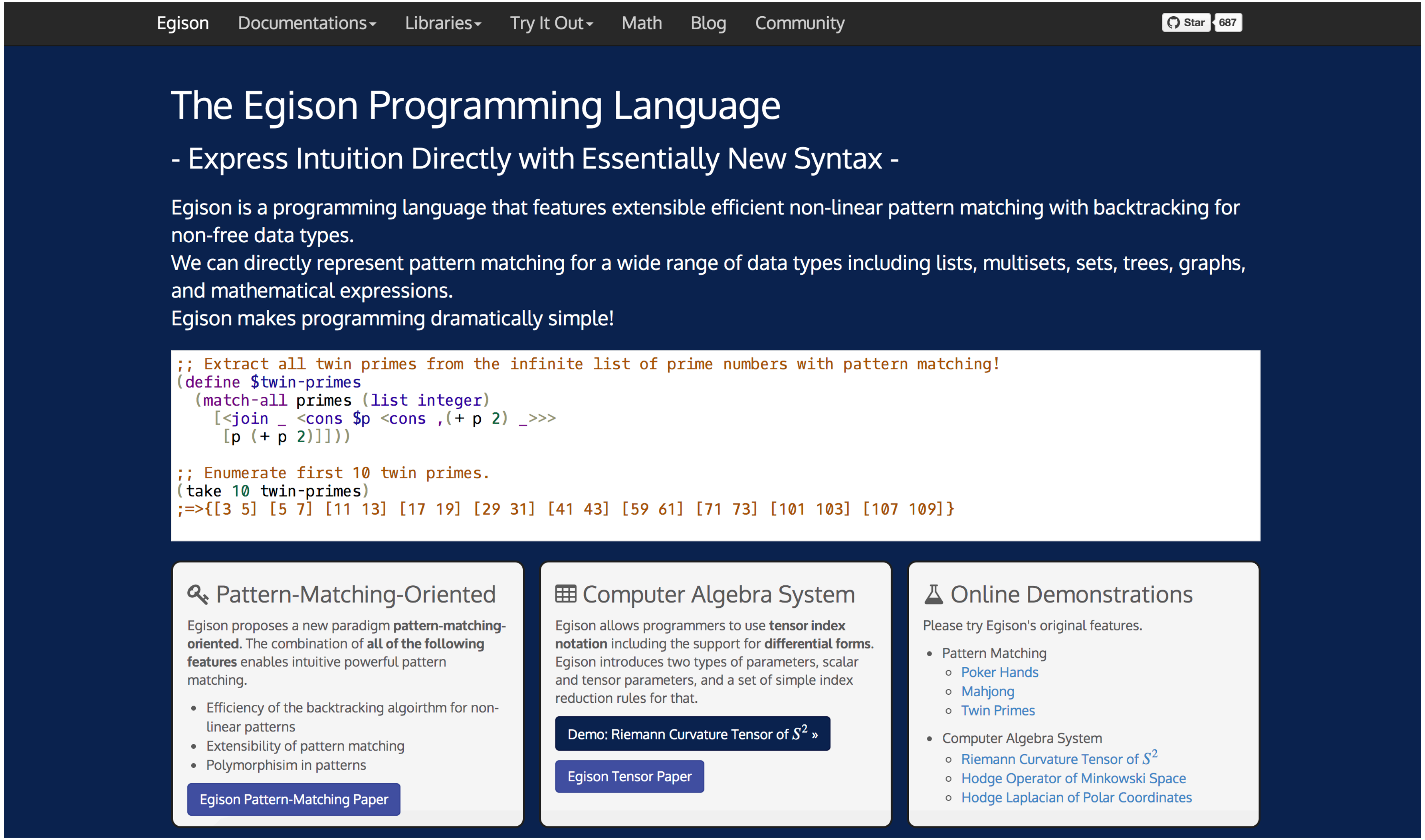
Haskellメタプログラミングによるで Egisonのパターンマッチの実装
— Meta-Programming in Haskell for Non-linear Pattern Matching with Backtracking for Non-free Data Types

Nov 9, 2019

Satoshi Egi
Rakuten Institute of Technology
Rakuten, Inc.

Egison — a functional language that features expressive pattern matching



The Egison Programming Language

- Express Intuition Directly with Essentially New Syntax -

Egison is a programming language that features extensible efficient non-linear pattern matching with backtracking for non-free data types.
We can directly represent pattern matching for a wide range of data types including lists, multisets, sets, trees, graphs, and mathematical expressions.
Egison makes programming dramatically simple!

```
;; Extract all twin primes from the infinite list of prime numbers with pattern matching!
(define $twin-primes
  (match-all primes (list integer)
    [<join _ <cons $p <cons ,(+ p 2) _>>>
     [p (+ p 2)]]))

;; Enumerate first 10 twin primes.
(take 10 twin-primes)
;=>{[3 5] [5 7] [11 13] [17 19] [29 31] [41 43] [59 61] [71 73] [101 103] [107 109]}
```

Pattern-Matching-Oriented

Egison proposes a new paradigm **pattern-matching-oriented**. The combination of **all of the following features** enables intuitive powerful pattern matching.

- Efficiency of the backtracking algoirthm for non-linear patterns
- Extensibility of pattern matching
- Polymorphisim in patterns

Computer Algebra System

Egison allows programmers to use **tensor index notation** including the support for **differential forms**. Egison introduces two types of parameters, scalar and tensor parameters, and a set of simple index reduction rules for that.

Online Demonstrations

Please try Egison's original features.

- Pattern Matching
  - Poker Hands
  - Mahjong
  - Twin Primes
- Computer Algebra System
  - Riemann Curvature Tensor of S^2
  - Hodge Operator of Minkowski Space
  - Hodge Laplacian of Polar Coordinates

S. Egi and Y. Nishiwaki: "Non-linear Pattern Matching with Backtracking for Non-free Data Types", APLAS 2018 https://arxiv.org/pdf/1808.10603.pdf

MiniEgison: a new pattern-matching library for Haskell

This presentation introduces the implementation of miniEgison, a Haskell library that provides the pattern-matching facility of Egison, which is **compilable** and **type-inferable** by GHC.

```
take 8 (matchAll primes (List Integer)
          [[mc| join _ (cons $p (cons #(p+2) _)) => (p, p+2) |]])
-- [(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),(59,61),(71,73)]
```

Haskell program that enumerates twin primes by pattern matching.

MiniEgison: a new pattern-matching library for Haskell

MiniEgison is implemented utilizing the following Haskell features (GHC extensions):

- Template Haskell;
- generalized algebraic data types;
- existential types;
- datatype promotion;
- multi-parameter type classes.

This presentation shows how these Haskell features are utilized for implementing miniEgison.

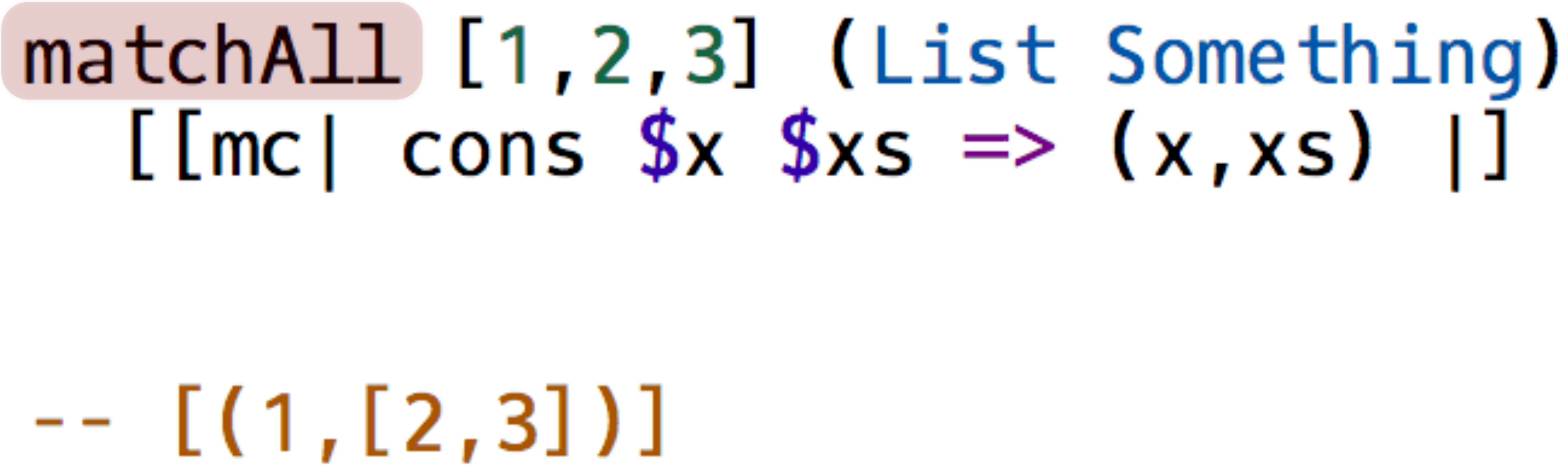
Today's Contents

- Tutorial of MiniEgison
- Background
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- Implementation of MiniEgison
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- Performance
- Conclusion

Today's Contents

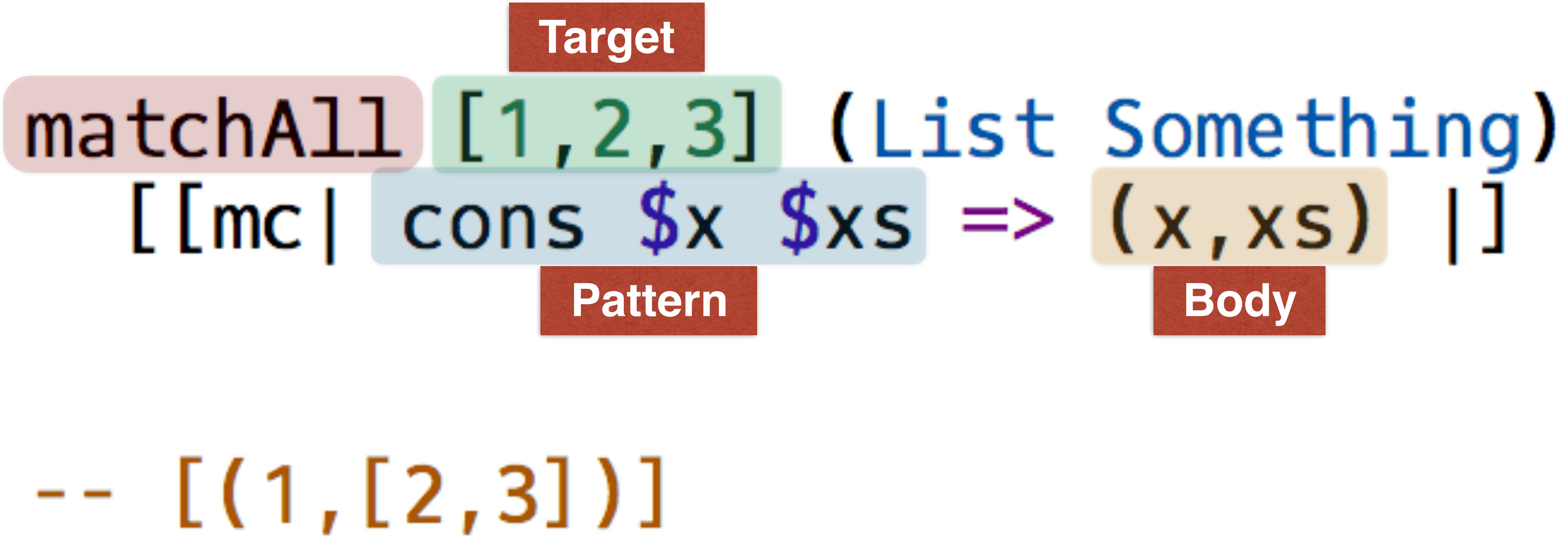
- **Tutorial of MiniEgison**
- Background
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- Implementation of MiniEgison
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- Performance
- Conclusion

Tutorial: the matchAll expression



```
matchAll [1,2,3] (List Something)
   [[mc| cons $x $xs => (x,xs) |]

-- [(1,[2,3])]
```

Tutorial: the matchAll expression

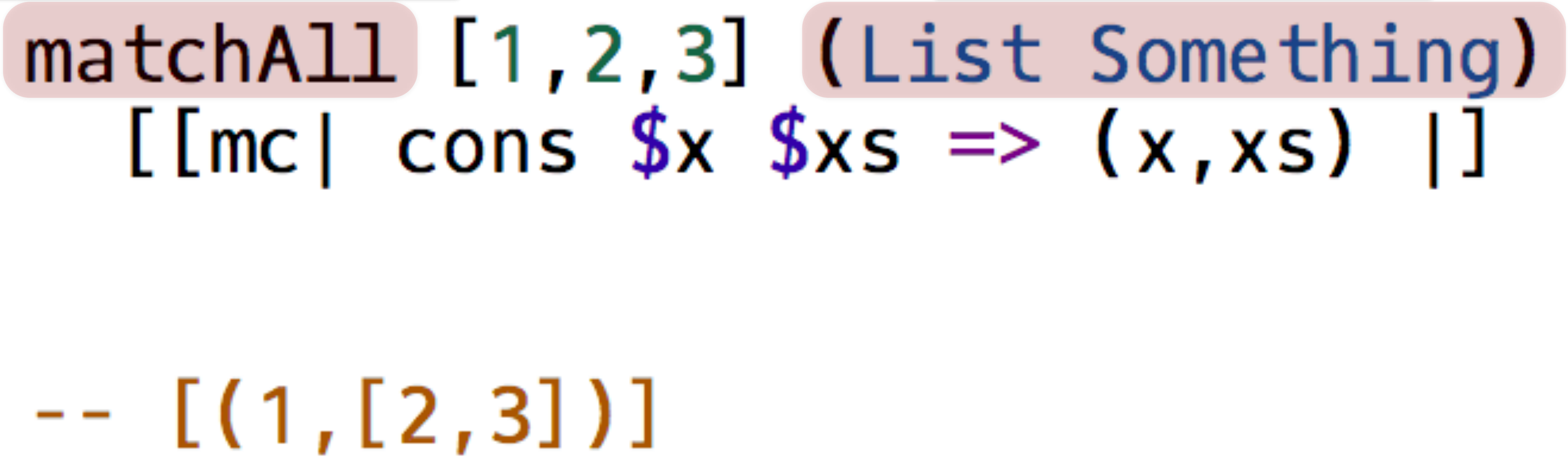


```
matchAll [1,2,3] (List Something)
  [[mc| cons $x $xs => (x,xs) |]]

-- [(1,[2,3])]
```

Tutorial: the matchAll expression

What is MatchAll?

What is matcher?



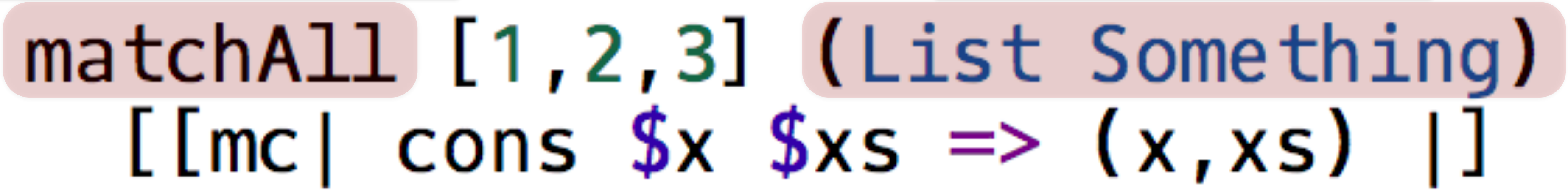
```
matchAll [1,2,3] (List Something)
   [[mc| cons $x $xs => (x,xs) |]

-- [(1,[2,3])]
```

This expression has two characteristic parts.

Tutorial: the matchAll expression

What is MatchAll?

What is matcher?



```
matchAll [1,2,3] (List Something)
  [[mc| cons $x $xs => (x,xs) |]]
```

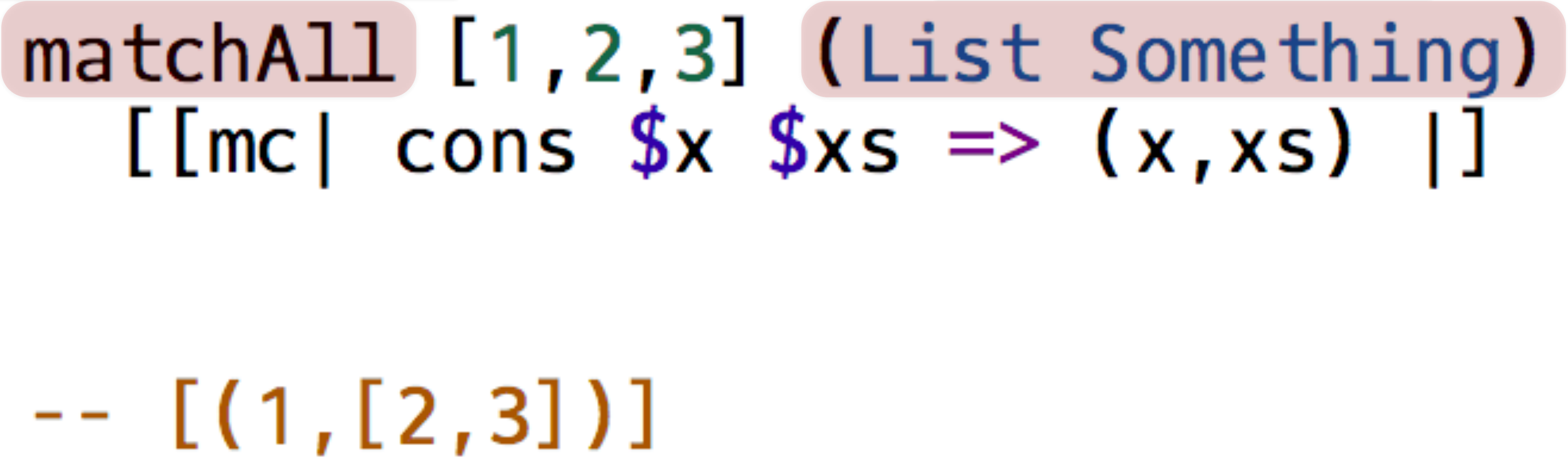
```
-- [(1,[2,3])]
```

MatchAll returns a list of all pattern-matching results.

Tutorial: the matchAll expression

What is MatchAll?

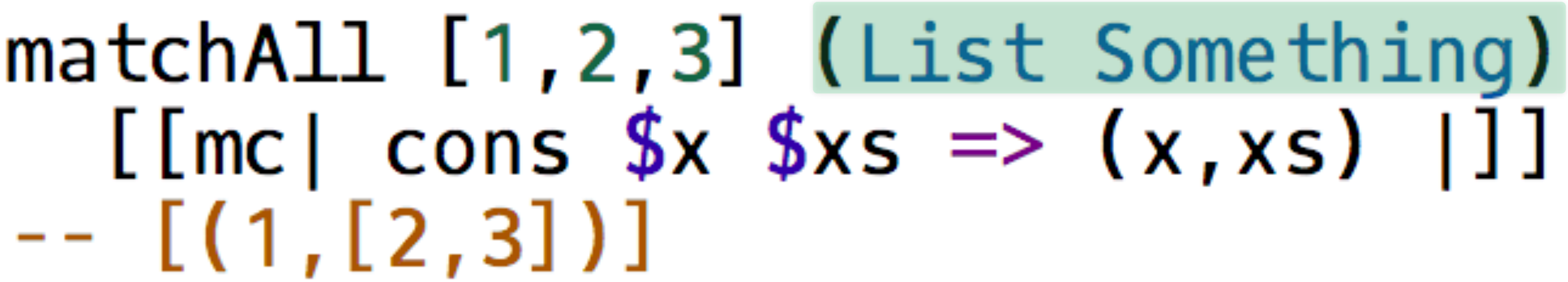
What is matcher?



```
matchAll [1,2,3] (List Something)
  [[mc| cons $x $xs => (x,xs) |]

-- [(1,[2,3])]
```

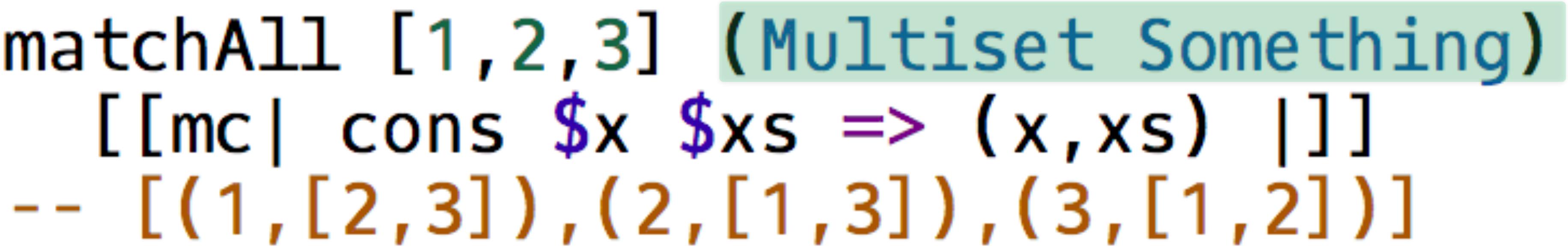
Matcher specifies the pattern matching method.

Tutorial: ad-hoc polymorphism of patterns



```
matchAll [1,2,3] (List Something)
  [[mc| cons $x $xs => (x,xs) |]]
-- [(1,[2,3])]
```

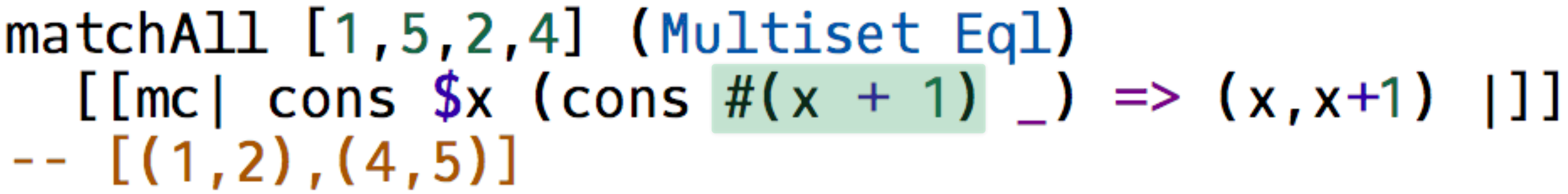
Pattern-matching result depends on matchers.



```
matchAll [1,2,3] (Multiset Something)
  [[mc| cons $x $xs => (x,xs) |]]
-- [(1,[2,3]),(2,[1,3]),(3,[1,2])]
```

The pattern-matching algorithms for `List` and `Multiset` are user-defined.

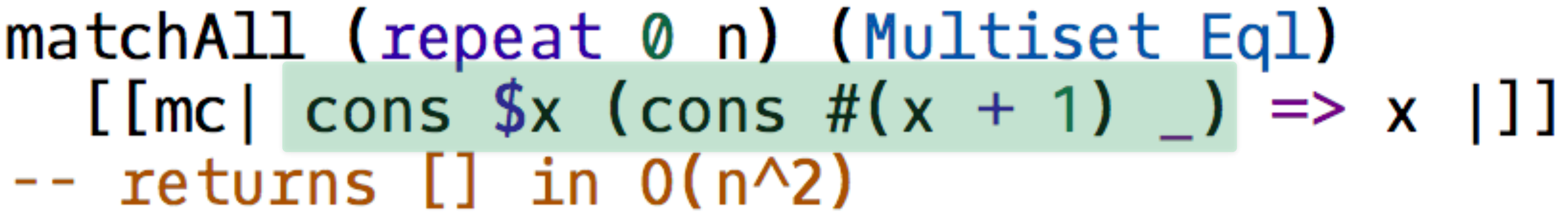
Tutorial: non-linear pattern matching



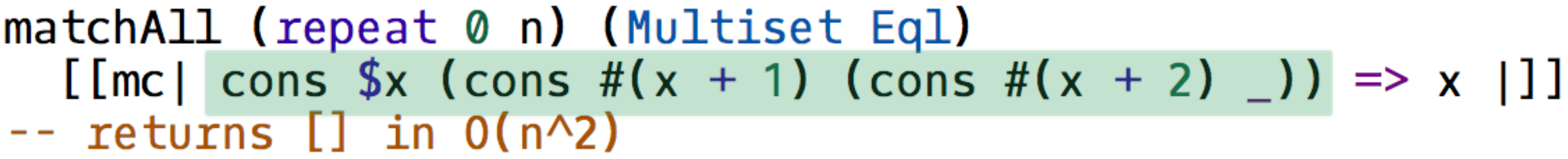
```
matchAll [1,5,2,4] (Multiset Eql)
  [[mc| cons $x (cons #(x + 1) _) => (x,x+1) |]]
-- [(1,2),(4,5)]
```

- (Mini)Egison can handle non-linear patterns.
- Non-linear patterns allow to refer to the value bound to the pattern variables appear in the left-side of the pattern.
- # is used to denote the value pattern.
- The expression that follows after # is evaluated and equality against the target is checked.

Tutorial: non-linear pattern matching with backtracking



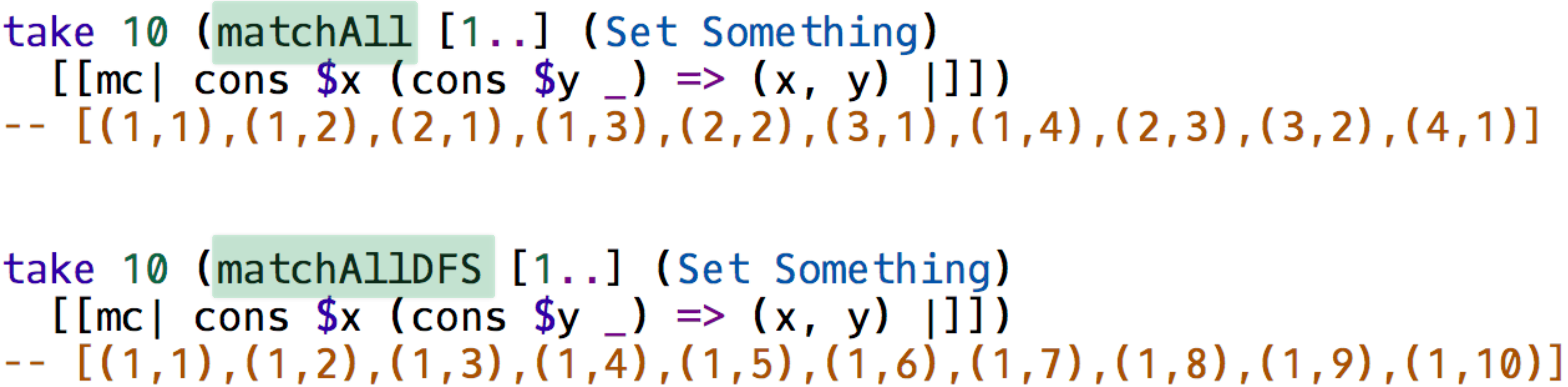
```
matchAll (repeat 0 n) (Multiset Eql)
  [[mc| cons $x (cons #(x + 1) _) => x |]]
-- returns [] in O(n^2)
```



```
matchAll (repeat 0 n) (Multiset Eql)
  [[mc| cons $x (cons #(x + 1) (cons #(x + 2) _)) => x |]]
-- returns [] in O(n^2)
```

- (Mini)Egison uses backtracking for traversing the search trees.
- Therefore, no unnecessary enumerations occur like (naively implemented) pattern guards.

Tutorial: infinitely many pattern-matching results



```
take 10 (matchAll [1..] (Set Something)
  [[mc| cons $x (cons $y _) => (x, y) |]])
-- [(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),(4,1)]

take 10 (matchAllDFS [1..] (Set Something)
  [[mc| cons $x (cons $y _) => (x, y) |]])
-- [(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]
```

Tutorial: the match expression

```
match tgt m cs = head $ matchAll tgt m cs
```

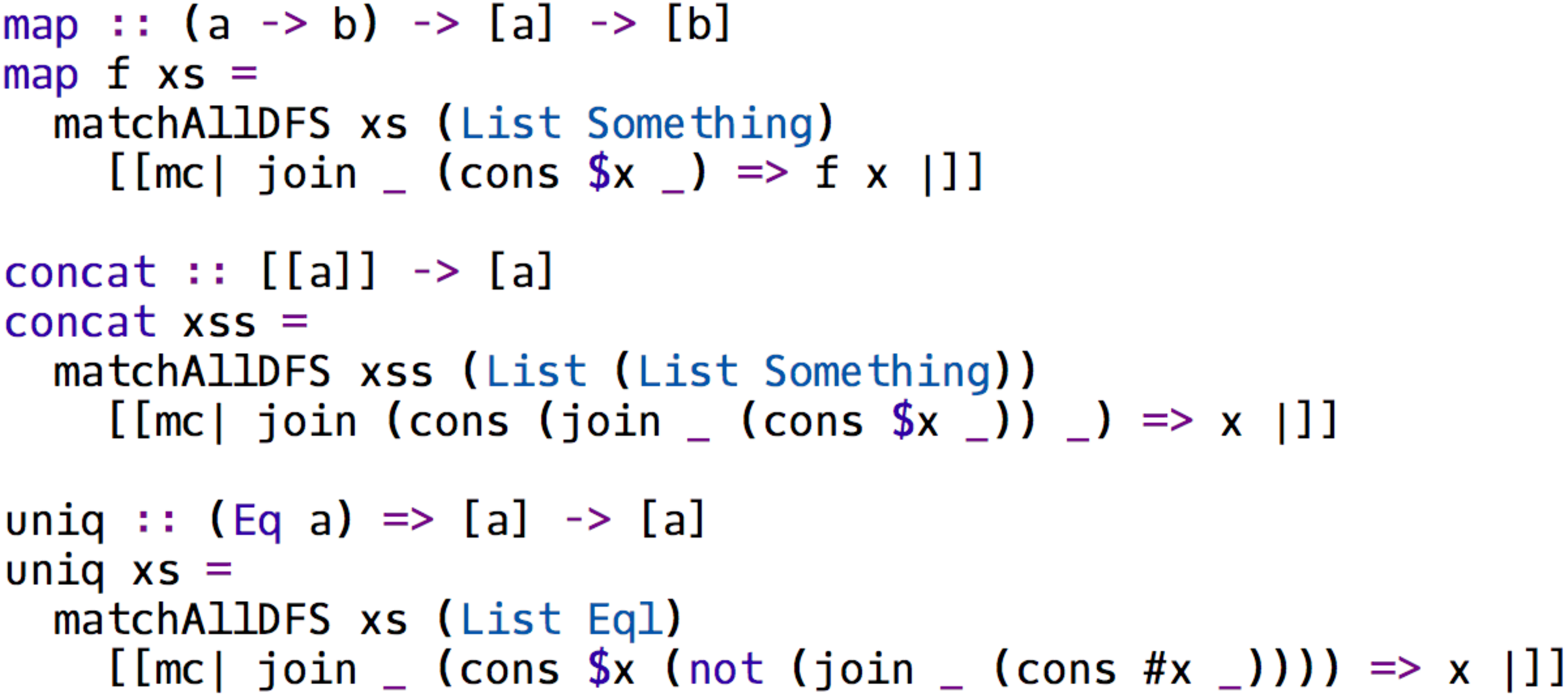
```
poker cs =
  match cs (Multiset CardM)
    [[mc| cons (card $s $n)
           (cons (card #s #(n-1))
            (cons (card #s #(n-2))
             (cons (card #s #(n-3))
              (cons (card #s #(n-4))
               _)))) => "Straight flush" |],
     [mc| cons (card _ $n)
           (cons (card _ #n)
            (cons (card _ #n)
             (cons (card _ #n)
              (cons _
               _)))) => "Four of a kind" |],
     ...]
```

Tutorial: the match expression

```
match tgt m cs = head $ matchAll tgt m cs
```

```
poker cs =
  match cs (Multiset CardM)
    [[mc| cons (card $s $n)
           (cons (card #s #(n-1))
            (cons (card #s #(n-2))
             (cons (card #s #(n-3))
              (cons (card #s #(n-4))
               _)))) => "Straight flush" |],
     [mc| cons (card _ $n)
           (cons (card _ #n)
            (cons (card _ #n)
             (cons (card _ #n)
              (cons _
               _)))) => "Four of a kind" |],
     ...]
```

Tutorial: list functions in pattern-matching-oriented programming



```
map :: (a -> b) -> [a] -> [b]
map f xs =
  matchAllDFS xs (List Something)
    [[mc| join _ (cons $x _) => f x |]]

concat :: [[a]] -> [a]
concat xss =
  matchAllDFS xss (List (List Something))
    [[mc| join (cons (join _ (cons $x _)) _) => x |]]

uniq :: (Eq a) => [a] -> [a]
uniq xs =
  matchAllDFS xs (List Eql)
    [[mc| join _ (cons $x (not (join _ (cons #x _)))) => x |]]
```

Tutorial: list functions in pattern-matching-oriented programming

- `join` splits a list into two lists.
- $x is matched to each element of the list.

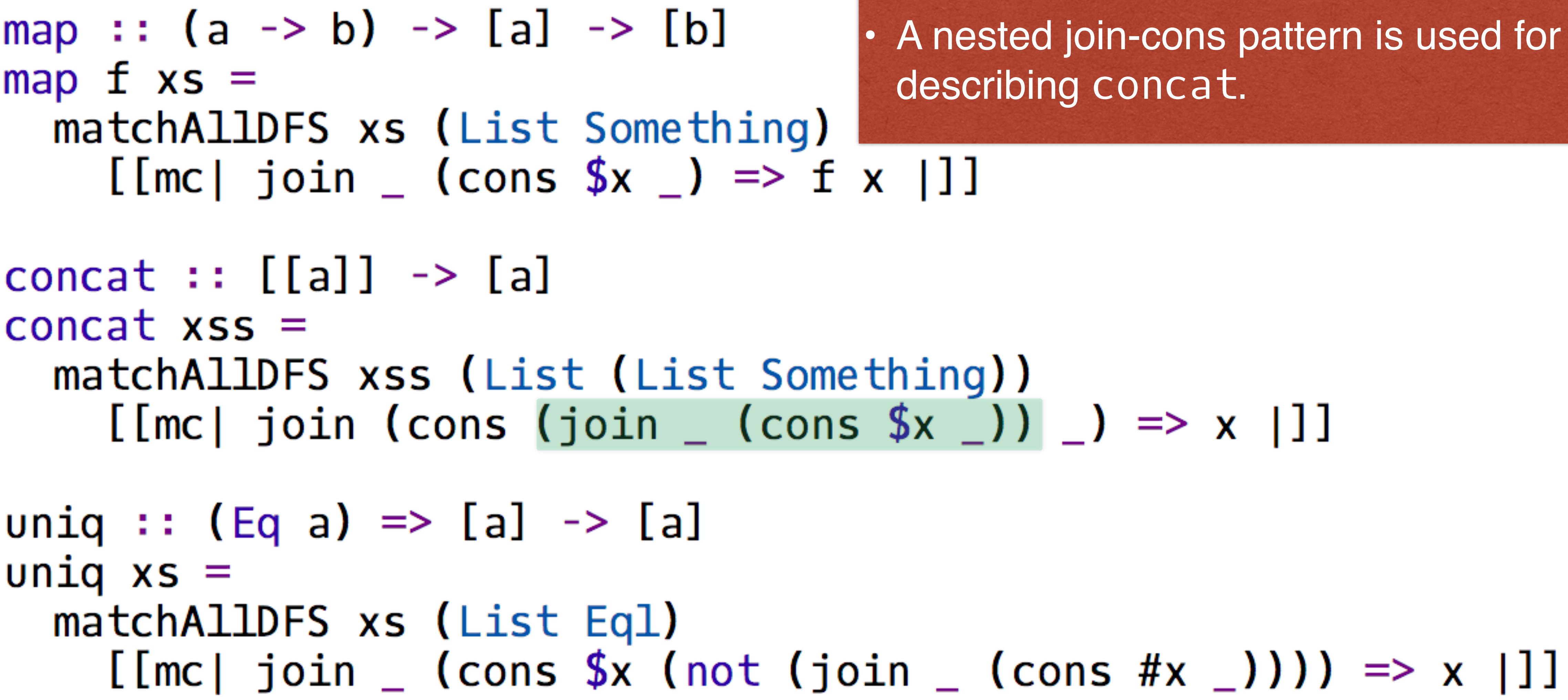
```
map :: (a -> b) -> [a] -> [b]
map f xs =
  matchAllDFS xs (List Something)
    [[mc| join _ (cons $x _) => f x |]]
```

```
concat :: [[a]] -> [a]
concat xss =
  matchAllDFS xss (List (List Something))
    [[mc| join (cons (join _ (cons $x _)) _) => x |]]
```

```
uniq :: (Eq a) => [a] -> [a]
uniq xs =
  matchAllDFS xs (List Eql)
    [[mc| join _ (cons $x (not (join _ (cons #x _)))) => x |]]
```

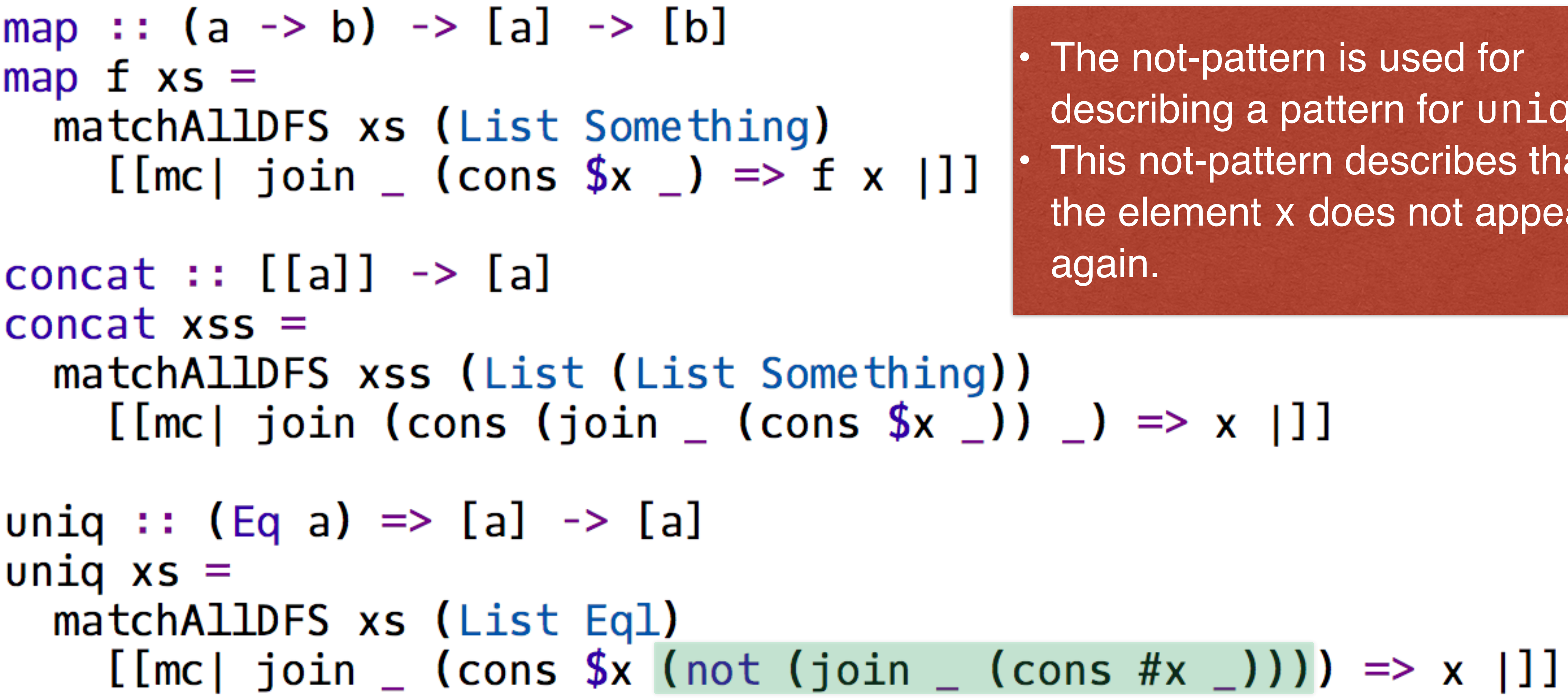
Tutorial: list functions in pattern-matching-oriented programming

- A nested join-cons pattern is used for describing `concat`.



```
map :: (a -> b) -> [a] -> [b]
map f xs =
  matchAllDFS xs (List Something)
    [[mc| join _ (cons $x _) => f x |]]

concat :: [[a]] -> [a]
concat xss =
  matchAllDFS xss (List (List Something))
    [[mc| join (cons (join _ (cons $x _)) _) => x |]]

uniq :: (Eq a) => [a] -> [a]
uniq xs =
  matchAllDFS xs (List Eql)
    [[mc| join _ (cons $x (not (join _ (cons #x _)))) => x |]]
```

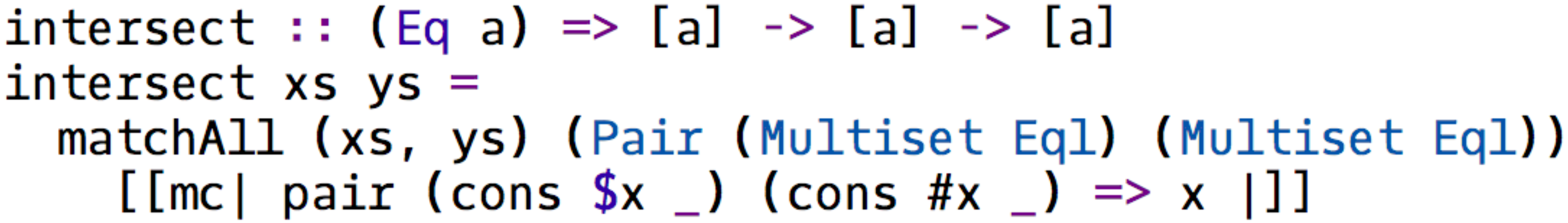
Tutorial: list functions in pattern-matching-oriented programming



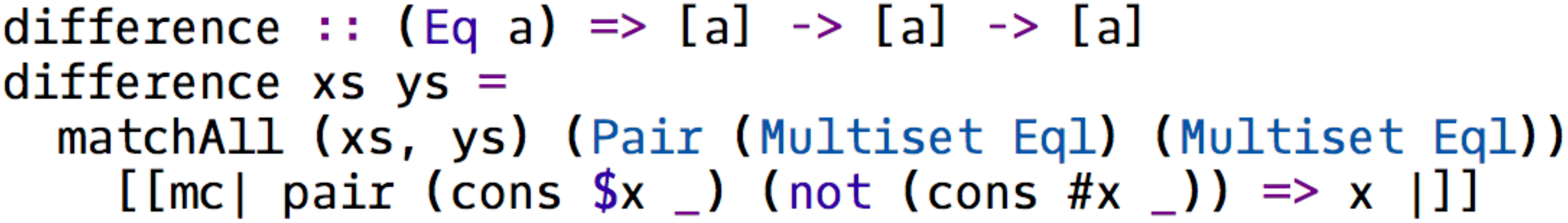
```
map :: (a -> b) -> [a] -> [b]
map f xs =
  matchAllDFS xs (List Something)
    [[mc| join _ (cons $x _) => f x |]]

concat :: [[a]] -> [a]
concat xss =
  matchAllDFS xss (List (List Something))
    [[mc| join (cons (join _ (cons $x _)) _) => x |]]

uniq :: (Eq a) => [a] -> [a]
uniq xs =
  matchAllDFS xs (List Eql)
    [[mc| join _ (cons $x (not (join _ (cons #x _)))) => x |]]
```

- The not-pattern is used for describing a pattern for `unique`.
- This not-pattern describes that the element x does not appear again.

Tutorial: list functions in pattern-matching-oriented programming



```
intersect :: (Eq a) => [a] -> [a] -> [a]
intersect xs ys =
  matchAll (xs, ys) (Pair (Multiset Eql) (Multiset Eql))
    [[mc| pair (cons $x _) (cons #x _) => x |]]
```



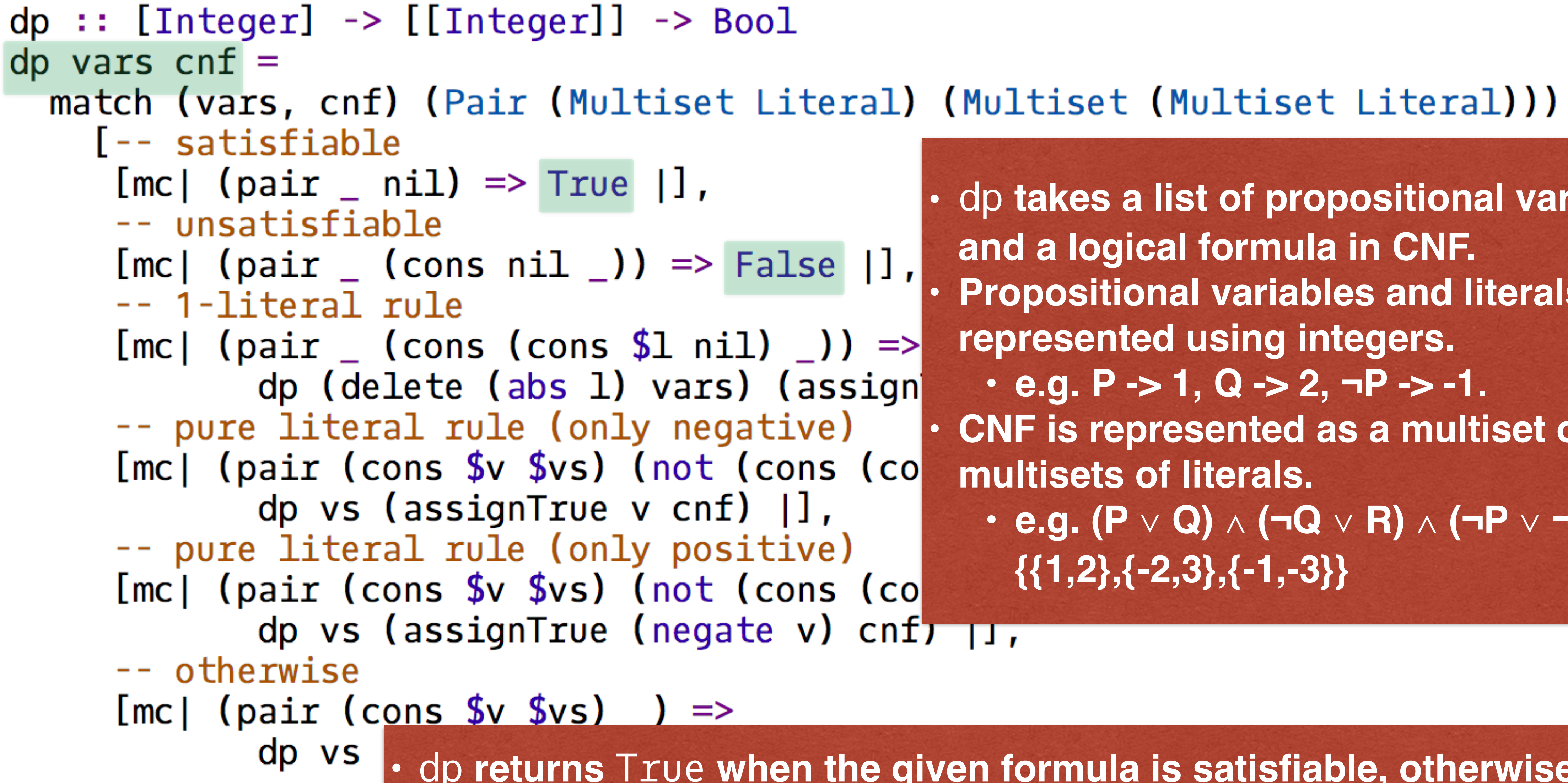
```
difference :: (Eq a) => [a] -> [a] -> [a]
difference xs ys =
  matchAll (xs, ys) (Pair (Multiset Eql) (Multiset Eql))
    [[mc| pair (cons $x _) (not (cons #x _)) => x |]]
```

- Pattern-matching against a tuple is often used for comparing two data.

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
            dp vs (resolveOn v cnf ++
                   deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

Tutorial: SAT solver (Davis-Putnam Algorithm)



```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assign
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (co
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (co
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs)  ) =>
            dp vs
```

- dp **takes a list of propositional variables and a logical formula in CNF.**
- **Propositional variables and literals are represented using integers.**
  - **e.g. P -> 1, Q -> 2, ¬P -> -1.**
- **CNF is represented as a multiset of multisets of literals.**
  - **e.g. (P ∨ Q) ∧ (¬Q ∨ R) ∧ (¬P ∨ ¬R) -> {{1,2},{-2,3},{-1,-3}}**

- dp **returns** `True` **when the given formula is satisfiable, otherwise returns** `False`**.**

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |]
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
            dp vs (resolveOn v cnf ++
                   deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- `vars` and `cnf` are pattern-matched as a multiset of integers and a multiset of multisets of integers, respectively.

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
            dp vs (resolveOn v cnf ++
                   deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- **If `cnf` is empty, `cnf` is satisfiable.**

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
            dp vs (resolveOn v cnf ++
                     deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- **If `cnf` contains an empty clause, `cnf` is unsatisfiable.**

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
            dp vs (resolveOn v cnf ++
                   deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- **If `cnf` contains a clause that consists of a single literal `x`, we can assign `x` `true` at once.**

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
            dp vs (resolveOn v cnf ++
                   deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- **If a propositional variable `v` appears only positively in `cnf`, we can assign `v` `true` at once.**

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
            dp vs (resolveOn v cnf ++
                   deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- **If a propositional variable** v **appears only negatively in** `cnf`**, we can assign** v `false` **at once.**

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
            dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
            dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
            dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
            dp vs (resolveOn v cnf ++
                   deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- **Otherwise, we apply the resolution principle.**

Tutorial: SAT solver (Davis-Putnam Algorithm)

```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
 match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
  [-- satisfiable
   [mc| (pair _ nil) => True |],
   -- unsatisfiable
   [mc| (pair _ (cons nil _)) => False |],
   -- 1-literal rule
   [mc| (pair _ (cons (cons $l nil) _)) =>
          dp (delete (abs l) vars) (assignTrue l cnf) |],
   -- pure literal rule (only negative)
   [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
          dp vs (assignTrue v cnf) |],
   -- pure literal rule (only positive)
   [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
          dp vs (assignTrue (negate v) cnf) |],
   -- otherwise
   [mc| (pair (cons $v $vs) _) =>
          dp vs (resolveOn v cnf ++
                 deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- **Egison pattern matching dramatically improves the readability of programs.**

Tutorial: the essence of pattern-matching-oriented programming

There are two kinds of loops in programming:

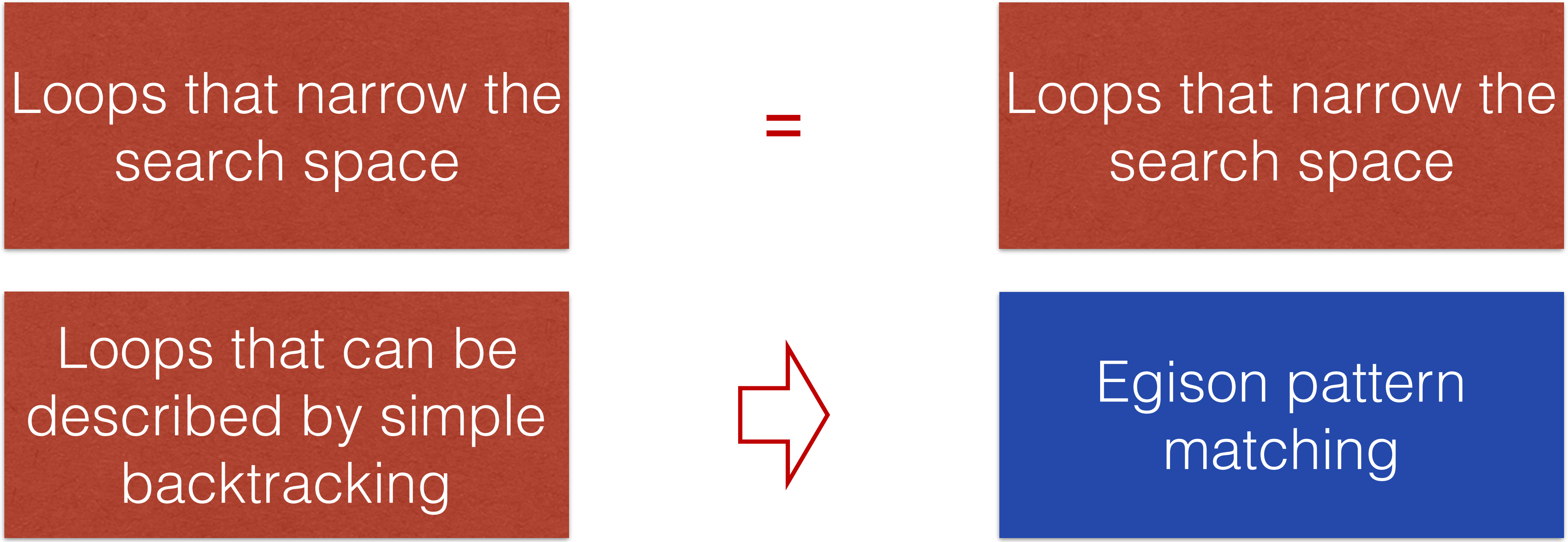
Loops that narrow the search space

Loops that can be described by simple backtracking

Functional programming mixes these loops together in a program.

Tutorial: the essence of pattern-matching-oriented programming

There are two kinds of loops in programming:



Loops that narrow the search space

=

Loops that narrow the search space

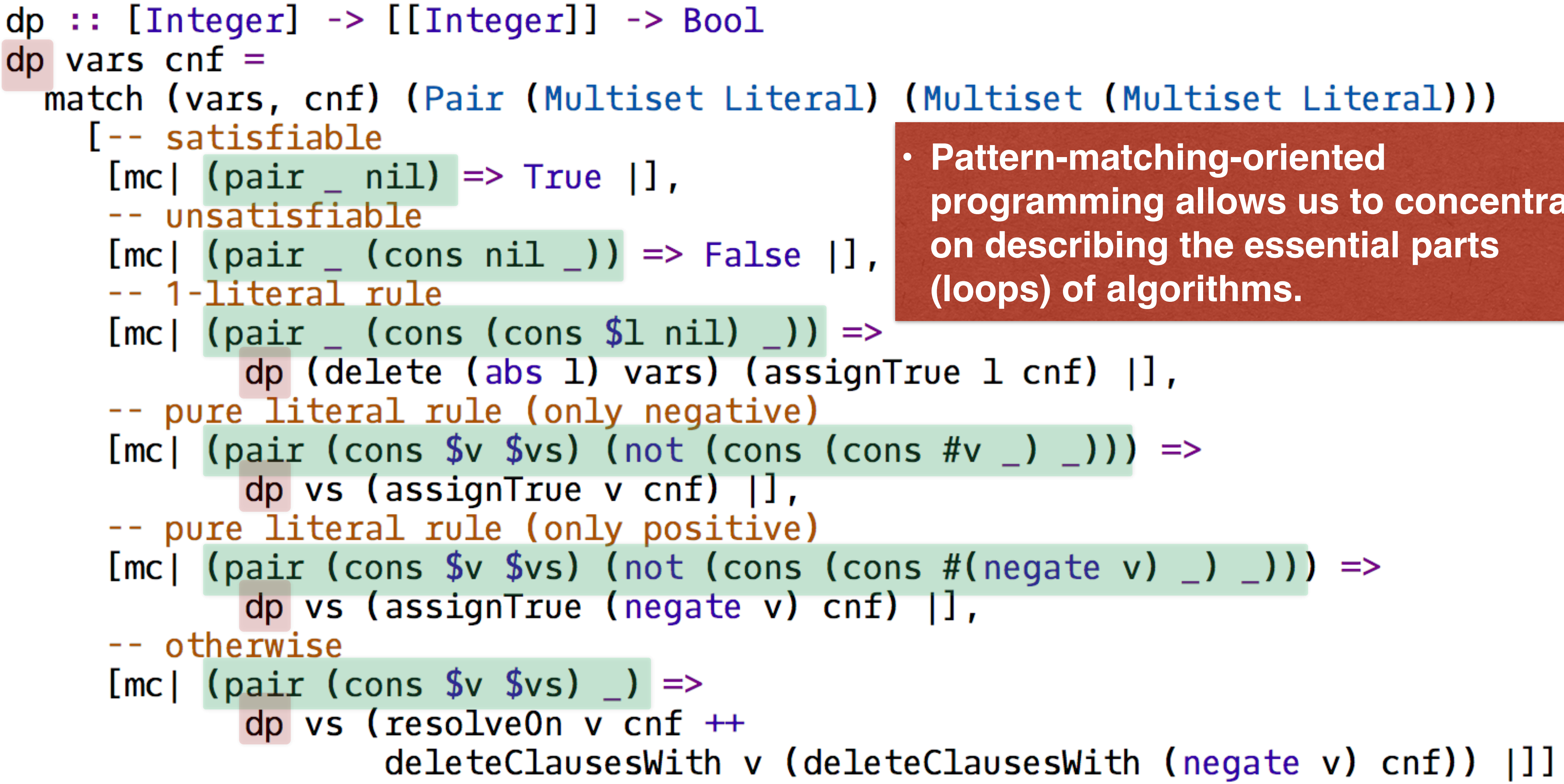
Loops that can be described by simple backtracking

Egison pattern matching

Functional programming mixes these loops together in a program.

Pattern-matching-oriented programming confines the later kind of loops in patterns.

Tutorial: SAT solver (Davis-Putnam Algorithm)



```
dp :: [Integer] -> [[Integer]] -> Bool
dp vars cnf =
  match (vars, cnf) (Pair (Multiset Literal) (Multiset (Multiset Literal)))
    [-- satisfiable
     [mc| (pair _ nil) => True |],
     -- unsatisfiable
     [mc| (pair _ (cons nil _)) => False |],
     -- 1-literal rule
     [mc| (pair _ (cons (cons $l nil) _)) =>
          dp (delete (abs l) vars) (assignTrue l cnf) |],
     -- pure literal rule (only negative)
     [mc| (pair (cons $v $vs) (not (cons (cons #v _) _))) =>
          dp vs (assignTrue v cnf) |],
     -- pure literal rule (only positive)
     [mc| (pair (cons $v $vs) (not (cons (cons #(negate v) _) _))) =>
          dp vs (assignTrue (negate v) cnf) |],
     -- otherwise
     [mc| (pair (cons $v $vs) _) =>
          dp vs (resolveOn v cnf ++
                   deleteClausesWith v (deleteClausesWith (negate v) cnf)) |]]
```

- **Pattern-matching-oriented programming allows us to concentrate on describing the essential parts (loops) of algorithms.**

Today’s Contents

- Tutorial of MiniEgison
- **Background**
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- Implementation of MiniEgison
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- Performance
- Conclusion

Today’s Contents

- Tutorial of MiniEgison
- **Background**
  - **Compilation of Egison Pattern Matching**
  - Type System for Egison Pattern Matching
- Implementation of MiniEgison
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- Performance
- Conclusion

Scheme macros for Egison pattern matching

Compilation of Egison Pattern Matching to dynamically typed programming languages (e.g., Scheme and Lisp) has been already proposed.

The remaining problem is compilation for statically typed languages (e.g., Haskell).

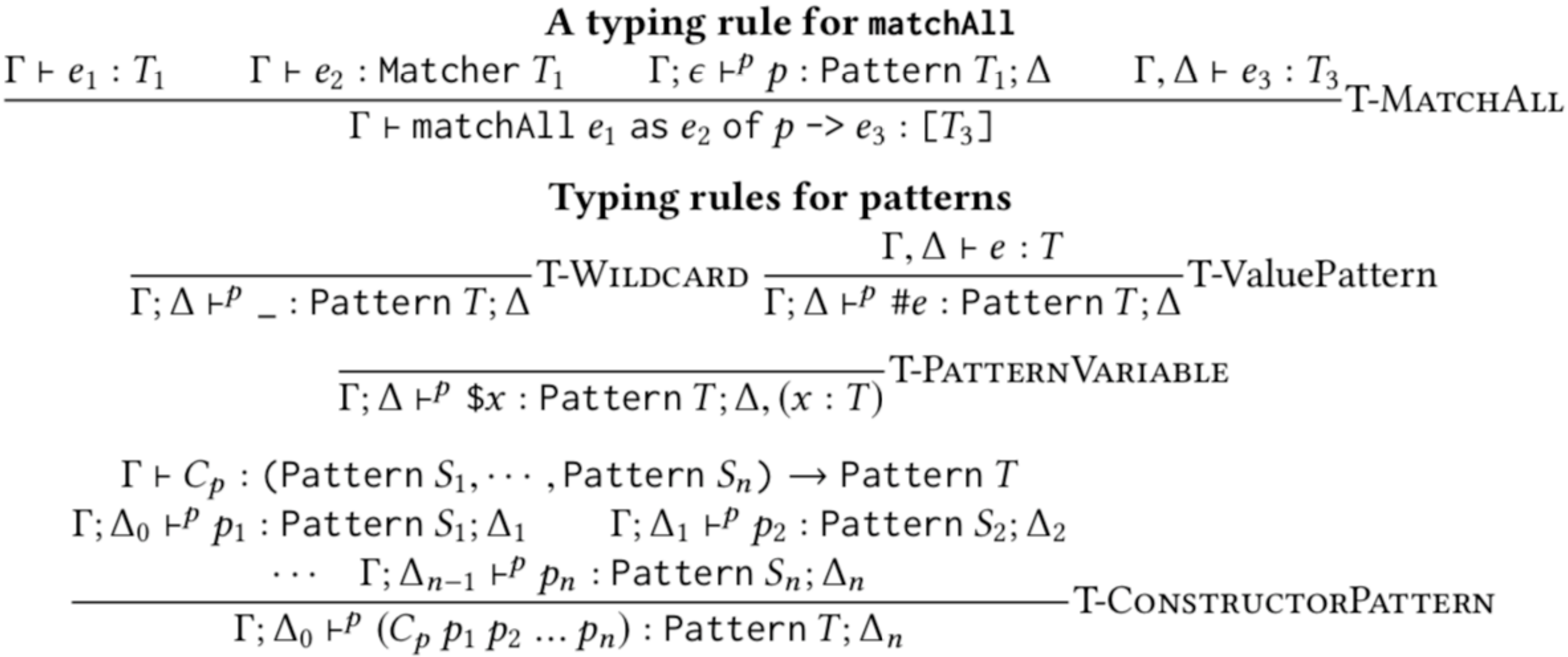
S. Egi: “Scheme Macros for Non-linear Pattern Matching with Backtracking for Non-free Data Types”, Scheme Workshop 2019

MiniEgison takes a similar approach with the Scheme macros.

Today's Contents

- Tutorial of MiniEgison
- **Background**
  - Compilation of Egison Pattern Matching
  - **Type System for Egison Pattern Matching**
- Implementation of MiniEgison
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- Performance
- Conclusion

Typing rules for matchAll and patterns



A typing rule for `matchAll`

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : \texttt{Matcher}\ T_1 \qquad \Gamma; \epsilon \vdash^{p} p : \texttt{Pattern}\ T_1; \Delta \qquad \Gamma, \Delta \vdash e_3 : T_3}{\Gamma \vdash \texttt{matchAll}\ e_1\ \texttt{as}\ e_2\ \texttt{of}\ p\ \texttt{->}\ e_3 : [T_3]}\ \textsc{T-MatchAll}$$

Typing rules for patterns

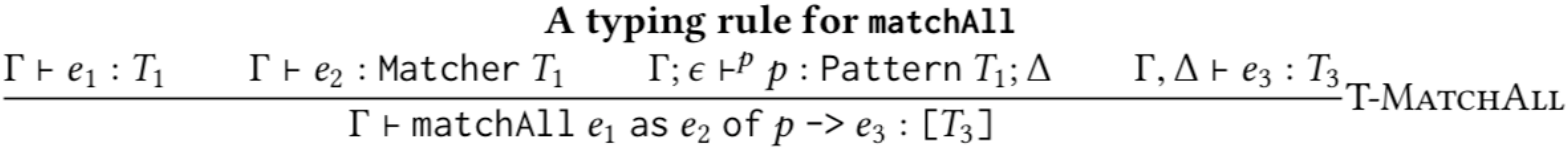
$$\frac{}{\Gamma; \Delta \vdash^{p} \_ : \texttt{Pattern}\ T; \Delta}\ \textsc{T-Wildcard} \qquad \frac{\Gamma, \Delta \vdash e : T}{\Gamma; \Delta \vdash^{p} \#e : \texttt{Pattern}\ T; \Delta}\ \text{T-ValuePattern}$$

$$\frac{}{\Gamma; \Delta \vdash^{p} \$x : \texttt{Pattern}\ T; \Delta, (x : T)}\ \textsc{T-PatternVariable}$$

$$\frac{\begin{array}{c}\Gamma \vdash C_p : (\texttt{Pattern}\ S_1, \cdots, \texttt{Pattern}\ S_n) \to \texttt{Pattern}\ T \\ \Gamma; \Delta_0 \vdash^{p} p_1 : \texttt{Pattern}\ S_1; \Delta_1 \qquad \Gamma; \Delta_1 \vdash^{p} p_2 : \texttt{Pattern}\ S_2; \Delta_2 \\ \cdots \quad \Gamma; \Delta_{n-1} \vdash^{p} p_n : \texttt{Pattern}\ S_n; \Delta_n\end{array}}{\Gamma; \Delta_0 \vdash^{p} (C_p\ p_1\ p_2 \ldots p_n) : \texttt{Pattern}\ T; \Delta_n}\ \textsc{T-ConstructorPattern}$$

Unpublished work by Kawata and Egi designed a type system for Egison.

Typing rules for matchAll



A typing rule for `matchAll`

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : \mathtt{Matcher}\ T_1 \qquad \Gamma; \epsilon \vdash^{p} p : \mathtt{Pattern}\ T_1; \Delta \qquad \Gamma, \Delta \vdash e_3 : T_3}{\Gamma \vdash \mathtt{matchAll}\ e_1\ \mathtt{as}\ e_2\ \mathtt{of}\ p\ \texttt{->}\ e_3 : [T_3]}\ \textsc{T-MatchAll}$$

Typing rules for matchAll

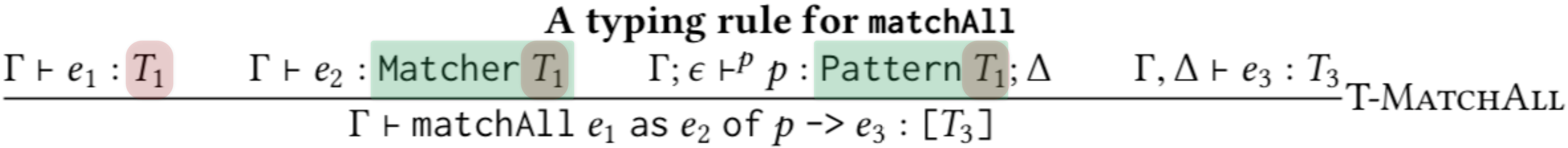
- `Matcher` and `Pattern` are built-in type operators:
  - `Matcher T` is a type for matchers for T;
  - `Pattern T` is a type for patterns for T.

A typing rule for `matchAll`

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : \texttt{Matcher}\ T_1 \qquad \Gamma; \epsilon \vdash^{p} p : \texttt{Pattern}\ T_1; \Delta \qquad \Gamma, \Delta \vdash e_3 : T_3}{\Gamma \vdash \texttt{matchAll}\ e_1\ \texttt{as}\ e_2\ \texttt{of}\ p\ \texttt{->}\ e_3 : [T_3]}\ \textsc{T-MatchAll}$$

Typing rules for matchAll

- `Matcher` and `Pattern` are built-in type operators:
    - `Matcher T` is a type for matchers for T;
    - `Pattern T` is a type for patterns for T.



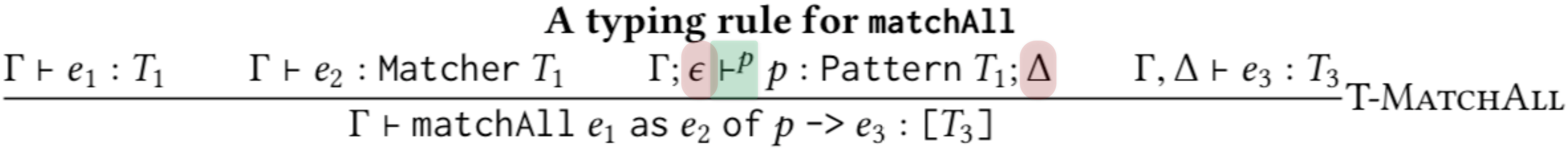
A typing rule for `matchAll`

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : \texttt{Matcher}\ T_1 \qquad \Gamma; \epsilon \vdash^{p} p : \texttt{Pattern}\ T_1; \Delta \qquad \Gamma, \Delta \vdash e_3 : T_3}{\Gamma \vdash \texttt{matchAll}\ e_1\ \texttt{as}\ e_2\ \texttt{of}\ p\ \texttt{->}\ e_3 : [T_3]}\ \textsc{T-MatchAll}$$

- `matchAll` takes a matcher and a pattern for the same type with the target.

Typing rules for matchAll

- Patterns have the special judgement operator for handling non-linear patterns.



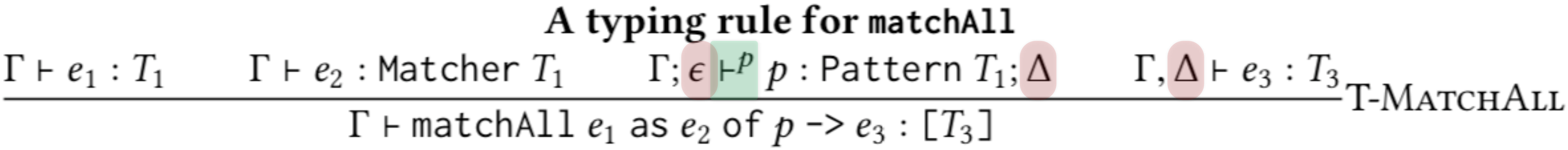
A typing rule for matchAll

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : \mathtt{Matcher}\ T_1 \qquad \Gamma; \epsilon \vdash^{p} p : \mathtt{Pattern}\ T_1 ; \Delta \qquad \Gamma, \Delta \vdash e_3 : T_3}{\Gamma \vdash \mathtt{matchAll}\ e_1\ \mathtt{as}\ e_2\ \mathtt{of}\ p\ \mathtt{->}\ e_3 : [T_3]}\ \textsc{T-MatchAll}$$

- ε and Δ denotes the a type environment for patterns.
- ε denotes an empty type environment.
- ε is an input type environment.
- Δ is an output type environment.

Typing rules for matchAll

- Patterns have the special judgement operator for handling non-linear patterns.



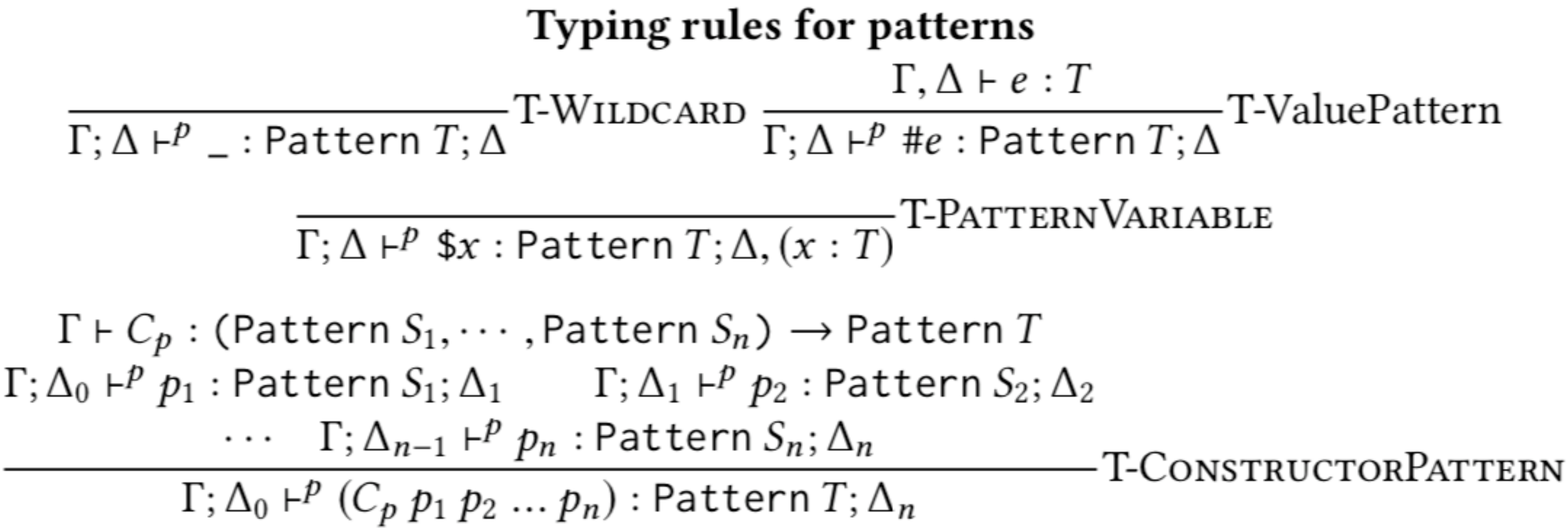
A typing rule for `matchAll`

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : \mathtt{Matcher}\ T_1 \qquad \Gamma; \epsilon \vdash^{p} p : \mathtt{Pattern}\ T_1; \Delta \qquad \Gamma, \Delta \vdash e_3 : T_3}{\Gamma \vdash \mathtt{matchAll}\ e_1\ \mathtt{as}\ e_2\ \mathtt{of}\ p\ \mathtt{->}\ e_3 : [T_3]}\ \text{T-MATCHALL}$$

- ε and Δ denotes the a type environment for patterns.
- ε denotes an empty type environment.
- ε is an input type environment.
- Δ is an output type environment.

- Δ is used for evaluating the body of the match clause.

Typing rules for patterns



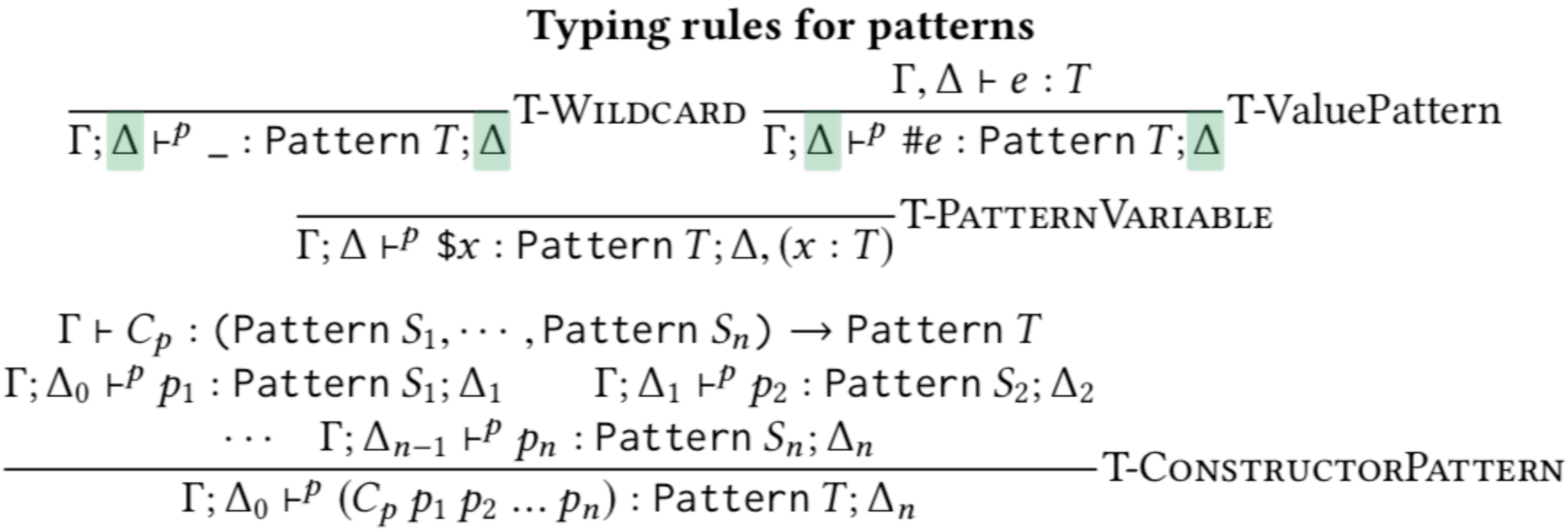
Typing rules for patterns

$$\frac{}{\Gamma; \Delta \vdash^{p} \_ : \mathtt{Pattern}\ T; \Delta}\textsc{T-Wildcard} \qquad \frac{\Gamma, \Delta \vdash e : T}{\Gamma; \Delta \vdash^{p} \#e : \mathtt{Pattern}\ T; \Delta}\text{T-ValuePattern}$$

$$\frac{}{\Gamma; \Delta \vdash^{p} \$x : \mathtt{Pattern}\ T; \Delta, (x : T)}\textsc{T-PatternVariable}$$

$$\frac{\begin{array}{c}\Gamma \vdash C_p : (\mathtt{Pattern}\ S_1, \cdots, \mathtt{Pattern}\ S_n) \to \mathtt{Pattern}\ T \\ \Gamma; \Delta_0 \vdash^{p} p_1 : \mathtt{Pattern}\ S_1; \Delta_1 \qquad \Gamma; \Delta_1 \vdash^{p} p_2 : \mathtt{Pattern}\ S_2; \Delta_2 \\ \cdots \quad \Gamma; \Delta_{n-1} \vdash^{p} p_n : \mathtt{Pattern}\ S_n; \Delta_n\end{array}}{\Gamma; \Delta_0 \vdash^{p} (C_p\ p_1\ p_2 \ldots p_n) : \mathtt{Pattern}\ T; \Delta_n}\textsc{T-ConstructorPattern}$$

Typing rules for patterns



Typing rules for patterns

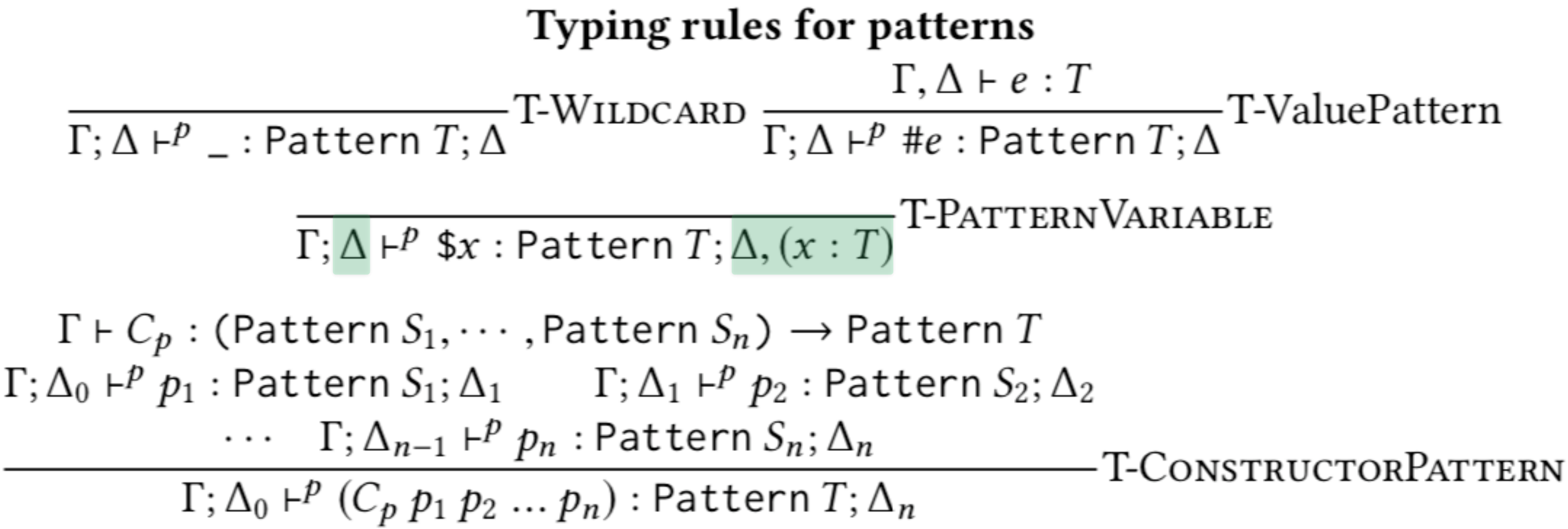
$$\frac{}{\Gamma; \Delta \vdash^{p} \_ : \mathtt{Pattern}\ T; \Delta}\ \textsc{T-Wildcard} \qquad \frac{\Gamma, \Delta \vdash e : T}{\Gamma; \Delta \vdash^{p} \#e : \mathtt{Pattern}\ T; \Delta}\ \text{T-ValuePattern}$$

$$\frac{}{\Gamma; \Delta \vdash^{p} \$x : \mathtt{Pattern}\ T; \Delta, (x : T)}\ \textsc{T-PatternVariable}$$

$$\frac{\begin{array}{c} \Gamma \vdash C_p : (\mathtt{Pattern}\ S_1, \cdots, \mathtt{Pattern}\ S_n) \to \mathtt{Pattern}\ T \\ \Gamma; \Delta_0 \vdash^{p} p_1 : \mathtt{Pattern}\ S_1; \Delta_1 \qquad \Gamma; \Delta_1 \vdash^{p} p_2 : \mathtt{Pattern}\ S_2; \Delta_2 \\ \cdots \quad \Gamma; \Delta_{n-1} \vdash^{p} p_n : \mathtt{Pattern}\ S_n; \Delta_n \end{array}}{\Gamma; \Delta_0 \vdash^{p} (C_p\ p_1\ p_2\ \ldots\ p_n) : \mathtt{Pattern}\ T; \Delta_n}\ \textsc{T-ConstructorPattern}$$

- Wildcards and value patterns make no new bindings.

Typing rules for patterns



Typing rules for patterns

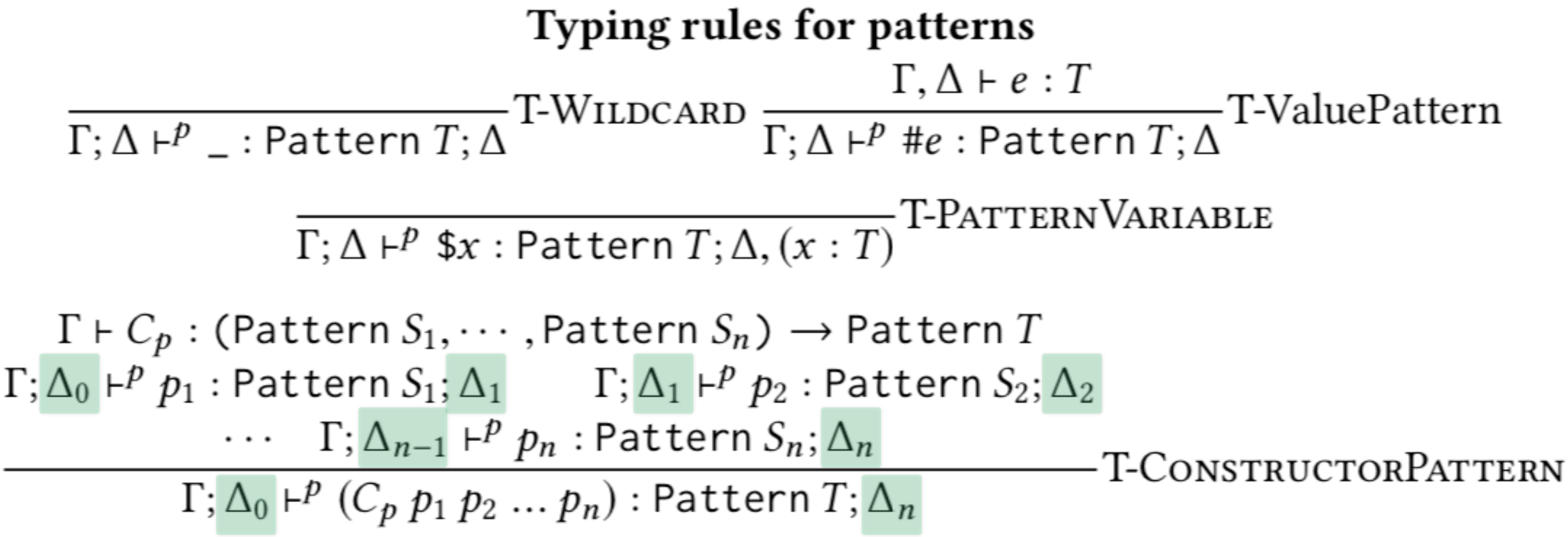
$$\frac{}{\Gamma; \Delta \vdash^{p} \_ : \mathtt{Pattern}\ T; \Delta}\textsc{T-Wildcard} \qquad \frac{\Gamma, \Delta \vdash e : T}{\Gamma; \Delta \vdash^{p} \#e : \mathtt{Pattern}\ T; \Delta}\text{T-ValuePattern}$$

$$\frac{}{\Gamma; \Delta \vdash^{p} \$x : \mathtt{Pattern}\ T; \Delta, (x : T)}\textsc{T-PatternVariable}$$

$$\frac{\begin{array}{c} \Gamma \vdash C_p : (\mathtt{Pattern}\ S_1, \cdots, \mathtt{Pattern}\ S_n) \to \mathtt{Pattern}\ T \\ \Gamma; \Delta_0 \vdash^{p} p_1 : \mathtt{Pattern}\ S_1; \Delta_1 \qquad \Gamma; \Delta_1 \vdash^{p} p_2 : \mathtt{Pattern}\ S_2; \Delta_2 \\ \cdots \quad \Gamma; \Delta_{n-1} \vdash^{p} p_n : \mathtt{Pattern}\ S_n; \Delta_n \end{array}}{\Gamma; \Delta_0 \vdash^{p} (C_p\ p_1\ p_2\ \ldots\ p_n) : \mathtt{Pattern}\ T; \Delta_n}\textsc{T-ConstructorPattern}$$

- A pattern variable adds a new binding to the type environment.
- If $x is the `Pattern T`, then x has the type T.

Typing rules for patterns



Typing rules for patterns

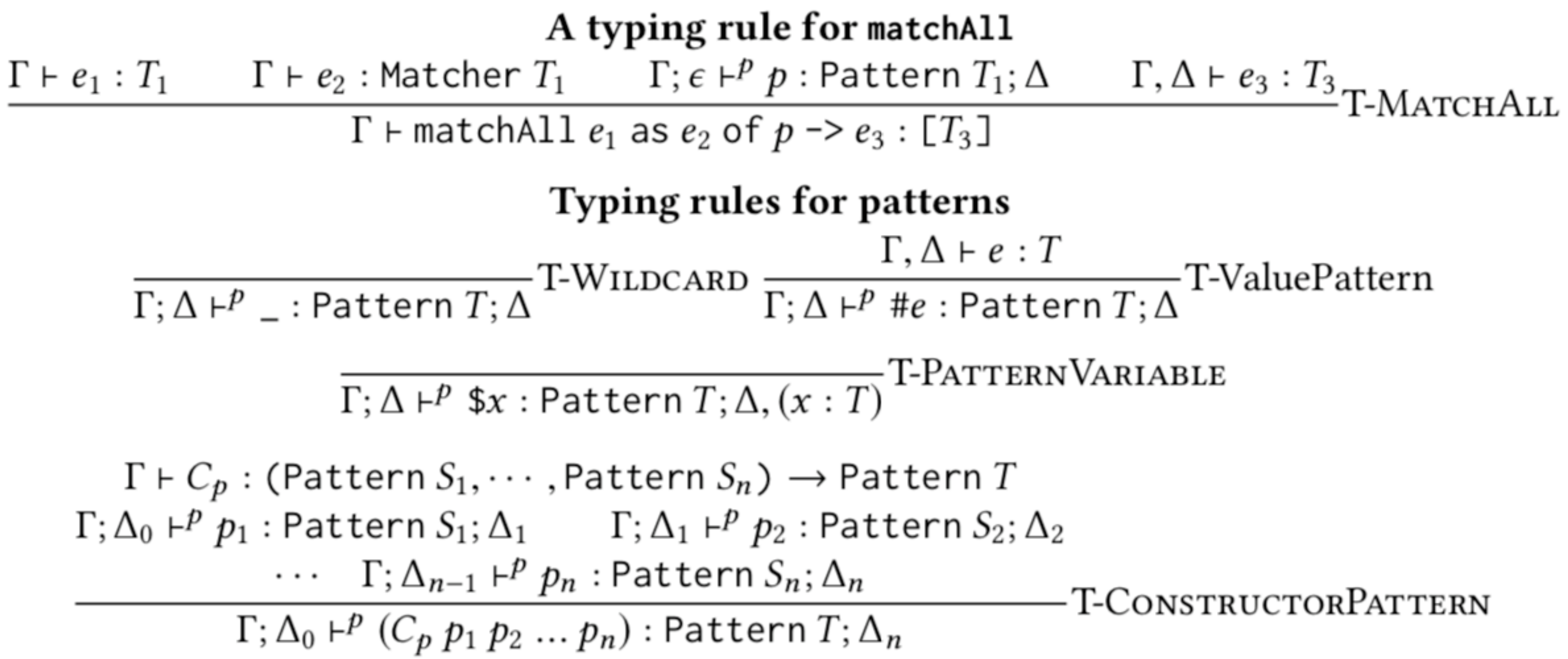
$$\frac{}{\Gamma; \Delta \vdash^p \_ : \mathtt{Pattern}\ T; \Delta}\textsc{T-Wildcard} \qquad \frac{\Gamma, \Delta \vdash e : T}{\Gamma; \Delta \vdash^p \#e : \mathtt{Pattern}\ T; \Delta}\text{T-ValuePattern}$$

$$\frac{}{\Gamma; \Delta \vdash^p \$x : \mathtt{Pattern}\ T; \Delta, (x : T)}\textsc{T-PatternVariable}$$

$$\frac{\begin{array}{c}\Gamma \vdash C_p : (\mathtt{Pattern}\ S_1, \cdots, \mathtt{Pattern}\ S_n) \to \mathtt{Pattern}\ T \\ \Gamma; \Delta_0 \vdash^p p_1 : \mathtt{Pattern}\ S_1; \Delta_1 \qquad \Gamma; \Delta_1 \vdash^p p_2 : \mathtt{Pattern}\ S_2; \Delta_2 \\ \cdots \quad \Gamma; \Delta_{n-1} \vdash^p p_n : \mathtt{Pattern}\ S_n; \Delta_n\end{array}}{\Gamma; \Delta_0 \vdash^p (C_p\ p_1\ p_2 \ldots p_n) : \mathtt{Pattern}\ T; \Delta_n}\textsc{T-ConstructorPattern}$$

- A constructor pattern passes the type environment from left to right to handle non-linear patterns.

Typing rules for matchAll and patterns



A typing rule for `matchAll`

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : \texttt{Matcher}\ T_1 \qquad \Gamma; \epsilon \vdash^p p : \texttt{Pattern}\ T_1; \Delta \qquad \Gamma, \Delta \vdash e_3 : T_3}{\Gamma \vdash \texttt{matchAll}\ e_1\ \texttt{as}\ e_2\ \texttt{of}\ p\ \texttt{->}\ e_3 : [T_3]}\ \textsc{T-MatchAll}$$

Typing rules for patterns

$$\frac{}{\Gamma; \Delta \vdash^p \_ : \texttt{Pattern}\ T; \Delta}\ \textsc{T-Wildcard} \qquad \frac{\Gamma, \Delta \vdash e : T}{\Gamma; \Delta \vdash^p \#e : \texttt{Pattern}\ T; \Delta}\ \text{T-ValuePattern}$$

$$\frac{}{\Gamma; \Delta \vdash^p \$x : \texttt{Pattern}\ T; \Delta, (x : T)}\ \textsc{T-PatternVariable}$$

$$\frac{\begin{array}{c}\Gamma \vdash C_p : (\texttt{Pattern}\ S_1, \cdots, \texttt{Pattern}\ S_n) \to \texttt{Pattern}\ T \\ \Gamma; \Delta_0 \vdash^p p_1 : \texttt{Pattern}\ S_1; \Delta_1 \qquad \Gamma; \Delta_1 \vdash^p p_2 : \texttt{Pattern}\ S_2; \Delta_2 \\ \cdots \quad \Gamma; \Delta_{n-1} \vdash^p p_n : \texttt{Pattern}\ S_n; \Delta_n\end{array}}{\Gamma; \Delta_0 \vdash^p (C_p\ p_1\ p_2 \ldots p_n) : \texttt{Pattern}\ T; \Delta_n}\ \textsc{T-ConstructorPattern}$$

Translating Egison pattern matching expressions to a Haskell program on which the Haskell type system does type-checking equivalent to the above typing rules is challenging!

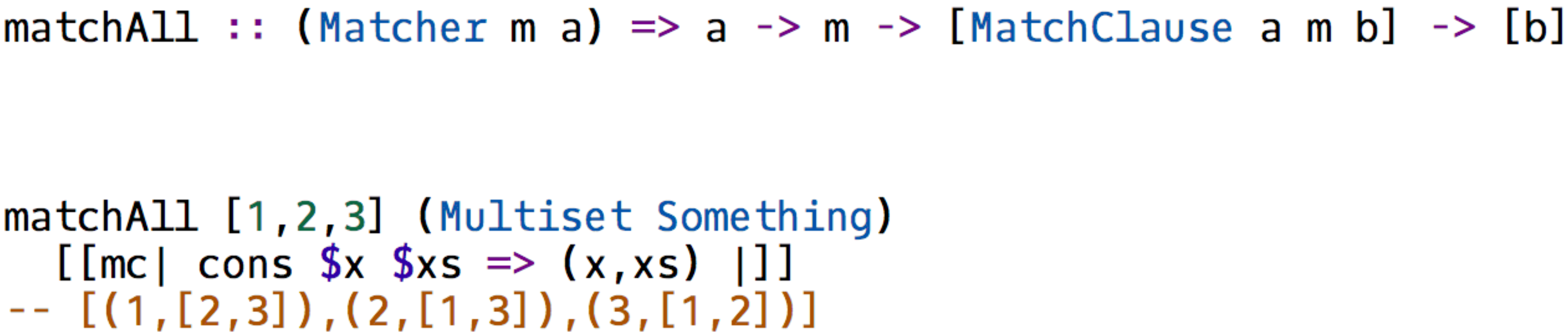
Today's Contents

- Tutorial of MiniEgison
- Background
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- **Implementation of MiniEgison**
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- Performance
- Conclusion

Today’s Contents

- Tutorial of MiniEgison
- Background
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- **Implementation of MiniEgison**
  - **Typing MatchAll**
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- Performance
- Conclusion

Typing the matchAll expression



```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

```
matchAll [1,2,3] (Multiset Something)
  [[mc| cons $x $xs => (x,xs) |]]
-- [(1,[2,3]),(2,[1,3]),(3,[1,2])]
```

Typing the matchAll expression

- MatchAll takes a target, a matcher, and match clauses and returns the results.

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

```
matchAll [1,2,3] (Multiset Something)
  [[mc| cons $x $xs => (x,xs) |]]
-- [(1,[2,3]),(2,[1,3]),(3,[1,2])]
```

- a: the type of the target.
- m: the type of the matcher.
- b: the type of the body of match clause.

Typing the matchAll expression — matchers

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

```
class Matcher m a
```

- Matcher is a type class with no methods.

```
data Eql = Eql
instance (Eq a) => Matcher Eql a
```

- The name of the type and data constructor of `Eql` are identical.
- This instance declaration asserts that `Eql` is a matcher for `a` that is an instance of `Eq`.

Typing the matchAll expression — matchers

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

```
class Matcher m a
```

- Matcher is a type class with no methods.

```
data Eql = Eql
instance (Eq a) => Matcher Eql a
```

- A multiple type class is effectively used to describe the relation between a matcher and a type of target data.

- The name of the type and data constructor of `Eql` are identical.
- This instance declaration asserts that `Eql` is a matcher for `a` that is an instance of `Eq`.

Typing the matchAll expression — match clauses

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

```
data MatchClause a m b = forall vs. (Matcher m a)
                                  => MatchClause (Pattern a m '[] vs)
                                                 (HList vs -> b)
```

```
[mc| cons $x $xs => (x,xs) |]
```

```
MatchClause (cons (PatVar "x") (PatVar "xs"))
            (\HCons x (HCons xs HNil) -> (x,xs))
```

Typing the matchAll expression — match clauses

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

- `vs` is the type of the pattern-matching results.

```
data MatchClause a m b = forall vs. (Matcher m a)
                              => MatchClause (Pattern a m '[] vs)
                                             (HList vs -> b)
```

- `vs` is a list of types (a type-level list).

```
[mc| cons $x $xs => (x,xs) |]
```

⇩

```
MatchClause (cons (PatVar "x") (PatVar "xs"))
            (\HCons x (HCons xs HNil) -> (x,xs))
```

Typing the matchAll expression — match clauses

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

- The type `vs` is existentially quantified.

```
data MatchClause a m b = forall vs. (Matcher m a)
                                 => MatchClause (Pattern a m '[] vs)
                                                (HList vs -> b)
```

- This is because each pattern of the match clauses in the same pattern-matching expression generally makes different bindings.

```
[mc| cons $x $xs => (x,xs) |]
```

```
MatchClause (cons (PatVar "x") (PatVar "xs"))
            (\HCons x (HCons xs HNil) -> (x,xs))
```

Typing the matchAll expression — match clauses

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

- The type `vs` is existentially quantified.

```
data MatchClause a m b = forall vs. (Matcher m a)
                               => MatchClause (Pattern a m '[] vs)
                                              (HList vs -> b)
```

- This is because each pattern of the match clauses in the same pattern-matching expression generally makes different bindings.

```
[mc| cons $x $xs => (x,xs) |]
```

- Existential types are effectively used to hide `vs` from the type of `MatchClause`.

```
MatchClause (cons (PatVar "x") (PatVar "xs"))
            (\HCons x (HCons xs HNil) -> (x,xs))
```

Typing the matchAll expression — match clauses

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

```
data MatchClause a m b = forall vs. (Matcher m a)
                             => MatchClause (Pattern a m '[] vs)
                                            (HList vs -> b)
```

- `mc` **is a quasi-quoter of Template Haskell for describing match clause concisely.**

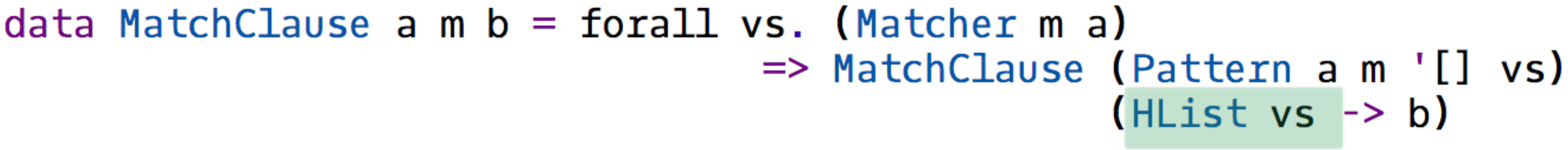
```
[mc| cons $x $xs => (x,xs) |]
```

⇩

```
MatchClause (cons (PatVar "x") (PatVar "xs"))
            (\HCons x (HCons xs HNil) -> (x,xs))
```

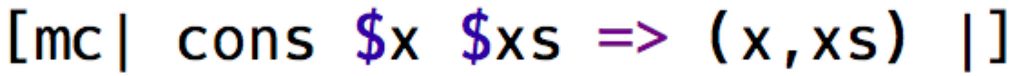
Typing the matchAll expression — match clauses

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

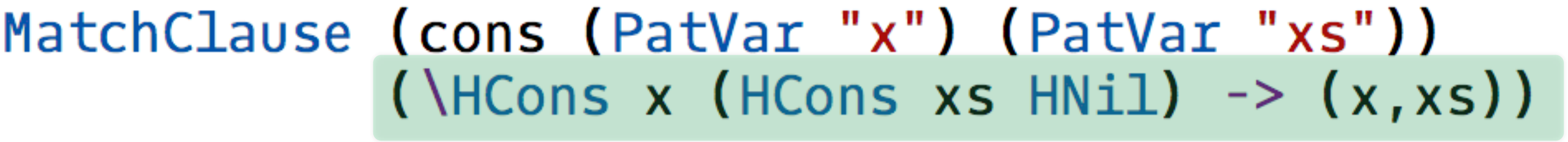


```
data MatchClause a m b = forall vs. (Matcher m a)
                                  => MatchClause (Pattern a m '[] vs)
                                                 (HList vs -> b)
```

- mc is a quasi-quoter of Template Haskell for describing match clause concisely.
- The body of match clause is transformed to a function that takes a heterogeneous list.



```
[mc| cons $x $xs => (x,xs) |]
```



```
MatchClause (cons (PatVar "x") (PatVar "xs"))
            (\HCons x (HCons xs HNil) -> (x,xs))
```

Typing the matchAll expression — match clauses

```
matchAll :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
```

```
data MatchClause a m b = forall vs. (Matcher m a)
                                   => MatchClause (Pattern a m '[] vs)
                                                  (HList vs -> b)
```

- **Template Haskell is effectively used for concise notation of match clauses.**

```
[mc| cons $x $xs => (x,xs) |]
```

⇩

```
MatchClause (cons (PatVar "x") (PatVar "xs"))
            (\HCons x (HCons xs HNil) -> (x,xs))
```

- `mc` **is a quasi-quoter of Template Haskell for describing match clause concisely.**
- **The body of match clause is transformed to a function that takes a heterogeneous list.**

Typing the matchAll expression — patterns

```
data MatchClause a m b = forall vs. (Matcher m a)
                                 => MatchClause (Pattern a m '[] vs)
                                                (HList vs -> b)

data Pattern a m ctx vs where
  Wildcard :: (Matcher m a)
           => Pattern a m ctx '[]
  PatVar   :: (Matcher m a)
           => String
           -> Pattern a m ctx '[a]
  Pattern  :: (Matcher m a)
           => (HList ctx -> m -> a -> [MList ctx vs])
           -> Pattern a m ctx vs
```

- `ctx`: **the type of the intermediate pattern-matching result** (the values bound to the pattern variables in the left-side of the pattern).
- `vs`: **the type of the values bound to the pattern variables appear in this pattern.**

Typing the matchAll expression — patterns

```
data MatchClause a m b = forall vs. (Matcher m a)
                                 => MatchClause (Pattern a m '[] vs)
                                                (HList vs -> b)

data Pattern a m ctx vs where
  Wildcard :: (Matcher m a)
           => Pattern a m ctx '[]
  PatVar   :: (Matcher m a)
           => String
           -> Pattern a m ctx '[a]
  Pattern  :: (Matcher m a)
           => (HList ctx -> m -> a -> [MList ctx vs])
           -> Pattern a m ctx vs
```

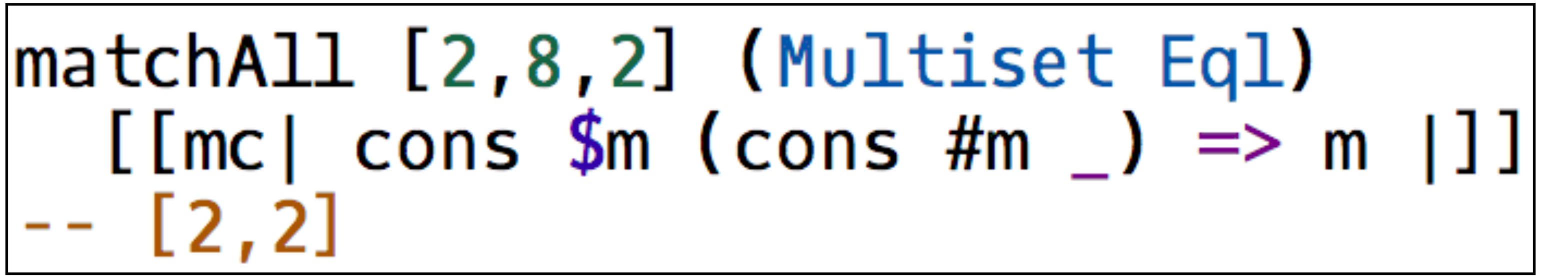
- **Datatype promotion (`DataKinds` extension) is effectively used to represent the type of pattern-matching results.**
- **`‘[]` represent a type of the empty list.**
- **`‘[a]` represent a type of the list that contains single element `a`.**

- **GADTs allow `vs` (the 4th parameter of `Pattern`) changes for each data constructor (`vs` of `Wildcard` is `‘[]`, `vs` of `PatVar` is `‘[a]`).**

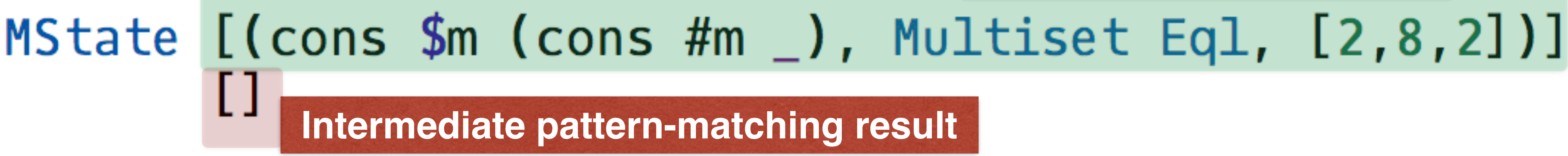
Today’s Contents

- Tutorial of MiniEgison
- Background
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- **Implementation of MiniEgison**
  - Typing MatchAll
  - **Typing Matching States and Matching Atoms**
  - User-Defined Pattern-Matching Algorithms
- Performance
- Conclusion

Overview of the pattern-matching algorithm inside miniEgison



```
matchAll [2,8,2] (Multiset Eql)
  [[mc| cons $m (cons #m _) => m |]]
-- [2,2]
```



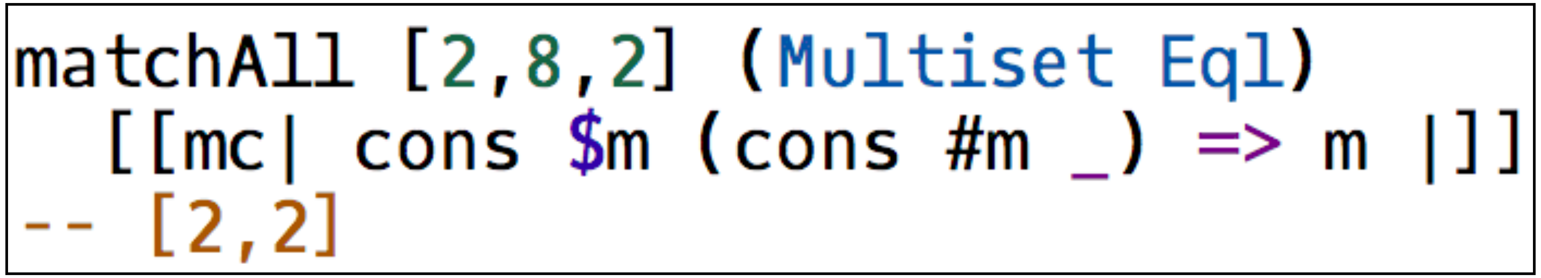
```
MState [(cons $m (cons #m _), Multiset Eql, [2,8,2])]
       []
```

Stack of matching atoms

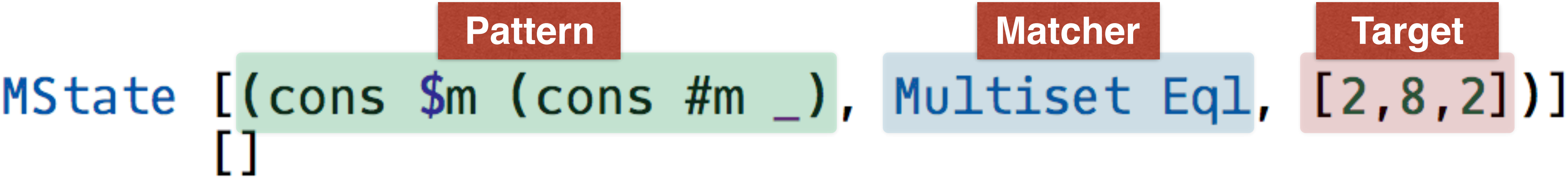
Intermediate pattern-matching result

Pattern matching algorithm is defined as reduction of matching states.

Overview of the pattern-matching algorithm inside miniEgison



```
matchAll [2,8,2] (Multiset Eql)
  [[mc| cons $m (cons #m _) => m |]]
-- [2,2]
```



```
MState [(cons $m (cons #m _), Multiset Eql, [2,8,2])]
       []
```

A matching atom consists of a pattern, matcher, and target.

Overview of the pattern-matching algorithm inside miniEgison

```
MState [(cons $m (cons #m _), Multiset Eql, [2,8,2])]
       []
```

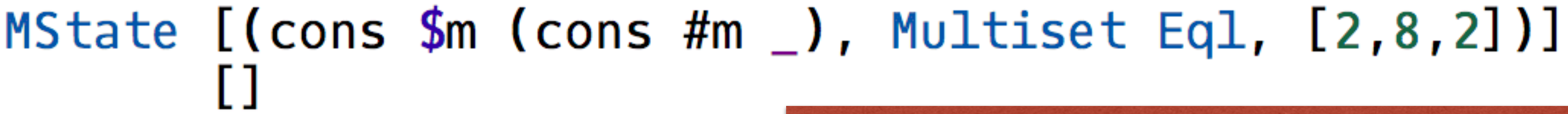
Next matching states are generated from the definition of cons in the multiset matcher.

1
```
MState [($m, Eql, 2), (cons #m _, Multiset Eql, [8, 2])]
       []
```

2
```
MState [($m, Eql, 8), (cons #m _, Multiset Eql, [2, 2])]
       []
```

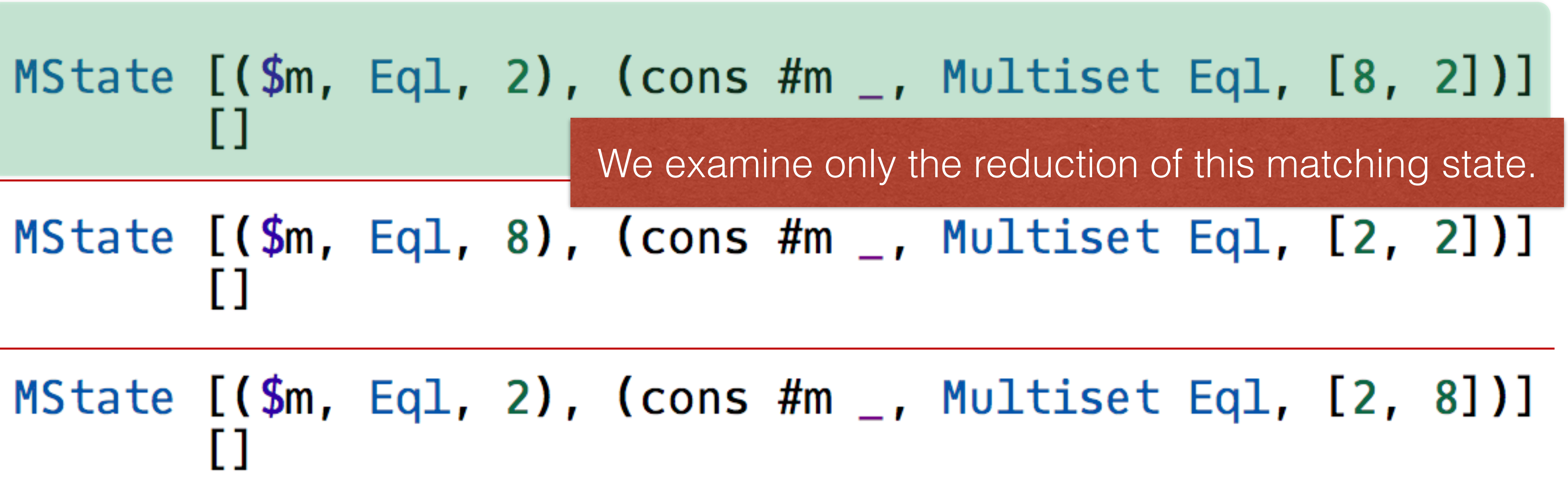
3
```
MState [($m, Eql, 2), (cons #m _, Multiset Eql, [2, 8])]
       []
```

Overview of the pattern-matching algorithm inside miniEgison



```
MState [(cons $m (cons #m _), Multiset Eql, [2,8,2])]
       []
```

Next matching states are generated from the definition of cons in the multiset matcher.



1
```
MState [($m, Eql, 2), (cons #m _, Multiset Eql, [8, 2])]
       []
```

We examine only the reduction of this matching state.

2
```
MState [($m, Eql, 8), (cons #m _, Multiset Eql, [2, 2])]
       []
```

3
```
MState [($m, Eql, 2), (cons #m _, Multiset Eql, [2, 8])]
       []
```

Overview of the pattern-matching algorithm inside miniEgison

```
MState [($m, Eql, 2), (cons #m _, Multiset Eql, [8, 2])]
       []
```

The target **2** is added to the intermediate pattern-matching result because the pattern is a pattern variable.

```
MState [(cons #m _, Multiset Eql, [8, 2])]
       [2]
```

Next matching states are generated from the definition of cons in the multiset matcher.

1
```
MState [(#m, Eql, 8), (_, Multiset Eql, [2])]
       [2]
```

This matching state fails to pattern-match and vanishes.

2
```
MState [(#m, Eql, 2), (_, Multiset Eql, [8])]
       [2]
```

Overview of the pattern-matching algorithm inside miniEgison

```
MState [($m, Eql, 2), (cons #m _, Multiset Eql, [8, 2])]
       []
```

The target **2** is added to the intermediate pattern-matching result because the pattern is a pattern variable.

```
MState [(cons #m _, Multiset Eql, [8, 2])]
       [2]
```

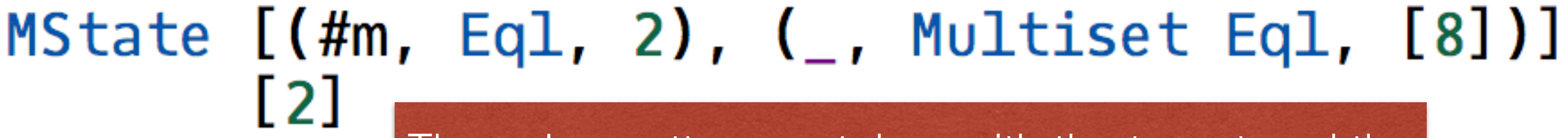
Next matching states are generated from the definition of cons in the multiset matcher.

1
```
MState [(#m, Eql, 8), (_, Multiset Eql,
       [2]
```

2
```
MState [(#m, Eql, 2), (_, Multiset Eql, [8])]
       [2]
```

We examine the reduction of this matching state.

Overview of the pattern-matching algorithm inside miniEgison

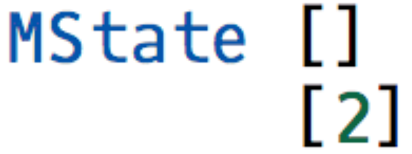


```
MState [(#m, Eql, 2), (_, Multiset Eql, [8])]
       [2]
```

The value pattern matches with the target and the matching atom is popped off.

```
MState [(_, Multiset Eql, [8])]
       [2]
```

The wildcard matches with any target.



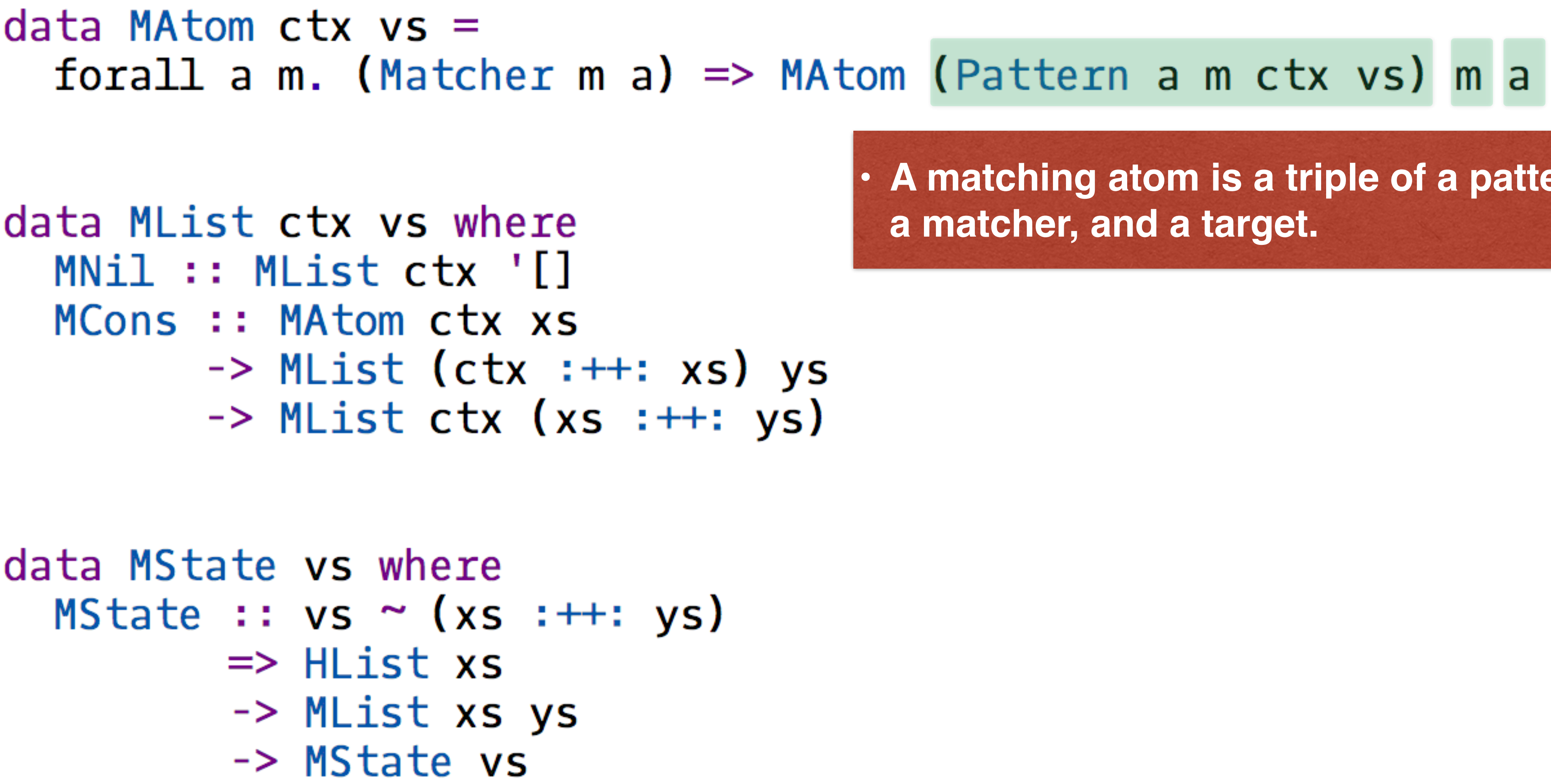
```
MState []
       [2]
```

If the stack of matching atoms becomes empty, pattern matching succeeds.

Typing matching atoms and matching states

```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
         => HList xs
         -> MList xs ys
         -> MState vs
```

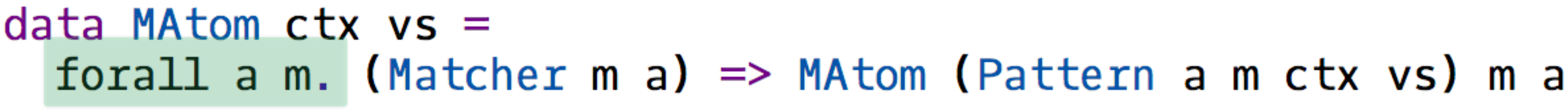
Typing matching atoms and matching states



```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
         => HList xs
         -> MList xs ys
         -> MState vs
```

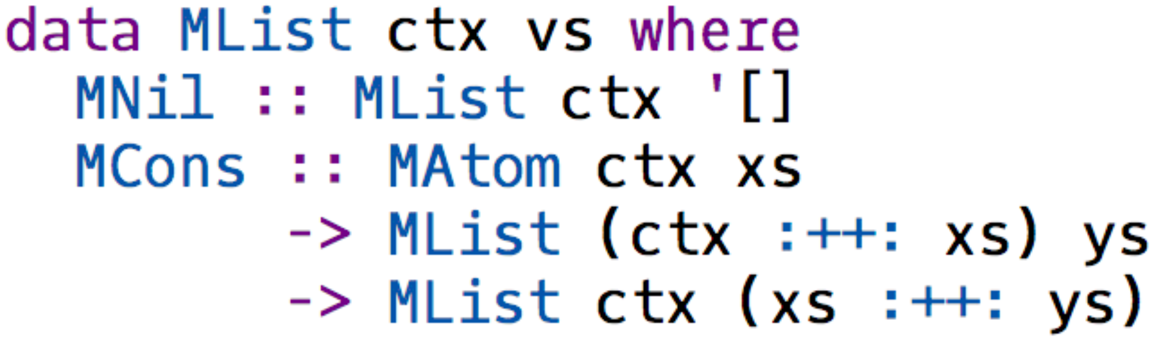
- A matching atom is a triple of a pattern, a matcher, and a target.

Typing matching atoms and matching states

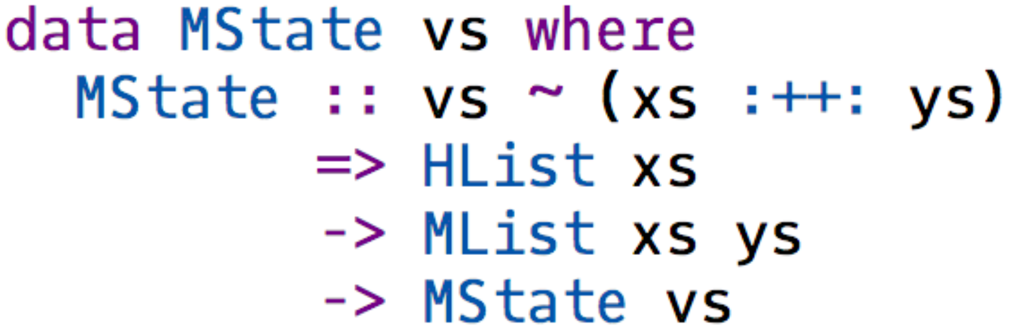


```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a
```

- `a` and `m` are existentially quantified.
- The reason is because the types of the targets of the matching atoms in a stack of matching atoms are generally different.

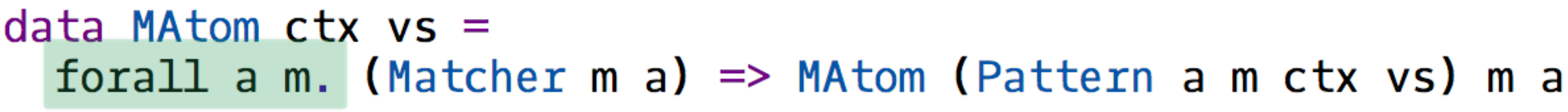


```
data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)
```

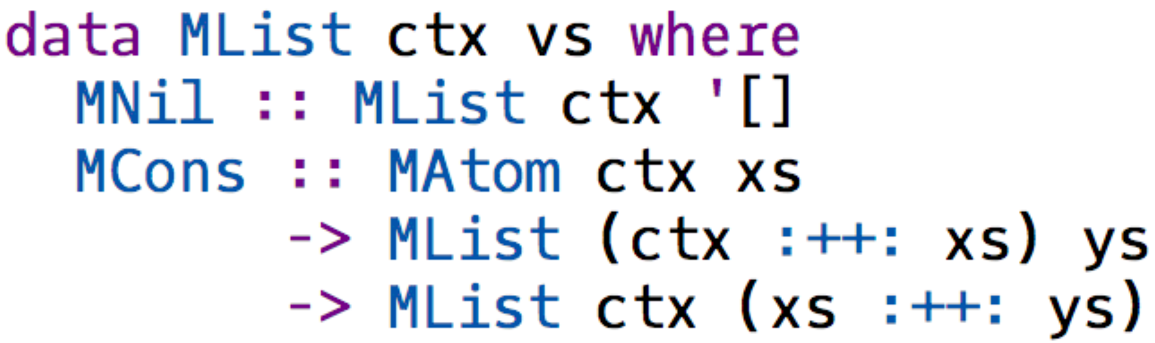


```
data MState vs where
  MState :: vs ~ (xs :++: ys)
          => HList xs
          -> MList xs ys
          -> MState vs
```

Typing matching atoms and matching states

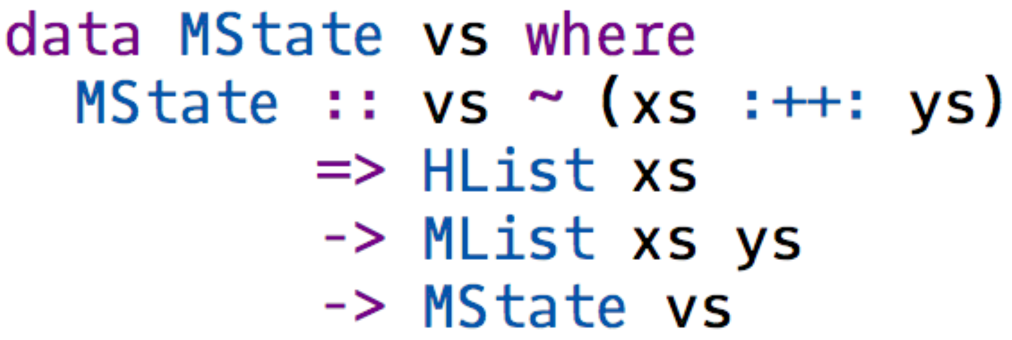


```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a
```



```
data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)
```

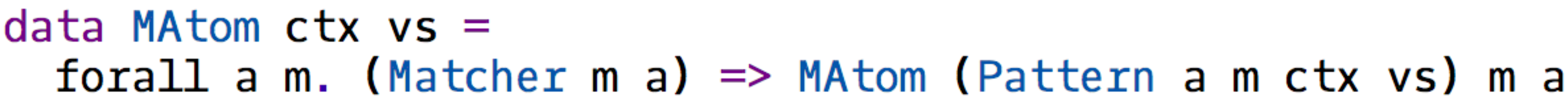
- `a` and `m` are existentially quantified.
- The reason is because the types of the targets of the matching atoms in a stack of matching atoms are generally different.



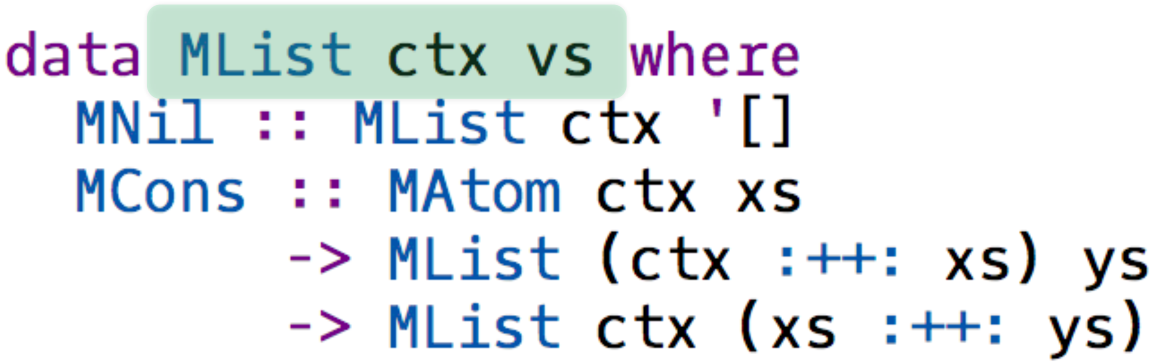
```
data MState vs where
  MState :: vs ~ (xs :++: ys)
         => HList xs
         -> MList xs ys
         -> MState vs
```

- Existential types are effectively used to hide `a` and `m` from the type of `MAtom`.

Typing matching atoms and matching states

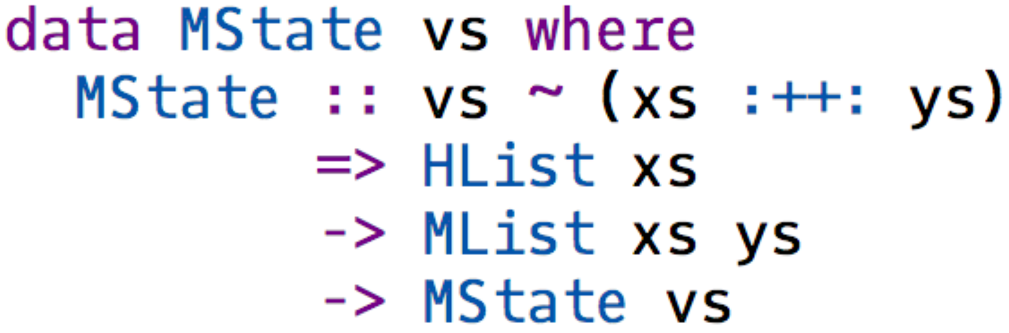


```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a
```



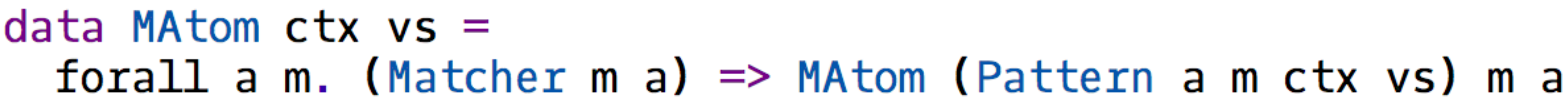
```
data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)
```

- `MList` **is a datatype for a stack of matching atoms.**
- `ctx`**: the type of intermediate pattern-matching results.**
- `vs`**: the type of the values bound by processing this** `MList` **itself.**

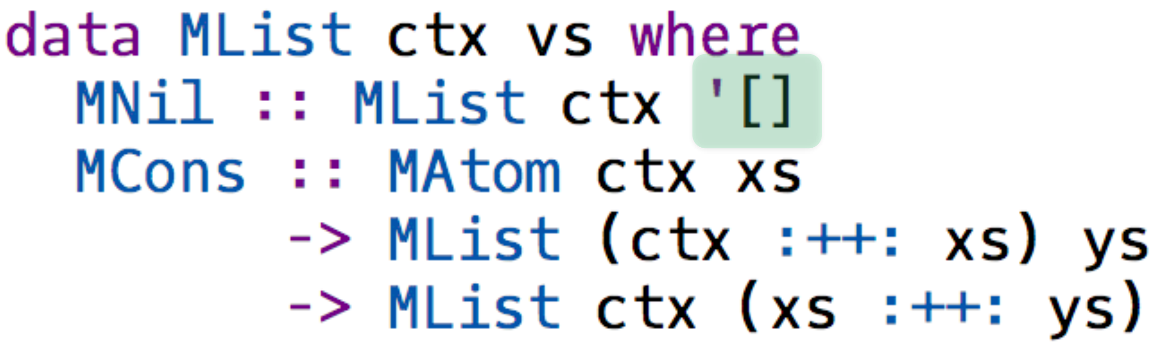


```
data MState vs where
  MState :: vs ~ (xs :++: ys)
          => HList xs
          -> MList xs ys
          -> MState vs
```

Typing matching atoms and matching states

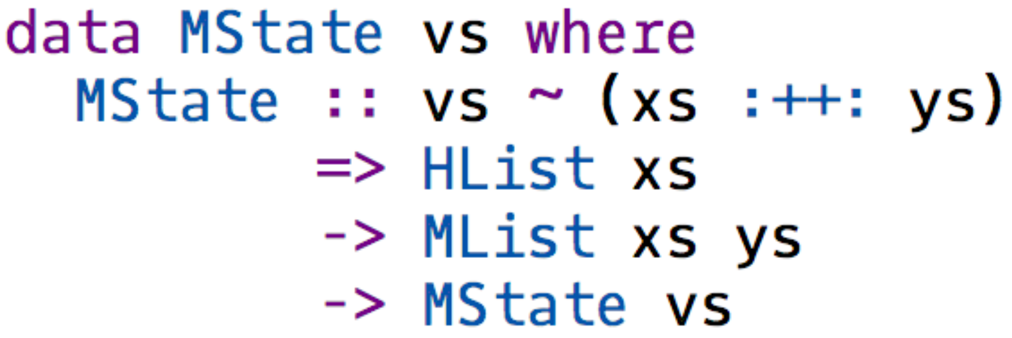


```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a
```



```
data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)
```

- `MList` **is a datatype for a stack of matching atoms.**
- `ctx`**: the type of intermediate pattern-matching results.**
- `vs`**: the type of the values bound by processing this** `MList` **itself.**



```
data MState vs where
  MState :: vs ~ (xs :++: ys)
          => HList xs
          -> MList xs ys
          -> MState vs
```

Typing matching atoms and matching states

```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
          => HList xs
          -> MList xs ys
          -> MState vs
```

- `MList` **is a datatype for a stack of matching atoms.**
- `ctx`**: the type of intermediate pattern-matching results.**
- `vs`**: the type of the values bound by processing this** `MList` **itself.**
  - `vs` **of** `MNil` **is an empty list.**

Typing matching atoms and matching states

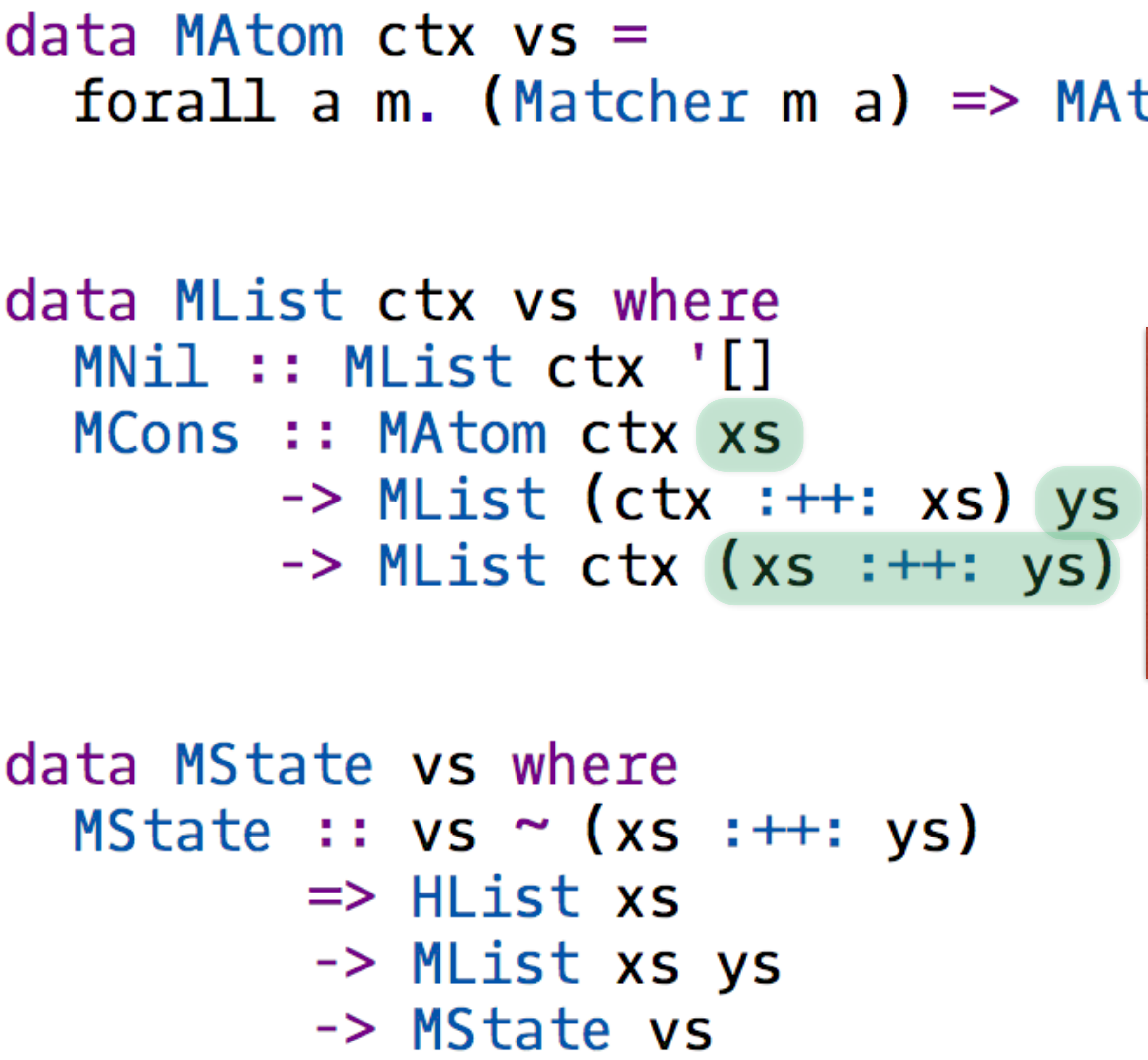
```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
          => HList xs
          -> MList xs ys
          -> MState vs
```

Typing matching atoms and matching states

```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
         => HList xs
         -> MList xs ys
         -> MState vs
```

- The pattern-matching result of the first matching atom is appended to the intermediate result of the rest matching atoms.
- `:++:` is a type operator for the type-level append operation.

Typing matching atoms and matching states



```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
         => HList xs
         -> MList xs ys
         -> MState vs
```

- **The concatenation of the pattern-matching result (`xs`) of the first matching atom and the rest matching atoms (`ys`) are the result of the whole stack of matching atoms.**

Typing matching atoms and matching states

```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
         => HList xs
         -> MList xs ys
         -> MState vs
```

- **Datatype promotion (**`DataKinds` **extension) is effectively used here.**
- `'[]` **represent a type of the empty list.**
- `:++:` **is defined using** `TypeFamily`**.**

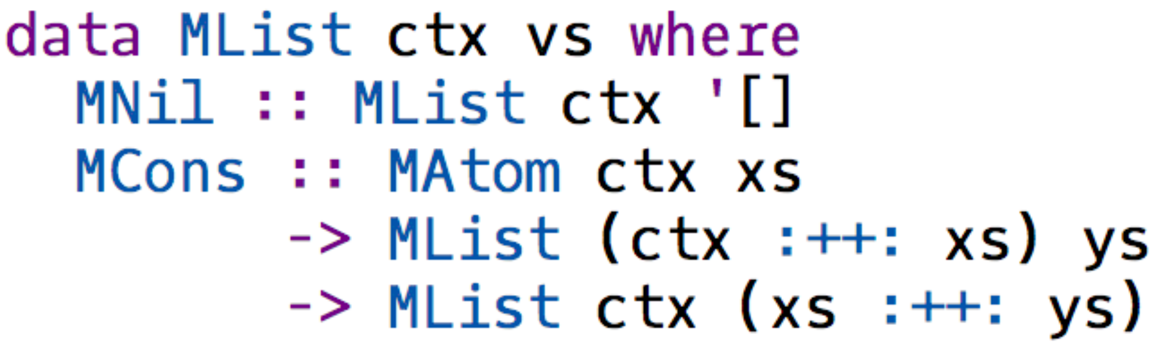
Typing matching atoms and matching states

```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
          => HList xs
          -> MList xs ys
          -> MState vs
```

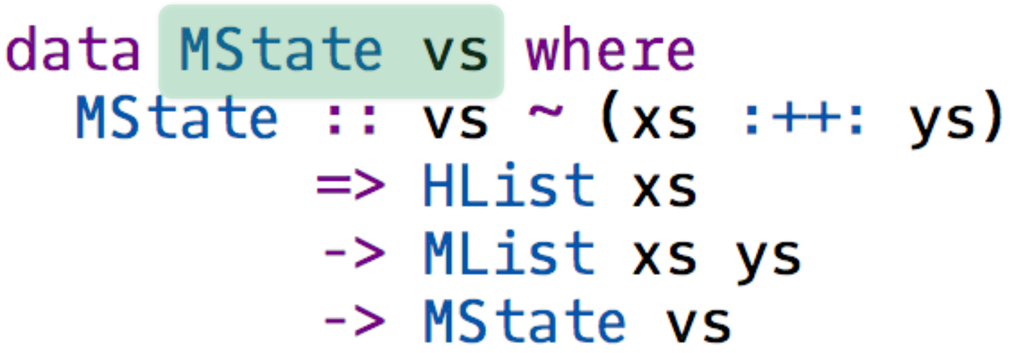
- **GADTs allow** `vs` **(the 2nd parameter of** `MList`**) changes for each data constructor (**`vs` **of** `MNil` **is** `'[]`**,** `vs` **of** `MCons` **is** `xs :++: ys`**).**

Typing matching atoms and matching states

```haskell
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a
```



```haskell
data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)
```



```haskell
data MState vs where
  MState :: vs ~ (xs :++: ys)
         => HList xs
         -> MList xs ys
         -> MState vs
```

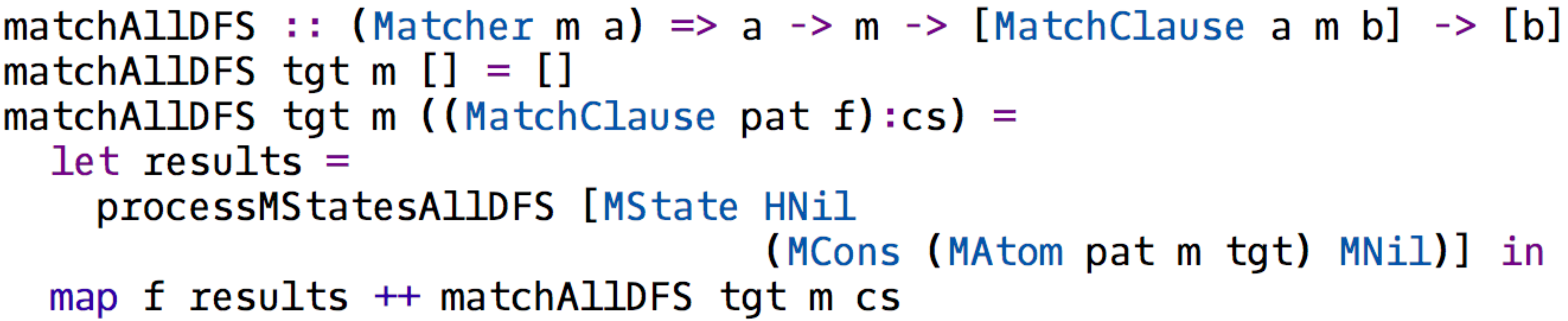
- `vs` **is a type of the final pattern-matching result.**

Typing matching atoms and matching states

```
data MAtom ctx vs =
  forall a m. (Matcher m a) => MAtom (Pattern a m ctx vs) m a


data MList ctx vs where
  MNil :: MList ctx '[]
  MCons :: MAtom ctx xs
        -> MList (ctx :++: xs) ys
        -> MList ctx (xs :++: ys)


data MState vs where
  MState :: vs ~ (xs :++: ys)
         => HList xs
         -> MList xs ys
         -> MState vs
```

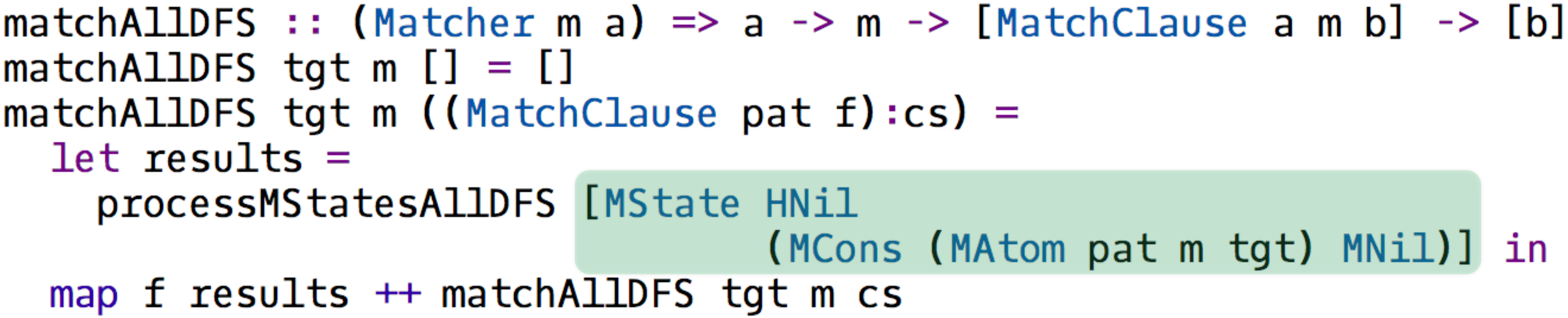
- `MState` **takes an intermediate pattern-matching results (**`HList xs`**) and a stack of matching atoms (**`MList xs ys`**).**
- `xs :++: ys` **is the final pattern-matching results:**

matchAllDFS



```haskell
matchAllDFS :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
matchAllDFS tgt m [] = []
matchAllDFS tgt m ((MatchClause pat f):cs) =
  let results =
    processMStatesAllDFS [MState HNil
                                 (MCons (MAtom pat m tgt) MNil)] in
  map f results ++ matchAllDFS tgt m cs
```

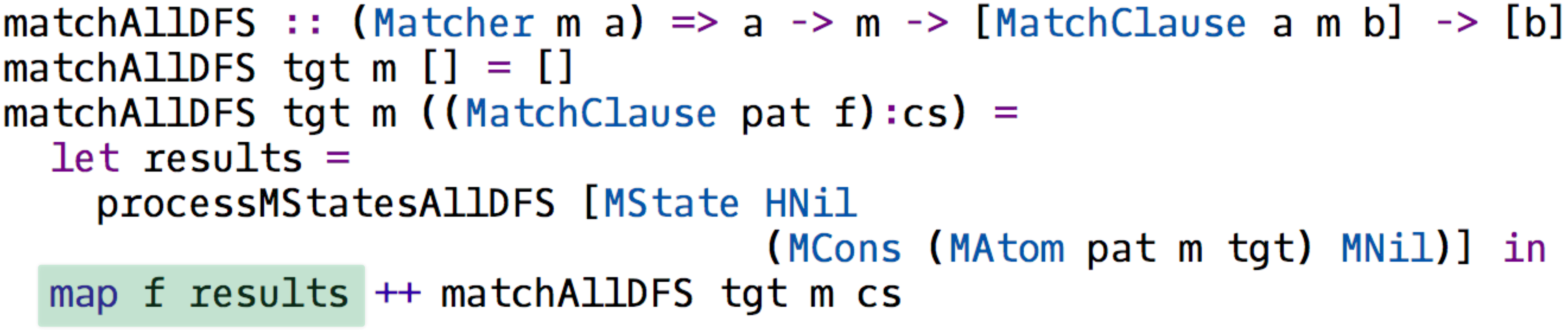
matchAllDFS



```
matchAllDFS :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
matchAllDFS tgt m [] = []
matchAllDFS tgt m ((MatchClause pat f):cs) =
  let results =
    processMStatesAllDFS [MState HNil
                                 (MCons (MAtom pat m tgt) MNil)] in
  map f results ++ matchAllDFS tgt m cs
```

- **The initial matching state is created.**

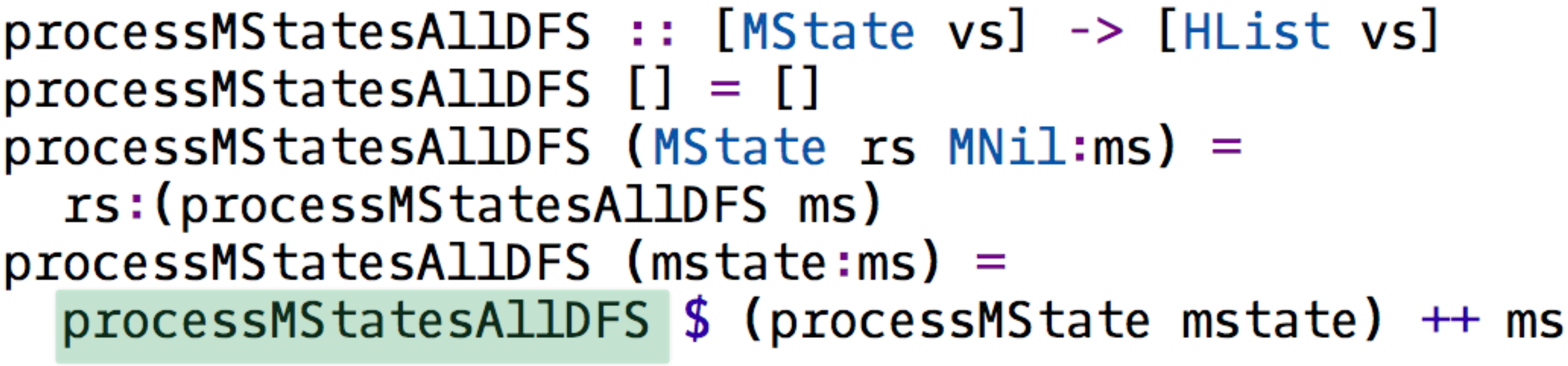
matchAllDFS



```haskell
matchAllDFS :: (Matcher m a) => a -> m -> [MatchClause a m b] -> [b]
matchAllDFS tgt m [] = []
matchAllDFS tgt m ((MatchClause pat f):cs) =
  let results =
    processMStatesAllDFS [MState HNil
                                    (MCons (MAtom pat m tgt) MNil)] in
  map f results ++ matchAllDFS tgt m cs
```

- **Each pattern-matching result is mapped to the body of match clause.**

matchAllDFS — processMStatesAllDFS



```haskell
processMStatesAllDFS :: [MState vs] -> [HList vs]
processMStatesAllDFS [] = []
processMStatesAllDFS (MState rs MNil:ms) =
  rs:(processMStatesAllDFS ms)
processMStatesAllDFS (mstate:ms) =
  processMStatesAllDFS $ (processMState mstate) ++ ms
```

- The main loop is tail-recursive.
- It is important for execution performance.

matchAllDFS — processMState

```
processMState :: MState vs -> [MState vs]
processMState (MState rs (MCons (MAtom pat m tgt) atoms)) =
  case pat of
    Wildcard -> [MState rs atoms]
    PatVar _ -> case patVarProof rs (HCons tgt HNil) atoms of
                  Refl -> [MState (happend rs (HCons tgt HNil)) atoms]
    Pattern p ->
      let matomss = p rs m tgt in
      map (\newAtoms -> MState rs (mappend newAtoms atoms)) matomss
```

Today’s Contents

- Tutorial of MiniEgison
- Background
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- **Implementation of MiniEgison**
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - **User-Defined Pattern-Matching Algorithms**
- Performance
- Conclusion

UnorderedPair

```
matchAll (1,2) (UnorderedPair Eql)
  [[mc| upair #2 $x => x |]]
-- [1]
```

```
data UnorderedPair m = UnorderedPair m
instance Matcher m a => Matcher (UnorderedPair m) (a, a)
```

```
upair :: (Matcher m a, a ~ (b, b), m ~ (UnorderedPair m'), Matcher m' b)
      => Pattern b m' ctx xs
      -> Pattern b m' (ctx :++: xs) ys
      -> Pattern a m ctx (xs :++: ys)

upair p1 p2 = Pattern (\_ (UnorderedPair m') (t1, t2) ->
                         [twoMAtoms (MAtom p1 m' t1) (MAtom p2 m' t2)
                         ,twoMAtoms (MAtom p1 m' t2) (MAtom p2 m' t1)])
```

UnorderedPair

```
matchAll (1,2) (UnorderedPair Eql)
  [[mc| upair #2 $x => x |]]
-- [1]
```

- **The pattern for 2-tuples for which we ignore the order of elements.**

```
data UnorderedPair m = UnorderedPair m
instance Matcher m a => Matcher (UnorderedPair m) (a, a)
```

```
upair :: (Matcher m a, a ~ (b, b), m ~ (UnorderedPair m'), Matcher m' b)
      => Pattern b m' ctx xs
      -> Pattern b m' (ctx :++: xs) ys
      -> Pattern a m ctx (xs :++: ys)

upair p1 p2 = Pattern (\_ (UnorderedPair m') (t1, t2) ->
                         [twoMAtoms (MAtom p1 m' t1) (MAtom p2 m' t2)
                         ,twoMAtoms (MAtom p1 m' t2) (MAtom p2 m' t1)])
```

UnorderedPair

```
matchAll (1,2) (UnorderedPair Eql)
  [[mc| upair #2 $x => x |]]
-- [1]
```

- `upair` **is a function that takes patterns and return a pattern.**

```
data UnorderedPair m = UnorderedPair m
instance Matcher m a => Matcher (UnorderedPair m) (a, a)
```

```
upair :: (Matcher m a, a ~ (b, b), m ~ (UnorderedPair m'), Matcher m' b)
      => Pattern b m' ctx xs
      -> Pattern b m' (ctx :++: xs) ys
      -> Pattern a m ctx (xs :++: ys)
```

```
upair p1 p2 = Pattern (\_ (UnorderedPair m') (t1, t2) ->
                        [twoMAtoms (MAtom p1 m' t1) (MAtom p2 m' t2)
                        ,twoMAtoms (MAtom p1 m' t2) (MAtom p2 m' t1)])
```

UnorderedPair

```
matchAll (1,2) (UnorderedPair Eql)
  [[mc| upair #2 $x => x |]]
-- [1]
```

- **Let's look into the definition of `upair`.**

```
data UnorderedPair m = UnorderedPair m
instance Matcher m a => Matcher (UnorderedPair m) (a, a)
```

```
upair :: (Matcher m a, a ~ (b, b), m ~ (UnorderedPair m'), Matcher m' b)
      => Pattern b m' ctx xs
      -> Pattern b m' (ctx :++: xs) ys
      -> Pattern a m ctx (xs :++: ys)
```

```
upair p1 p2 = Pattern (\_ (UnorderedPair m') (t1, t2) ->
                         [twoMAtoms (MAtom p1 m' t1) (MAtom p2 m' t2)
                         ,twoMAtoms (MAtom p1 m' t2) (MAtom p2 m' t1)])
```

UnorderedPair

```
data Pattern a m ctx vs where
  Wildcard :: (Matcher m a)
           => Pattern a m ctx '[]
  PatVar   :: (Matcher m a)
           => String
           -> Pattern a m ctx '[a]
  Pattern  :: (Matcher m a)
           => (HList ctx -> m -> a -> [MList ctx vs])
           -> Pattern a m ctx vs
```

- **The `Pattern` data constructor is used to define a user-defined patterns.**

```
upair p1 p2 = Pattern (\_ (UnorderedPair m') (t1, t2) ->
                 [twoMAtoms (MAtom p1 m' t1) (MAtom p2 m' t2)
                 ,twoMAtoms (MAtom p1 m' t2) (MAtom p2 m' t1)])
```

UnorderedPair

- `Pattern` **takes a function that take**
  - **a intermediate pattern-matching result,**
  - **a matcher, and**
  - **a target,**
- **and returns**
  - **a list of lists of matching atoms.**

```
data Pattern a m ctx vs where
  Wildcard :: (Matcher m a)
           => Pattern a m ctx '[]
  PatVar   :: (Matcher m a)
           => String
           -> Pattern a m ctx '[a]
  Pattern  :: (Matcher m a)
           => (HList ctx -> m -> a -> [MList ctx vs])
           -> Pattern a m ctx vs
```

```
upair p1 p2 = Pattern (\_ (UnorderedPair m') (t1, t2) ->
                         [twoMAtoms (MAtom p1 m' t1) (MAtom p2 m' t2)
                         ,twoMAtoms (MAtom p1 m' t2) (MAtom p2 m' t1)])
```

UnorderedPair

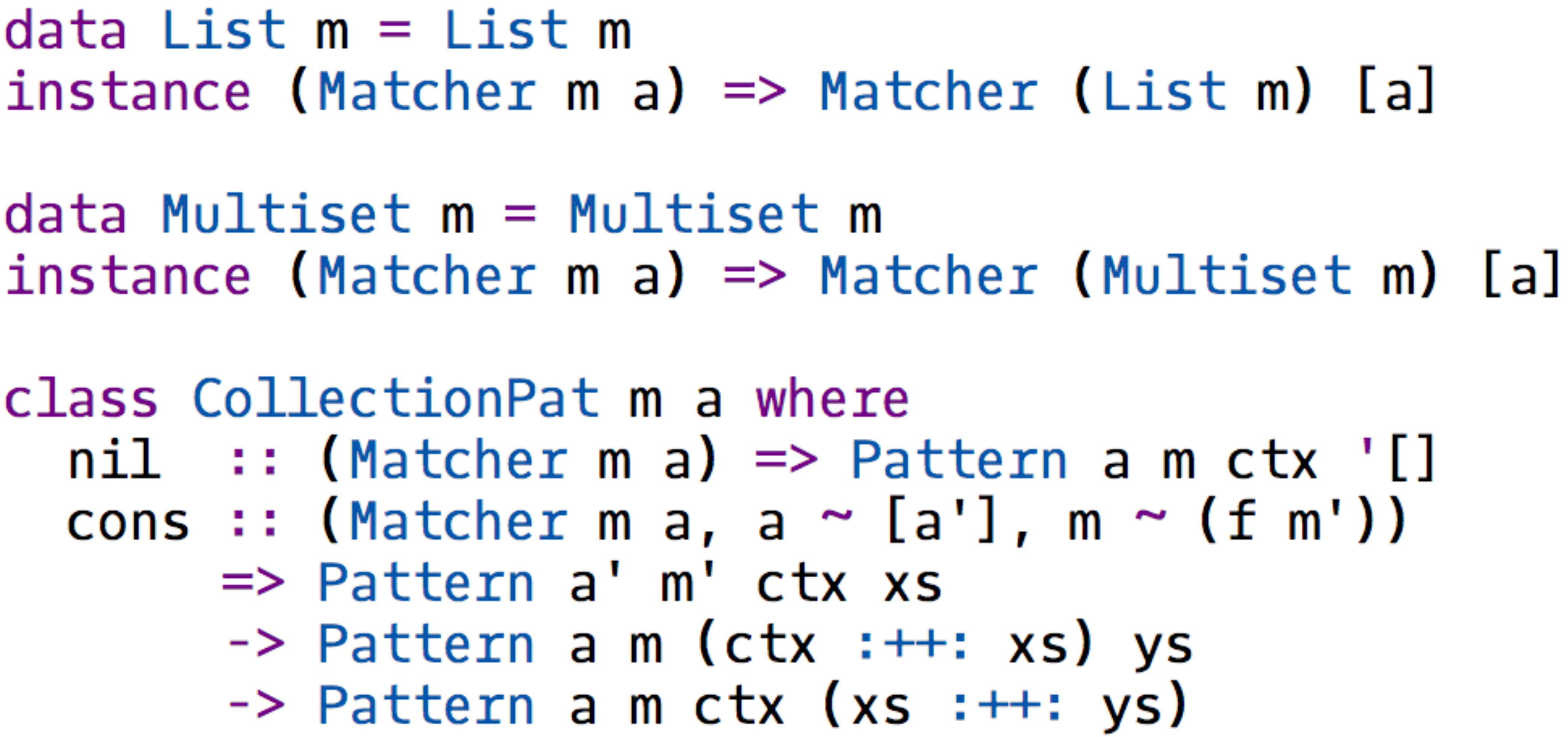
```
data Pattern a m ctx vs where
  Wildcard :: (Matcher m a)
           => Pattern a m ctx '[]
  PatVar   :: (Matcher m a)
           => String
           -> Pattern a m ctx '[a]
  Pattern  :: (Matcher m a)
           => (HList ctx -> m -> a -> [MList ctx vs])
           -> Pattern a m ctx vs
```

- `Pattern` takes a function that take
  - a intermediate pattern-matching result,
  - a matcher, and
  - a target,
- and returns
  - a list of lists of matching atoms.

```
upair p1 p2 = Pattern (\_ (UnorderedPair m') (t1, t2) ->
                [twoMAtoms (MAtom p1 m' t1) (MAtom p2 m' t2)
                ,twoMAtoms (MAtom p1 m' t2) (MAtom p2 m' t1)])
```

- `twoMAtoms` is a utility function to create an `MList` that consists of two matching atoms.

List and Multiset



```
data List m = List m
instance (Matcher m a) => Matcher (List m) [a]

data Multiset m = Multiset m
instance (Matcher m a) => Matcher (Multiset m) [a]

class CollectionPat m a where
  nil  :: (Matcher m a) => Pattern a m ctx '[]
  cons :: (Matcher m a, a ~ [a'], m ~ (f m'))
       => Pattern a' m' ctx xs
       -> Pattern a m (ctx :++: xs) ys
       -> Pattern a m ctx (xs :++: ys)
```

List and Multiset

- `CollectionPat` **is a type class for changing the meaning of** `nil` **and** `cons` **for each matcher.**

```
class CollectionPat m a where
  nil  :: (Matcher m a) => Pattern a m ctx '[]
  cons :: (Matcher m a, a ~ [a'], m ~ (f m'))
       => Pattern a' m' ctx xs
       -> Pattern a m (ctx :++: xs) ys
       -> Pattern a m ctx (xs :++: ys)

instance (Matcher m a) => CollectionPat (Multiset m) [a] where
  nil = Pattern (\_ _ tgt -> [MNil | null tgt])
  cons p1 p2 =
    Pattern (\_ (Multiset m) tgt ->
               map (\(x, xs) -> twoMAtoms (MAtom p1 m x)
                                          (MAtom p2 (Multiset m) xs))
                   (matchAll tgt (List m)
                      [[mc| join $hs (cons $x $ts) => (x, hs ++ ts) |]]))
```

Value Patterns

```
class ValuePat m a where
  valuePat :: (Matcher m a, Eq a) => (HList ctx -> a) -> Pattern a m ctx '[]
```

- The pattern constructor of the value patterns are defined as a method of type class.
- This is because ad-hoc polymorphism is important for value patterns.

```
data Eql = Eql
instance (Eq a) => Matcher Eql a
```

```
instance Eq a => ValuePat Eql a where
  valuePat f = Pattern (\ctx _ tgt -> [MNil | f ctx == tgt])
```

```
instance (Matcher m a, Eq a, ValuePat m a) => ValuePat (Multiset m) [a] where
  valuePat f = Pattern (\ctx (Multiset m) tgt ->
                          match (f ctx, tgt) (Pair (List m) (Multiset m)) $
                            [[mc| pair nil nil => [MNil] |],
                             [mc| pair (cons $x $xs) (cons #x #xs) => [MNil] |],
                             [mc| Wildcard => [] |]]
```

Value Patterns

```
class ValuePat m a where
  valuePat :: (Matcher m a, Eq a) => (HList ctx -> a) -> Pattern a m ctx '[]
```

- `valuePat` **takes a function that takes an intermediate pattern-matching result.**

```
data Eql = Eql
instance (Eq a) => Matcher Eql a


instance Eq a => ValuePat Eql a where
  valuePat f = Pattern (\ctx _ tgt -> [MNil | f ctx == tgt])


instance (Matcher m a, Eq a, ValuePat m a) => ValuePat (Multiset m) [a] where
  valuePat f = Pattern (\ctx (Multiset m) tgt ->
                              match (f ctx, tgt) (Pair (List m) (Multiset m)) $
                                [[mc| pair nil nil => [MNil] |],
                                 [mc| pair (cons $x $xs) (cons #x #xs) => [MNil] |],
                                 [mc| Wildcard => [] |]]
```

Value Patterns

- **A value pattern is rewritten to the function that takes an intermediate pattern-matching result.**

```
[mc| cons $x (cons $y (cons #(x + 1) (cons $z nil))) => (x, y, z) |]
```

```
MatchClause (cons (PatVar "x")
             (cons (PatVar "y")
              (cons (ValuePat (\HCons x (HCons (y HNil)) -> x + 1))
               (cons (PatVar "z") nil))))
            (\HCons x (HCons (y (HCons z HNil))) -> (x, y, z))
```

Value Patterns

```
class ValuePat m a where
  valuePat :: (Matcher m a, Eq a) => (HList ctx -> a) -> Pattern a m ctx '[]

data Eql = Eql
instance (Eq a) => Matcher Eql a

instance Eq a => ValuePat Eql a where
  valuePat f = Pattern (\ctx _ tgt -> [MNil | f ctx == tgt])
```

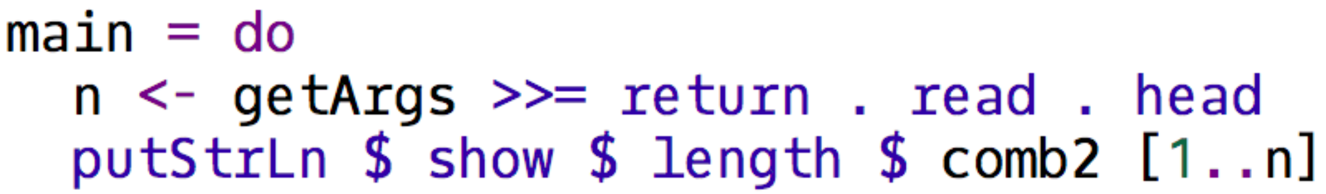
Value Patterns

```
class ValuePat m a where
  valuePat :: (Matcher m a, Eq a) => (HList ctx -> a) -> Pattern a m ctx '[]

data Eql = Eql
instance (Eq a) => Matcher Eql a

instance Eq a => ValuePat Eql a where
  valuePat f = Pattern (\ctx _ tgt -> [MNil | f ctx == tgt])

instance (Matcher m a, Eq a, ValuePat m a) => ValuePat (Multiset m) [a] where
  valuePat f = Pattern (\ctx (Multiset m) tgt ->
                            match (f ctx, tgt) (Pair (List m) (Multiset m)) $
                              [[mc| pair nil nil => [MNil] |],
                               [mc| pair (cons $x $xs) (cons #x #xs) => [MNil] |],
                               [mc| Wildcard => [] |]]
```

Today’s Contents

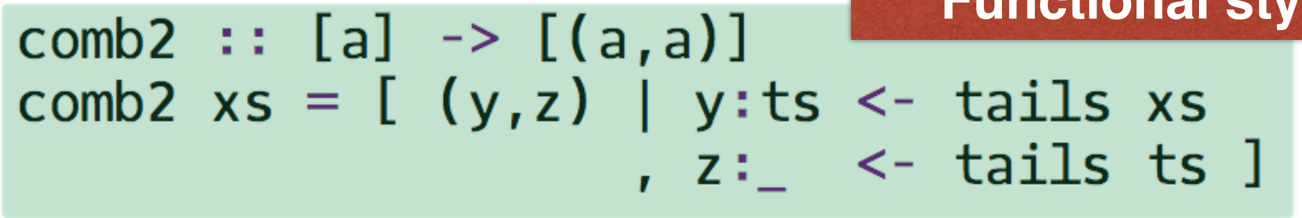
- Tutorial of MiniEgison
- Background
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- Implementation of MiniEgison
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- **Performance**
- Conclusion

Experiment the overhead of miniEgison



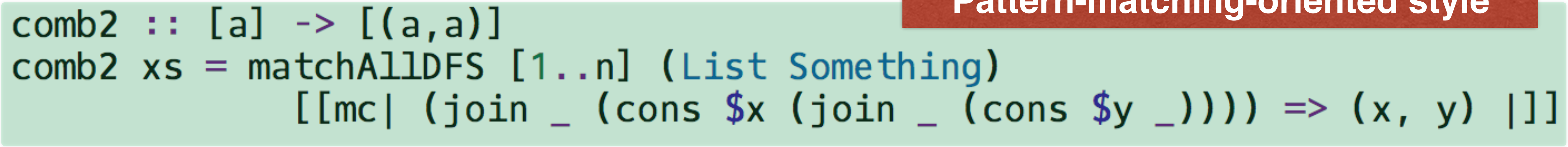
```
main = do
  n <- getArgs >>= return . read . head
  putStrLn $ show $ length $ comb2 [1..n]
```

Functional style



```
comb2 :: [a] -> [(a,a)]
comb2 xs = [ (y,z) | y:ts <- tails xs
                   , z:_  <- tails ts ]
```

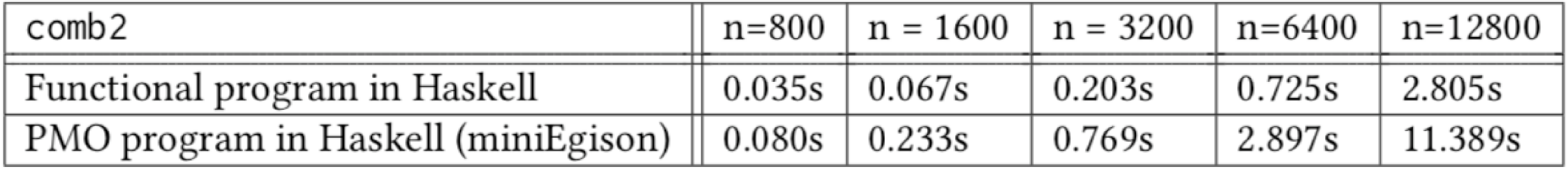
Pattern-matching-oriented style



```
comb2 :: [a] -> [(a,a)]
comb2 xs = matchAllDFS [1..n] (List Something)
             [[mc| (join _ (cons $x (join _ (cons $y _)))) => (x, y) |]]
```

Benchmark results

- **The overhead of miniEgison is not so large (only 2-4 times in this case).**



comb2	n=800	n = 1600	n = 3200	n=6400	n=12800
Functional program in Haskell	0.035s	0.067s	0.203s	0.725s	2.805s
PMO program in Haskell (miniEgison)	0.080s	0.233s	0.769s	2.897s	11.389s

Today's Contents

- Tutorial of MiniEgison
- Background
  - Compilation of Egison Pattern Matching
  - Type System for Egison Pattern Matching
- Implementation of MiniEgison
  - Typing MatchAll
  - Typing Matching States and Matching Atoms
  - User-Defined Pattern-Matching Algorithms
- Performance
- **Conclusion**

MiniEgison: a new pattern-matching library for Haskell

This presentation showed how miniEgison is implemented utilizing the following Haskell features (GHC extensions):

- **Template Haskell** is used to transform match clauses;
- **generalized algebraic data types** are used to define patterns;
- **existential types** are used to define match clauses and matching atoms;
- **datatype promotion** is used to represent intermediate pattern-matching results;
- **multi-parameter type classes** are used to type matchers.

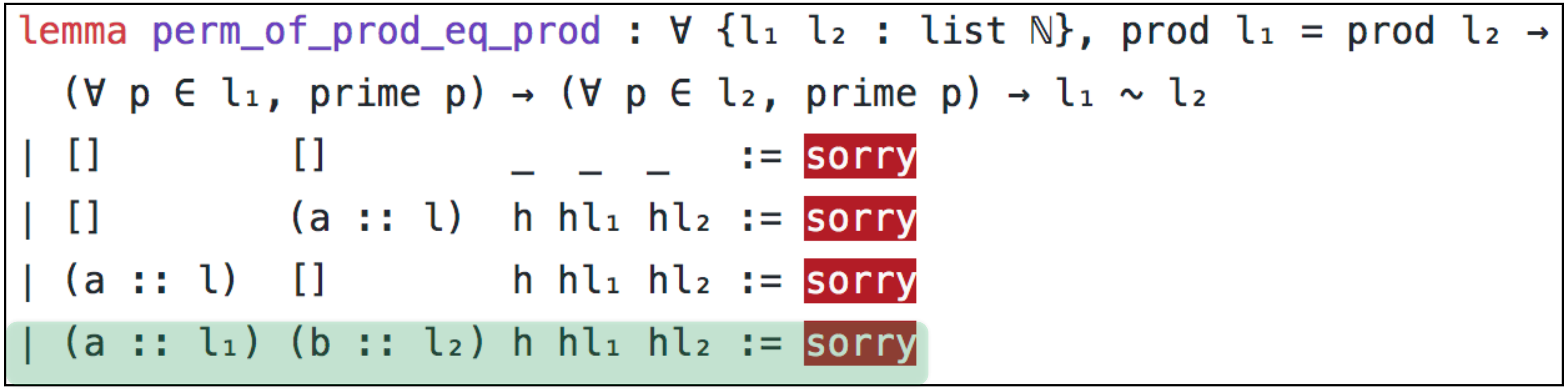
Future work

- Implement miniEgison as a GHC extension.
- Implement Egison pattern matching on theorem provers.

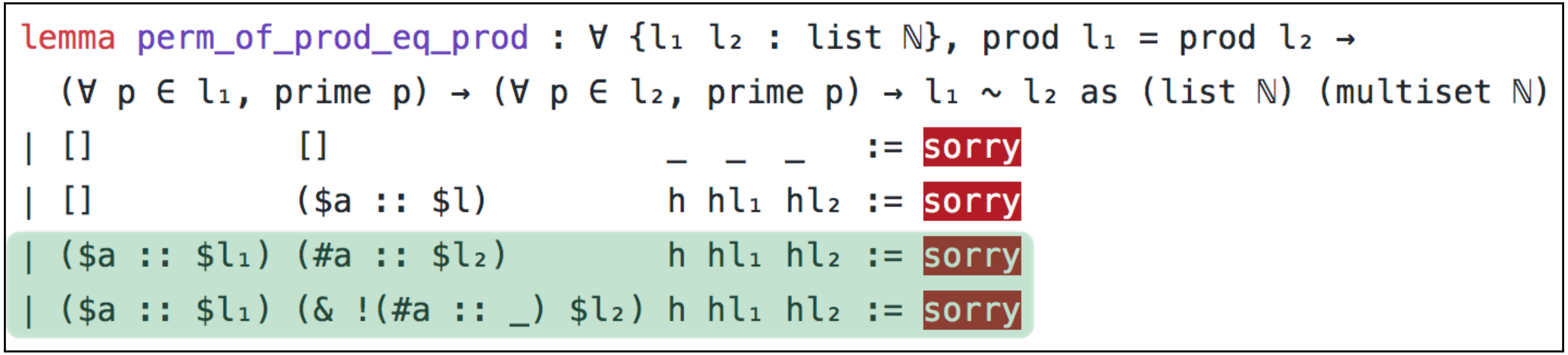
Future work

- Implement miniEgison as a GHC extension.
- **Implement Egison pattern matching on theorem provers.**

Proofs of fundamental theorem of arithmetic in Lean and Lean + Egison.



```
lemma perm_of_prod_eq_prod : ∀ {l₁ l₂ : list ℕ}, prod l₁ = prod l₂ →
  (∀ p ∈ l₁, prime p) → (∀ p ∈ l₂, prime p) → l₁ ~ l₂
| []        []        _  _  _   := sorry
| []        (a :: l)  h hl₁ hl₂ := sorry
| (a :: l)  []        h hl₁ hl₂ := sorry
| (a :: l₁) (b :: l₂) h hl₁ hl₂ := sorry
```



```
lemma perm_of_prod_eq_prod : ∀ {l₁ l₂ : list ℕ}, prod l₁ = prod l₂ →
  (∀ p ∈ l₁, prime p) → (∀ p ∈ l₂, prime p) → l₁ ~ l₂ as (list ℕ) (multiset ℕ)
| []          []                 _  _  _   := sorry
| []          ($a :: $l)         h hl₁ hl₂ := sorry
| ($a :: $l₁) (#a :: $l₂)        h hl₁ hl₂ := sorry
| ($a :: $l₁) (& !(#a :: _) $l₂) h hl₁ hl₂ := sorry
```

- **Pattern matching for non-free data types (e.g., multisets) will make descriptions of proofs concise.**

Acknowledgments

- I thank **Mayuko Kori** for her help for implementing miniEgison.
- I thank **Akira Kawata** for implementing a proof-of-concept type system of Egison in Typed Egison.
- I thank **Yuichi Nishiwaki** and **Matthew Roberts** for constructive discussion while implementing miniEgison.

Rakuten