Predicate Dispatch

or The Incredible Benefits of Wishful Thinking

(defmulti stepwise identity <=)
(defmethod stepwise 0 [_] 0)
(defmethod stepwise 5 [n] n)
(defmethod stepwise :default [n] (* n n))
(prefer-method stepwise 0 5) ; since -1 < 0 and -1 < 5

I find this much cleaner, and the subtle error above was been eliminated. And if I want to add another case, its not too hard (though the number of prefer-methods might increase exponentially). This is a rather trivial case, but I think it makes the need clear: per-multi comparison functions.

For multiple hierarchies, this would be a common idiom:
(defmulti mymulti **dispatch**-function (fn [object-key method-key] (isa? *my-hiearchy* object-key method-key))

A user could use a ref or atom as needed, no special casing in the Clojure code.

What do people think?
-Mark

On Jan 7, 2009, at 10:44 AM, Meikel Brandmeyer wrote:

- Show quoted text -

- Mark Fredrickson
mark.m.fredrick...@gmail.com
http://www.markmfredrickson.com

Rich Hickey View profile

On Jan 20, 12:13 am, Mark Fredrickson <mark.m.fredrick...@gmail.com> wrote:

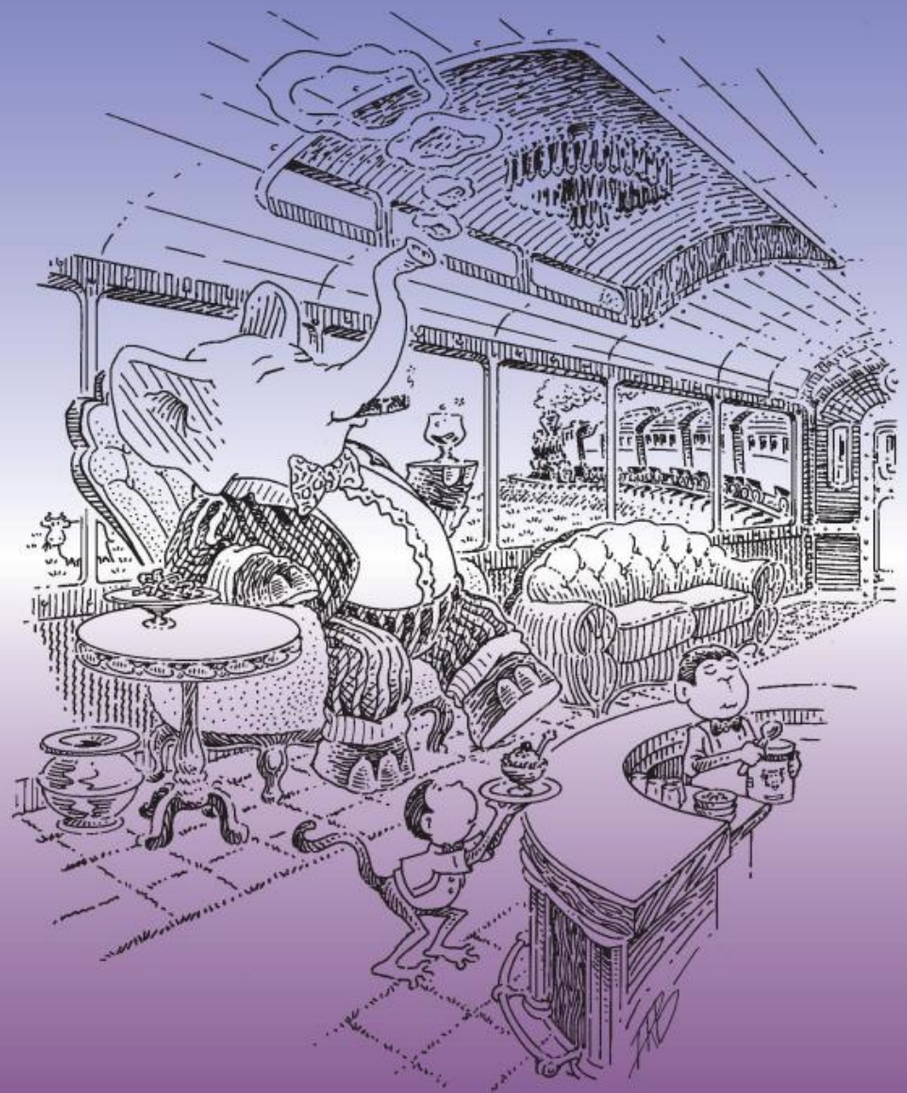
- Show quoted text -

This seems to me to be a half step towards **predicate dispatch**. With the a la carte hierarchies freeing us from type-based inheritance, it might be worth someone revisiting Chambers/Chen style **predicate dispatch**.

Rich

1 year ago

The Reasoned Schemer



Daniel P. Friedman, William E. Byrd,
and Oleg Kiselyov

Relational Programming in miniKanren: Techniques, Applications, and Implementations

William E. Byrd

Submitted to the faculty of the
University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University

August, 2009

Wishful Thinking

Hello William Byrd

& Dan Friedman!

core.logic

core.logic

- A faithful miniKanren implementation based on William Byrd's dissertation

core.logic

- A faithful miniKanren implementation based on William Byrd's dissertation
- Performance oriented enhancements

core.logic

- A faithful miniKanren implementation based on William Byrd's dissertation
- Performance oriented enhancements
- Adds knowledge base facilities

core.logic

- A faithful miniKanren implementation based on William Byrd's dissertation
- Performance oriented enhancements
- Adds knowledge base facilities
- cKanren enhancements coming up!

Pattern Matching

Pattern Matching

- Was interested in implementing alphaKanren for nominal logic programming

Pattern Matching

- Was interested in implementing alphaKanren for nominal logic programming
- Didn't like the pattern matching macro in the dissertation (sorry Mr. Kiselyov)

Pattern Matching

- Was interested in implementing alphaKanren for nominal logic programming
  - Didn't like the pattern matching macro in the dissertation (sorry Mr. Kiselyov)
  - Down the rabbit hole!

Compiling Pattern Matching to Good Decision Trees

Lazy Pattern Matching

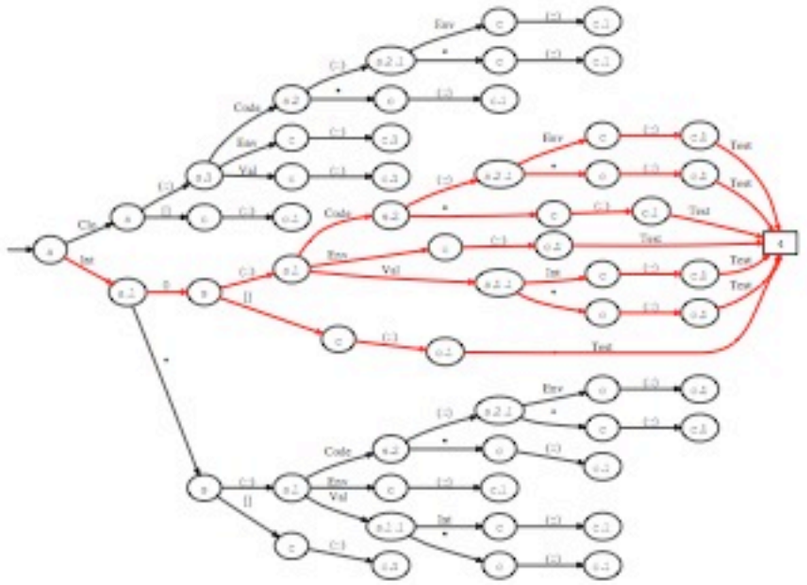


Figure 6. Naive decision tree for example 3

Figure 7. Minimal decision tree for example 3

Oddly familiar...

Our algorithm first reduces methods written in the general predicate dispatching model (which generalizes single dispatching, multiple dispatching, predicate classes and classifiers, and pattern-matching)

Predicate Dispatch

Predicate Dispatch

- Chambers (Self) & Chen

Predicate Dispatch

- Chambers (Self) & Chen
- Some details are hardwired

Predicate Dispatch

- Chambers (Self) & Chen
- Some details are hardwired
- How to remove the hardwired details?

Predicate Dispatch

- Chambers (Self) & Chen
- Some details are hardwired
- How to remove the hardwired details?
- Will it be dog slow?

Predicate Dispatch

- Chambers (Self) & Chen
- Some details are hardwired
- How to remove the hardwired details?
- Will it be dog slow?
- Challenges, challenges ...

Rich Hickey doesn't like pattern matching

Sam Tobin-Hochstadt

says multimethods are scary

:(

Then why bother?

This is not Java

The Art of the Metaobject Protocol

Gregor Kiczales
Jim des Rivières
Daniel G. Bobrow

Why Predicate Dispatch?

What's wrong with multimethods?

```
(derive ::rect ::shape)
(defmulti bar (fn [x y] [x y]))
(defmethod bar [::rect ::shape] [x y] :rect-shape)
```

dispatch fn hardwired

Can we do better?

Expressiveness, Simplicity & Users

Trends

- Simpler ideas easier to adopt
  - Sophisticated ideas need a simple story to be impactful
  - Ideal: “deceptively simple”

- Unification != Swiss Army Knife

- Language papers have had more citations; compiler work has had more practical impact
  - The combination can work well

Conclusions

- Simpler ideas easier to adopt
  - By researchers and by users
- Sophisticated ideas still needed, to support simple interfaces
- Doing things dynamically instead of statically can be liberating

Is predicate dispatch simple?

is open dispatch simple(r)?

```
(defn walk [inner outer form]
  (cond
    (list? form) (outer (apply list (map inner form)))
    (instance? clojure.lang.IMapEntry form) (outer (vec (map inner form)))
    (seq? form) (outer (doall (map inner form)))
    (coll? form) (outer (into (empty form) (map inner form)))
    :else (outer form)))
```

```
(defm walk [inner outer (form :when list?)] ...)
(defm walk [inner outer (form :when #(instance? IMapEntry %))] ...)
(defm walk [inner outer (form :when seq?)] ...)
(defm walk [inner outer (form :when coll?)] ...)
(defm walk [inner outer form] ...)
```

```
(define unify (λ (u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond
      ((eq? u v) s)
      ((var? u)
       (cond
         ((var? v) (ext-s-no-check u v s))
         (else (ext-s u v s))))
      ((var? v)
       (ext-s v u s))
      ((and (pair? u) (pair? v))
       (let ((s (unify (car u) (car v) s)))
         (and s (unify (cdr u) (cdr v) s))))
      ((equal? u v) s)
      (else #f)))))
```

```
(unify [this u v]
  (if (identical? u v)
    this
    (let [u (walk this u)
          v (walk this v)]
      (if (identical? u v)
        this
        (unify-terms u v this)))))
```

```
(extend-type Object
  IUnifyWithLVar
  (unify-with-lvar [v u s]
    (ext s u v)))
```

```
(defm unify [(u :when lvar?) _]
  ...)

(defm unify [_ (v :when lvar?)]
  ...)

(defm unify [(u :when lvar?) (v :when map?)]
  ...)

(defm unify [(u :when map?) (v :when lvar?)]
  (unify v u))

...
```

Winding roads

Pattern Matching

Pattern Matching

- Rich history of papers going back to the mid 80s

Pattern Matching

- Rich history of papers going back to the mid 80s
- Goldmine of ideas on efficiency

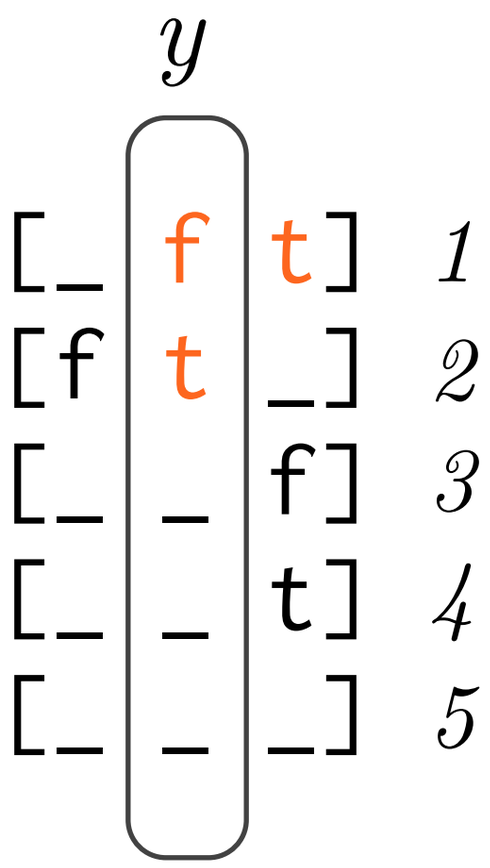
Pattern Matching

- Rich history of papers going back to the mid 80s
- Goldmine of ideas on efficiency
- The key enhancement that predicate dispatch offers is open extension

```
(match [x y z]
   [_ false true] 1
   [false true _ ] 2
   [_ _ false] 3
   [_ _ true] 4
   :else 5)
```

x	y	z	
[_	f	t]	*1*
[f	t	_]	*2*
[_	_	f]	*3*
[_	_	t]	*4*
[_	_	_]	*5*

x	y	z	
[_	f	t]	*1*
[f	t	_]	*2*
[_	_	f]	*3*
[_	_	t]	*4*
[_	_	_]	*5*


y
[_ f t] 1
[f t _] 2
[_ _ f] 3
[_ _ t] 4
[_ _ _] 5

y	x	z	
[f	_	t]	*1*
[t	f	_]	*2*
[_	_	f]	*3*
[_	_	t]	*4*
[_	_	_]	*5*

Each pattern type is handed all matching rows for analysis

Matching seqs

```
(match [x]
    [([1 2 3 4] :seq)] 1
    [([1 2 & r] :seq)] 2
    [([1 & r] :seq)] 3
    :else 4)
```

$$x$$

```
[(1 2 3 4)]  1
[(1 2 & r)]  2
[(1 & r)  ]  3
[ _       ]  4
```

x_h	x_t	
[1	(2 3 4)]	*1*
[1	(2 & r)]	*2*
[1	(1 & r)]	*3*
[_	_]	*4*

Matching maps

```
(match [x]
    [{:a a :b 2}] 1
    [{:b 3 :c 4}] 2
    [{:a 1 :d 5}] 3
    :else 4)
```

x

```
[{:a a :b 2}]  1
[{:b 3 :c 4}]  2
[{:a 1 :d 5}]  3
[ _         ]  4
```

$x_{:a}$ $x_{:b}$ $x_{:c}$ $x_{:d}$

[a 2 _ _] *1*
[_ 3 4 _] *2*
[1 _ _ 5] *3*
[_ _ _ _] *4*

$x_{:a}$ $x_{:b}$ $x_{:c}$ $x_{:d}$

[a 2 _ _] *1*
[_ 3 4 _] *2*
[1 _ _ 5] *3*
[_ _ _ _] *4*

```
(match [x]
    [({:a a :b 2} :only [:a :b])] 1
    [{:b 3 :c 4}] 2
    [{:a 1 :d 5}] 3
    :else 4)
```

$x_{:a}$ $x_{:b}$ $x_{:c}$ $x_{:d}$

[a 2 ⚡ ⚡] *1*
[_ 3 4 _] *2*
[1 _ _ 5] *3*
[_ _ _ _] *4*

$x_{:a}$ $x_{:b}$ $x_{:c}$ $x_{:d}$

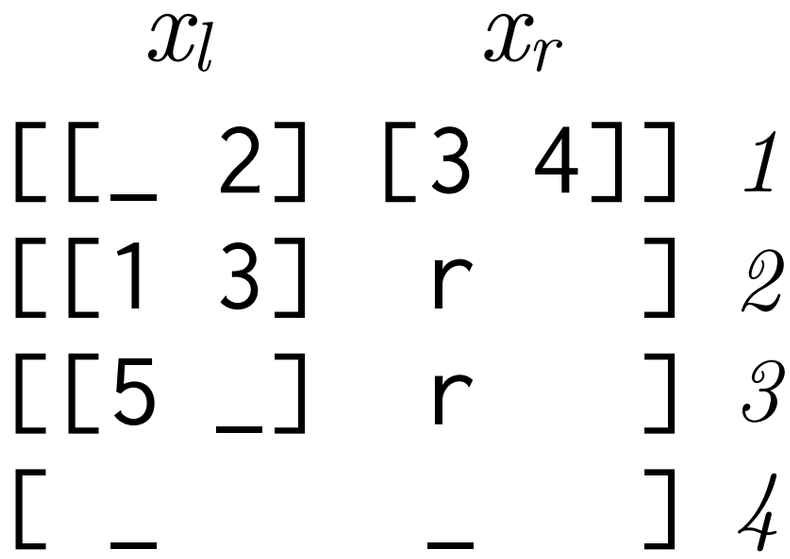
[a 2 ⚡ ⚡] *1*
[_ 3 4 _] *2*
[1 _ _ 5] *3*
[_ _ _ _] *4*

Matching vectors

```
(match [x]
   [[_ 2 3 4]] 1
   [[1 3 & r]] 2
   [[5 _ & r]] 3
   :else 4)
```

x

```
[[_ 2 3 4]] 1
[[1 3 & r]] 2
[[5 _ & r]] 3
[ _       ] 4
```


x_l
x_r
[[_ 2] [3 4]] 1
[[1 3] r] 2
[[5 _] r] 3
[_ _] 4

x_l x_r

```
[[_ 2] [3 4]] 1
[[1 3]   r  ] 2
[[5 _]   r  ] 3
[ _      _  ] 4
```

x_{l0} x_{l1} x_r

```
[_ 2 [3 4]] 1
[1 3   r  ] 2
[5 _   r  ] 3
[_ _   _  ] 4
```

x_{l0} x_{l1} x_r

```
[_ 2 [3 4]] 1
[1 3   r  ] 2
[5 _   r  ] 3
[_ _   _  ] 4
```

Interesting, no?

Closed to Open

Static to Dynamic

No hardwiring

Open Extension

Open Extension

- Pattern matching is top down

Open Extension

- Pattern matching is top down
- Open extension is not!

Open Extension

- Pattern matching is top down
- Open extension is not!
  - even? implies 2

Open Extension

- Pattern matching is top down
- Open extension is not!
  - even? implies 2
  - Inserting some (core)logic

Open Extension

- Pattern matching is top down
- Open extension is not!
  - even? implies 2
  - Inserting some (core)logic
  - But we don't want to break other people's code (namespace local changes?)

```
(defpred even? even?)

(defm foo [(x :when even?)]
 ...)

(defm foo [2]
 ...)
```

More Challenges

More Challenges

- core.match being closed is simple - generate the correct static source.

More Challenges

- core.match being closed is simple - generate the correct static source.
- Supporting dynamism - performant implementation no longer simple

More Challenges

- core.match being closed is simple - generate the correct static source.
- Supporting dynamism - performant implementation no longer simple
- If we change to a graph representation we lose the performance benefits of host branching primitives (if ... else)

More Challenges

- core.match being closed is simple - generate the correct static source.
- Supporting dynamism - performant implementation no longer simple
  - If we change to a graph representation we lose the performance benefits of host branching primitives (if ... else)
  - lazy compilation? complicates targeting ClojureScript

- We could emit the entire dispatch at each extension point. Code size will suffer.

- We could emit the entire dispatch at each extension point. Code size will suffer.

- Like deftype / protocols perhaps we could fully optimize all cases defined “together”, future extension taking a reasonable performance hint.

- We could emit the entire dispatch at each extension point. Code size will suffer.

- Like deftype / protocols perhaps we could fully optimize all cases defined "together", future extension taking a reasonable performance hint.

- Grouped future extension receive shared performance benefits?

- We could emit the entire dispatch at each extension point. Code size will suffer.

- Like deftype / protocols perhaps we could fully optimize all cases defined "together", future extension taking a reasonable performance hint.

- Grouped future extension receive shared performance benefits?

- What about redefinition? What about introspection?

- Extensions that require a reordering - “game over”, perhaps we emit the entire decision tree locally to the namespace?

- Extensions that require a reordering - “game over”, perhaps we emit the entire decision tree locally to the namespace?

- Big idea: non-overlapping clauses

- Extensions that require a reordering - "game over", perhaps we emit the entire decision tree locally to the namespace?

- Big idea: non-overlapping clauses

- Other ideas / solutions / approaches?

- Extensions that require a reordering - “game over”, perhaps we emit the entire decision tree locally to the namespace?

- Big idea: non-overlapping clauses

- Other ideas / solutions / approaches?

- Wishful thinking is fun isn’t it? :)

Questions?