         (pattern:literal ())))))
```

Pattern Compiler

```
(define (compile-pattern expr env)
  (cond ((nullary? expr)
         `(pattern:literal ,expr))
        ((quoted? expr)
         `(pattern:literal ',(cadr expr)))
        ((variable? expr)
         `(pattern:variable ',expr))
        ((wildcard? expr)
         `(pattern:wildcard))
        ((application? expr)
         `(pattern:deconstructor
           (lambda ()
             (constructor->deconstructor
              ,(close-syntax (car expr) env)))
           (list . ,(map (lambda (arg) (compile-pattern arg env))
                         (cdr expr)))))
        ((pcase? expr)
         `(pattern:pcase
           ,(compile-pattern (cadr expr) env)
           (list ,@(map (lambda (clause)
                          `(pattern:pcase:clause
                            ,(compile-pattern (car clause) env)
                            ,(compile-pattern (cadr clause) env)))
                        (cddr expr)))))
        (else
         (error:invalid-pattern-syntax "pattern" expr))))
```

Constructorification

We can do the same for data...

The constructorifier picks apart pairs, converting the data

```
'(1 a)
```

to

```
(pattern:deconstructor
 (lambda ()
   (constructor->deconstructor cons))
 (list
  (pattern:literal 1)
  (pattern:deconstructor
   (lambda ()
     (constructor->deconstructor cons))
   (list (pattern:variable a)
         (pattern:literal ())))))
```

Patterns and data have the same representation!

plambda

- **plambda** defines both a constructor and a deconstructor
- **plambda** body is a pattern $\Longrightarrow$ *also* valid Scheme code
- Returns a real Scheme procedure, and
  tags the procedure with a deconstructor in a eq hash table
- Lexical scoping of deconstructor bindings is automatic

Matching is Unification

- This pattern matching actually requires *unification*: generalization of matching with variables on *both* sides
- Takes two patterns and initial set of bindings ("environment"); returns new bindings necessary for both sides to match

```
(unify (list 1 2 b) (list 1 a 3) empty-environment)
```

$\Longrightarrow$ a $\leftarrow$ 2; b $\leftarrow$ 3

Basic Unification

```
(define (unify p1 p2 env)
  (cond ((both? pattern:literal?)
         (if (equal? (pattern:literal/value p1)
                     (pattern:literal/value p2))
             env
             ((unifier-environment/fail env))))
        ((either? pattern:wildcard?) env)
        ((either? pattern:variable?)
         => (w/l (lambda (var other)
                   (extend-if-possible var other env))))
        ((either? pattern:literal?)
         ((unifier-environment/fail env)))
        (else (error "Don't know how to unify" p1 p2))))
```

Unifying with Fundamental Deconstructors

- Two fundamental deconstructors unify if they have the same type, and their arguments are unifiable

```
(define (unify-two-fundamental-deconstructors
         p1 p1-dtor p2 p2-dtor env)
  (and (eq? (deconstructor:fundamental/symbol p1-dtor)
            (deconstructor:fundamental/symbol p2-dtor))
       (= (length (pattern:deconstructor/args p1))
          (length (pattern:deconstructor/args p2)))
       (let lp ((args1 (pattern:deconstructor/args p1))
                (args2 (pattern:deconstructor/args p2))
                (env env))
         (if (null? args1)
             env
             (lp (cdr args1) (cdr args2)
                 (unify (car args1) (car args2) env))))))
```

Unifying with Compound Deconstructors

Algorithm

- First, unify the deconstructor body with the other pattern
- Then, unify the formal arguments with their values
- Need alpha-renaming to ensure scoping

Unifying with Compound Deconstructors

```
(define (make-computer model os)
  (cons '*computer* (cons os (cons model ()))))

(unify (make-computer x 'linux)
       (cons '*computer* (cons 'linux (cons 'pc ()))))

(unify (cons '*computer* (cons os (cons model ())))
       (cons '*computer* (cons 'linux (cons 'pc ()))))
```

$\Longrightarrow$ os $\leftarrow$ linux; model $\leftarrow$ pc

```
  (unify model x)
```

$\Longrightarrow$ x $\leftarrow$ pc

```
(unify os 'linux)
```

Unifying with Compound Deconstructors

```
(define (unify-compound-deconstructor var dtor other env)
  (receive (rename-table renamed-body)
      (alpha-rename-pattern
       (deconstructor:compound/body dtor)
       (improper-list-to-proper-list
        (deconstructor:compound/arguments dtor)))
    (let ((env (unify renamed-body other env)))
      (let lp ((env env)
               (formals (deconstructor:compound/arguments dtor))
               (args (pattern:deconstructor/args var)))
        (cond ((and (null? formals) (null? args)) env)
              ((null? formals) (error "Wrong number of arguments"))
              ((null? args) (error "Wrong number of arguments"))
              ((pair? formals) (lp (unify (car args)
                                          (pattern:variable
                                           (cdr (assq (car formals)
                                                      rename-table)))
                                          env)
                                   (cdr formals) (cdr args)))
              (else (unify (pattern:variable
                            (cdr (assq formals rename-table)))
                           (list-to-pattern-list args)
                           env)))))))
```

Unification and the Power of Choice

- **pcase** introduces nondeterminism — multiple possible results:

```
(unify x (pcase 'foo
                ('foo 1)
                (_ 2)))
```

  $\Longrightarrow$ x can be either 1 or 2

- Return failure continuation to try next possibility if unification fails down the line
- Environments now contain bindings and a failure continuation

Unification becomes a search problem!

Unification and the Power of Choice

Algorithm

- Try first clause
  - Install new failure continuation in environment
  - Unify consequent with argument
  - Unify condition with key
- If failure, try next clause

Unification and the Power of Choice

```
(define (unify-pcase var other env)
  (let ((var (alpha-rename-pcase var)))
    (let lp ((clauses (pattern:pcase/clauses var)))
      (if (null? clauses)
          ((unifier-environment/fail env))
          (let ((env1 (call-with-current-continuation
                       (lambda (k)
                         (unifier-environment/new-failure-continuation
                          env (lambda () (k #f)))))))
            (if (false? env1)
                (lp (cdr clauses))
                (let* ((clause (car clauses))
                       (env2 (unify
                              (pattern:pcase:clause/consequent
                               clause)
                              other env1))
                       (env3 (unify
                              (pattern:pcase:clause/condition
                               clause)
                              (pattern:pcase/variable var)
                              env2)))
                  env3)))))))
```