

Cornell University  
Department of Computer Science

## Reconciling Exhaustive Pattern Matching with Objects

**Chin Isradisaikul**

chinawat@cs.cornell.edu

Andrew Myers

andru@cs.cornell.edu

Department of Computer Science, Cornell University | Ithaca, NY

PLDI 2013 | Wed, June 19, 2013 | Seattle, WA

integrating

# pattern matching

(makes code concise & safer)

with

# object-oriented programming

(helps software scale)

# Pattern matching in OCaml: concise & safer code

```
type list = Nil | Cons of int * list
```

```
match l with
```

```
| Nil -> ...
```

```
| Cons(x1, Cons(x2, l')) -> ...
```

```
| Cons(x1, Cons(x2, Cons(x3, l'))) -> ...
```

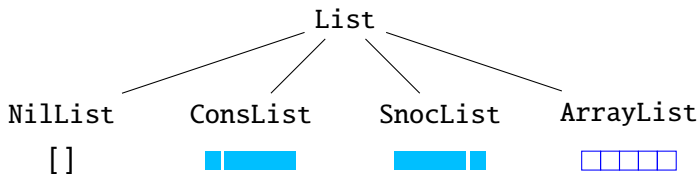
list =   $\sqcup$   Cons

Cons : int × list → list  
Cons<sup>-1</sup> : list → int × list

Exhaustiveness: `Cons(17, Nil)` not matched → warning

Nonredundancy: Third arm unnecessary → warning

## data abstraction



## multiple implementations

```
switch (l) {        // Want this!  
case Nil(): ...  
case Cons(int x1, Cons(int x2, List tl)): ...  
case Cons(int x1, Cons(int x2, Cons(int x3, List tl))): ...  
}
```

# Have your cake and eat it too?

## Problem statement

Can we satisfy all these goals **without** violating data abstraction?

- 1 implementation-oblivious pattern matching
- 2 verification of exhaustive and nonredundant pattern matching



# Comparison: prior & our approaches

<b>approach</b>			data abstraction	constructors usable as patterns	multiple impls of data types	exhaustiveness check
ML pattern matching			✓			✓
views	[W 87]	✓				
active patterns in F#	[SNM 07]	✓				✓
extractors	[EOW 07]	✓	✓	✓		
sealed classes in Scala	[OSV 08]		✓			✓
JMatch 1.1.6	[LM 03]	✓	✓	✓		
<b>JMatch 2.0</b>		✓	✓	✓	✓	✓

## Modal abstraction in JMatch 1.1.6

list Cons in ~Java:

hd      tl

Cons : int × list → list

```
Cons(int x, List l) {  
  this.hd = x;  
  this.tl = l;  
}
```

Cons<sup>-1</sup> : list → int × list

```
(int * List) cons() {  
  return (this.hd, this.tl);  
}
```

Different views of the same relation:

$$\{(this, x, l) \in \text{Cons} \times \text{int} \times \text{List} \mid this.hd = x \wedge this.tl = l\}$$

## Modal abstraction in JMatch 1.1.6

list Cons in ~Java:

hd      tl

Cons : int × list → list

```
Cons(int x, List l) {  
  this.hd = x;  
  this.tl = l;  
}
```

Cons<sup>-1</sup> : list → int × list

```
(int * List) cons() {  
  return (this.hd, this.tl);  
}
```

Different views of the same relation:

$$\{(this, x, l) \in \text{Cons} \times \text{int} \times \text{List} \mid this.hd = x \wedge this.tl = l\}$$

JMatch 1.1.6:

```
Cons(int x, List l) returns(x, l) (  
  this.hd = x && this.tl = l  
)
```



## Modal abstraction in JMatch 1.1.6

list Cons in ~Java:

hd      tl

Cons : int × list → list

```
Cons(int x, List l) {  
  this.hd = x;  
  this.tl = l;  
}
```

Cons<sup>-1</sup> : list → int × list

```
(int * List) cons() {  
  return (this.hd, this.tl);  
}
```

Different views of the same relation:

$$\{(this, x, l) \in \text{Cons} \times \text{int} \times \text{List} \mid this.hd = x \wedge this.tl = l\}$$

JMatch 1.1.6:

```
Cons(int x, List l) returns(x, l) (  
  this.hd = x && this.tl = l  
)
```

## Modal abstraction in action

```
Cons(int x, List l) returns(x, l) (  
  this.hd = x && this.tl = l  
)
```

```
// forward mode  
let List l = Cons(hd, tl);  
// backward mode  
let l = Cons(int hd, List tl);
```

```
List l0 = Nil();           // l0 = []  
List l1 = Cons(17, l0);    // l1 = [17; []]  
List l2 = Cons(42, l1);    // l2 = [42; [17; []]]
```

```
switch (l2) {  
  case Nil(): ...  
  case Cons(int x1, List l): ... // x1 ↦ 42, l ↦ [17; []]  
  case Cons(int x1, Cons(int x2, List l)): ...  
} // x1 ↦ 42, x2 ↦ 17, l ↦ Nil()
```

# Have your cake and eat it too?

## Problem statement

Can we satisfy all these goals **without** violating data abstraction?

- 1 implementation-oblivious pattern matching
- 2 verification of exhaustive and nonredundant pattern matching



# Implementation-oblivious pattern matching

```
// JMatch 1.1.6
Cons(int x, List l) returns(x, l) (
  this.hd = x && this.tl = l
)
```

Problem: `Cons` constructors belong to the `Cons` class.

Solution: Declare `Cons` independently of implementations.

## JMatch 2.0 — Constructors in interfaces

Constructors can be declared in interfaces:

```
interface List {  
    constructor nil() returns();  
    constructor cons(int x, List l) returns(x, l);  
}
```

## JMatch 2.0 — Constructors in interfaces

Constructors can be declared in interfaces:

```
interface List {  
    constructor nil() returns();  
    constructor cons(int x, List l) returns(x, l);  
}
```

```
class Nil implements List {  
    public constructor nil() returns() ( true )  
    public constructor cons(int x, List l)  
        returns(x, l) ( false )  
}  
class Cons implements List {  
    int hd; List tl;  
    public constructor nil() returns() ( false )  
    public constructor cons(int x, List l)  
        returns(x, l) ( this.hd = x && this.tl = l )  
}
```

## Another List implementation

*snoc* list:

hd      tl

```
class Snoc implements List {
  List hd;
  int tl;

  public constructor nil() returns() ( false )
  public constructor cons(int x, List l) returns(x, l) (
    l = nil() && this.hd = l && this.tl = x
    | l = Snoc(List lhd, int ltl)
      && this.hd = cons(x, lhd) && this.tl = ltl
  )
  Snoc(List l, int x) returns(l, x) (
    this.hd = l && this.tl = x
  )
}
```

## Another List implementation

snoc list:

hd

tl

```
class Snoc implements List {
  List hd;
  int tl;

  public constructor nil() returns() ( false )
  public constructor cons(int x, List l) returns(x, l) (
    l = nil() && this.hd = l && this.tl = x
    | l = Snoc(List lhd, int ltl)
      && this.hd = cons(x, lhd) && this.tl = ltl
  )
  Snoc(List l, int x) returns(l, x) (
    this.hd = l && this.tl = x
  )
}
```



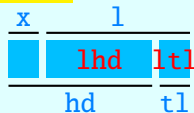
## Another List implementation

snoc list:

hd tl

```
class Snoc implements List {
  List hd;
  int tl;

  public constructor nil() returns() ( false )
  public constructor cons(int x, List l) returns(x, l) (
    l = nil() && this.hd = l && this.tl = x
  | l = Snoc(List lhd, int ltl)
    && this.hd = cons(x, lhd) && this.tl = ltl
  )
  Snoc(List l, int x) returns(l, x) (
    this.hd = l && this.tl = x
  )
}
```



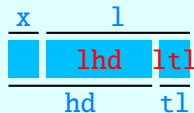
## Another List implementation

snoc list:

hd tl

```
class Snoc implements List {
  List hd;
  int tl;

  public constructor nil() returns() ( false )
  public constructor cons(int x, List l) returns(x, l) (
    l = nil() && this.hd = l && this.tl = x
  | l = Snoc(List lhd, int ltl)
    && this.hd = cons(x, lhd) && this.tl = ltl
  )
  Snoc(List l, int x) returns(l, x) (
    this.hd = l && this.tl = x
  )
}
```



## JMatch 2.0 — Equality constructors

```
l = Snoc(List lhd, int lt1)
l.equals(Snoc(List lhd, int lt1)) // equals multimodal
```

Problem: `l` is nonempty but might not be a Snoc.

Solution:

- Convert `l` into a Snoc first (always succeeds).
- Do this implicitly; don't bother programmer.

*Equality constructors* specify *how* the conversion should be done.

```
public constructor equals(List l) (
  l = cons(int lhd, List lt1) && cons(lhd, lt1)
)
```

# Have your cake and eat it too?

## Problem statement

Can we satisfy all these goals **without** violating data abstraction?

- 1 implementation-oblivious pattern matching ✓
- 2 verification of exhaustive and nonredundant pattern matching



## Checking exhaustiveness and nonredundancy

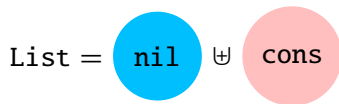
```
interface List {  
    constructor nil() returns();  
    constructor cons(int x, List l) returns(x, l);  
}
```

```
switch (l) {  
case nil(): ...  
case cons(int hd, List tl): ...  
}
```

- 1 **switch** exhaustive?
- 2 Any **case** redundant?

# Invariants

- 1 `nil` and `cons` can construct every `List`.
- 2 No value can be constructed by both `nil` and `cons`.

$$\text{List} = \text{nil} \uplus \text{cons}$$


# Invariants

- 1 `nil` and `cons` can construct every `List`.
- 2 No value can be constructed by both `nil` and `cons`.

$$\text{List} = \text{nil} \uplus \text{cons}$$

| represents disjoint disjunction:

```
invariant(this = nil() | this = cons(_, _));
```

Add invariants to interfaces.

# Invariants

- 1 `nil` and `cons` can construct every `List`.
- 2 No value can be constructed by both `nil` and `cons`.

$$\text{List} = \text{nil} \uplus \text{cons}$$

| represents disjoint disjunction:

```
invariant(this = nil() | this = cons(_, _));
```

| for disjoint patterns:

```
invariant(this = nil() | cons(_, _));
```

Add invariants to interfaces.



# Invariants not enough

```
interface List {  
  constructor nil() returns();  
  constructor cons(int x, List l) returns(x, l);  
  constructor snoc(List l, int x) returns(l, x);  
}
```

```
switch (l) {  
  case nil(): ...  
  case snoc(List hd, int tl): ...  
}
```

- 1 **switch** exhaustive?
- 2 Any **case** redundant?

# Matching precondition

Know: exhaustiveness of

```
switch (l) {  
  case nil(): ...  
  case cons(int hd, List tl): ...  
}
```

Want: exhaustiveness of

```
switch (l) {  
  case nil(): ...  
  case snoc(List hd, int tl): ...  
}
```

If `cons` matches, `snoc` matches.

# Matching precondition

Know: exhaustiveness of

```
switch (l) {  
  case nil(): ...  
  case cons(int hd, List tl): ...  
}
```

Want: exhaustiveness of

```
switch (l) {  
  case nil(): ...  
  case snoc(List hd, int tl): ...  
}
```

If **cons matches**, **snoc** matches.

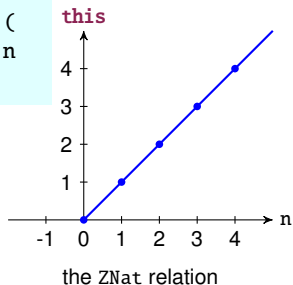


*matching precondition*      **this = cons(, )**

# Partial functions

Natural numbers represented by integers:

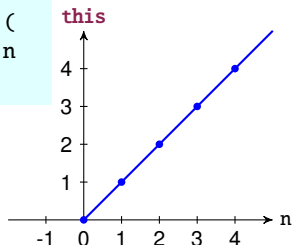
```
ZNat(int n) returns(n) (  
  n >= 0 && this.rep = n  
)
```



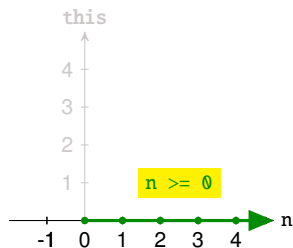
# Partial functions

Natural numbers represented by integers:

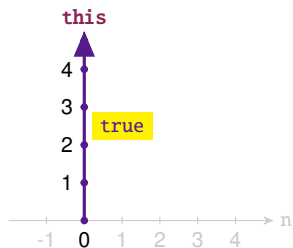
```
ZNat(int n) returns(n) (  
  n >= 0 && this.rep = n  
)
```



the ZNat relation



matching precondition for **returns(this)**



matching precondition for **returns(n)**

# Matches clauses

In ZNat, matching precondition for

- forward mode: `n >= 0`
- backward mode: `true`

Writing a matching precondition per mode is tedious.

# Matches clauses

In ZNat, matching precondition for

- forward mode: `n >= 0`
- backward mode: `true`

Writing a matching precondition per mode is tedious.

Modal abstraction → **consolidated** method body  
    ??? → **consolidated** matching precondition

# Matches clauses

In ZNat, matching precondition for

- forward mode: `n >= 0`
- backward mode: `true`

Writing a matching precondition per mode is tedious.

Modal abstraction → **consolidated** method body

**Matches clause** → **consolidated** matching precondition

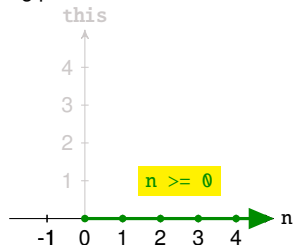
```
ZNat(int n) matches(n >= 0) returns(n) (  
  n >= 0 && this.rep = n  
)
```

How to recover individual matching preconditions?

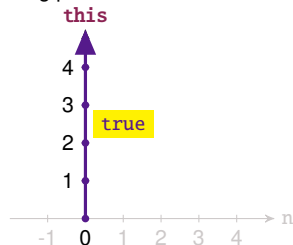


# Specifying & interpreting a matches clause

matching precondition for `returns(this)`

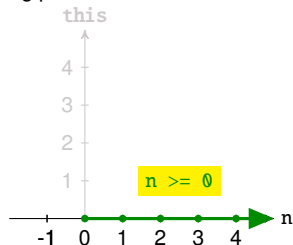


matching precondition for `returns(n)`

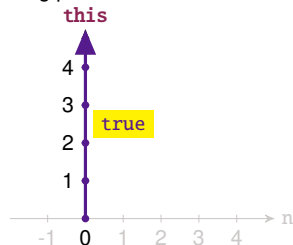


# Specifying & interpreting a matches clause

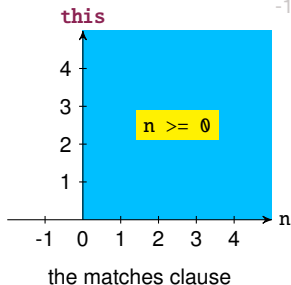
matching precondition for `returns(this)`



matching precondition for `returns(n)`



Use projections to recover individual matching preconditions.



# Verification summary

**matching precondition**  $\Rightarrow$  **method body**

forward mode of ZNat:  $n \geq 0 \Rightarrow \exists rep : n \geq 0 \wedge rep = n$

backward mode of ZNat:  $true \Rightarrow \exists n : n \geq 0 \wedge rep = n$   
(need invariant  $rep \geq 0$ )

```
class ZNat implements Nat {  
  private invariant(rep >= 0);  
  ZNat(int n) matches(n >= 0) returns(n) (  
    n >= 0 && this.rep = n  
  )  
  ...  
}
```

# Have your cake and eat it too?

## Problem statement

Can we satisfy all these goals **without** violating data abstraction?

- 1 implementation-oblivious pattern matching ✓
- 2 verification of exhaustive and nonredundant pattern matching ✓



# Implementation

Pattern matching features:

- Translate to Java (extends JMatch 1.1.6).
- Original semantics redefined to handle implicit equality constructor calls.

Verification:

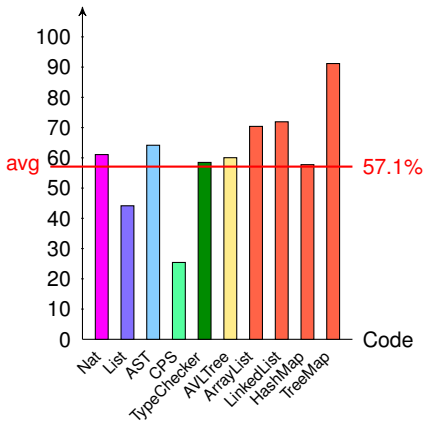
- Encode verification conditions for Z3 theorem prover.

# Evaluation

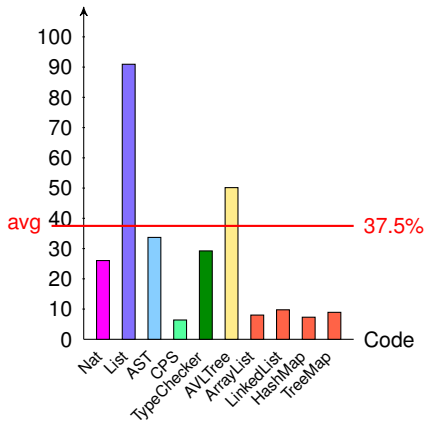
On code examples (include Java collections framework):

- Implemented concisely in JMatch & Java, compare token counts.
- Verification correctness and overhead during compilation.

Expressiveness (%)



Overhead (%)



```

interface Tree {
  invariant(this = leaf() | branch(_,_,_));
  constructor leaf() matches(height() = 0) ensures(height() = 0);
  constructor branch(Tree l, int v, Tree r) matches(height() > 0)
    ensures(height() > 0 &&
      (height() = l.height() + 1 && height() > r.height() ||
        height() > l.height() && height() = r.height() + 1))
    returns(l, v, r);
  int height() ensures(result >= 0);
}

static Tree rebalance(Tree l, int v, Tree r) matches(true) ( // in AVLTree
  result = Branch(Branch(Tree a, int x, Tree b), int y,
    Branch(Tree c, int z, Tree d))
  && ( l.height() - r.height() > 1 && d = r && z = v // rot. from left
    && ( l = branch(Tree ll, y, c) && ll = branch(a, x, b) &&
      ll.height() >= c.height()
      | l = branch(a, x, Tree lr) && lr = branch(b, y, c) &&
        a.height() < lr.height())
    | r.height() - l.height() > 1 && a = l && x = v // rot. from right
    && ( r = branch(Tree rl, z, d) && rl = branch(b, y, c) &&
      rl.height() > d.height()
      | r = branch(b, y, Tree rr) && rr = branch(c, z, d) &&
        b.height() <= rr.height()))
  | abs(l.height() - r.height()) <= 1 && result = Branch(l, v, r)
)

```

# Takeaways

- Compact code for pattern matching possible in object-oriented settings
  - named and equality constructors, disjoint disjunctions
  - pattern disjunctions, tuples
- Verifying exhaustiveness and nonredundancy of pattern matching possible
  - invariants, multimodal matches clauses
  - ensures clauses, opaque matching preconditions







Cornell University  
Department of Computer Science

## Reconciling Exhaustive Pattern Matching with Objects

**Chin Isradisaikul**  
chinawat@cs.cornell.edu

Andrew Myers  
andru@cs.cornell.edu

<http://www.cs.cornell.edu/projects/JMatch/>

# additional slides

## Equality constructors in action

```
public constructor cons(int x, List l) returns(x, l) (  
  l = nil() && this.hd = l && this.tl = x  
  | l = Snoc(List lhd, int ltl) &&  
    this.hd = cons(x, lhd) && this.tl = ltl  
)  
public constructor snoc(List l, int x) returns(l, x) (  
  this.hd = l && this.tl = x  
)  
public constructor equals(List l) (  
  l = cons(int lhd, List ltl) && cons(lhd, ltl)  
)
```

```
List result = Snoc.cons(42, Cons.cons(17, Nil.nil()));
```

Convert `[17; []]` into a Snoc, calling `Snoc.cons(17, Nil.nil())`.  
The conversion is `[[]; 17]`, so `lhd = []`, `ltl = 17`

## Equality constructors in action

```
public constructor cons(int x, List l) returns(x, l) (  
  l = nil() && this.hd = l && this.tl = x  
  | l = Snoc(List lhd, int ltl) &&  
    this.hd = cons(x, lhd) && this.tl = ltl  
)  
public constructor snoc(List l, int x) returns(l, x) (  
  this.hd = l && this.tl = x  
)  
public constructor equals(List l) (  
  l = cons(int lhd, List ltl) && cons(lhd, ltl)  
)
```

```
List result = Snoc.cons(42, Cons.cons(17, Nil.nil()));
```

```
hd = cons(42, []) = [[]; 42], tl = 17
```

```
result = [[]; 42]; 17]
```

## Equality constructors in action

Generic equality constructors for any List:

```
public constructor equals(List l) (  
    l = nil() && nil()  
    | l = cons(int lhd, IntList ltl) && cons(lhd, ltl)  
)
```

In action...

```
List l0 = Nil.nil();           // l0 = []  
List l1 = Cons.cons(17, l0);  // l1 = [17; []]  
List l2 = Snoc.cons(42, l1);  // l2 = [[[]; 42]; 17]  
List l3 = Cons.cons(47, l2);  // l3 = [47; [42; [17; []]]]
```

## Equality constructors in action

Generic equality constructors for any List:

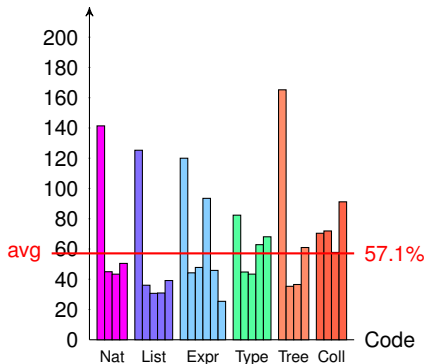
```
public constructor equals(List l) (  
    l = nil() && nil()  
    | l = cons(int lhd, IntList ltl) && cons(lhd, ltl)  
)
```

In action...

```
List l0 = Nil.nil();           // l0 = []  
List l1 = Cons.cons(17, l0);   // l1 = [17; []]  
List l2 = Snoc.cons(42, l1);   // l2 = [[[]; 42]; 17]  
List l3 = Cons.cons(47, l2);   // l3 = [47; [42; [17; []]]]
```

# Evaluation

Expressiveness (%)



Overhead (%)

