Finite Automata Theory Based Optimization of Conditional Variable Binding

An efficient type-aware destructuring-case

Jim Newton

12th European Lisp Symposium

1,2 April 2019
We would like to introduce a *user-defined* construct called **destructuring-case**, which *efficiently* selects a clause to evaluate designated by a **destructuring lambda list** depending on run-time value of a given expression.

There semantics of the macro usage **should be intuitive**.

There are **several cases** to consider.
Different number of required arguments

(destructuring-case expression
  ((X)
   (* X 100)
   ((X Y)
    (* X Y))
   ((X Y Z)
    (+ (* X Y) Z)))))
Different optional arguments

(destructuring-case expression
  ((X &optional (Y 1))
   (* X Y))
  ((X &key (Y 1))
   (* X Y))
  ((X &key (Y 1) (Z 0) &allow-other-keys)
   (+ (* X Y) Z))))
(destructuring-case expression
  ((X Y)
   (declare (type fixnum X Y))
   (* X Y))
  ((X Y)
   (declare (type fixnum X)
             (type integer Y))
   (* X Y))
  ((X Y)
   (declare (type (or string fixnum) X)
             (type number Y))
   (* (if (stringp X)
        (string-to-number X)
        X)
      Y)))
1 Motivating Example

2 Efficient Type-Based Pattern Matching

3 Destructuring Lambda lists as Patterns

4 Efficiently implementing destructuring-case

5 Short Demo

6 Conclusion
Efficient Type-Based Pattern Matching
Does this sequence:
(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)
follow the pattern: \((symbol \cdot (number^+ \lor string^+))^+)\)?
Does this sequence:
(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)
follow the pattern: \((symbol \cdot (number^+ \lor string^+))^+\) ?

We construct a deterministic finite automaton (DFA).

We want to support :not and :and in our DSL.
How does a DFA work as a type predicate?
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)

```
Jim Newton (12th European Lisp Symposium: Finite Automata Theory Based Optimization)
```

```
1,2 April 2019  8 / 32
```
How does a DFA work as a type predicate?

```
(\( a 8 8.0 \ b \ "a" "an" "the" c 8 88 888 d 8/3 \)
```
How does a DFA work as a type predicate?

(a 8.0 "a" "an" "the" c 8 88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8.0 b "a" "an" "the" c 8 88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b
"a" "an" "the"
c 8 88 888 d
8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "the" "an" c 8 88 888 d 8/3)

Diagram:

- **States**:
  - 0
  - 1
  - 2
  - 3

- **Transitions**:
  - From 0 to 1 on symbol
  - From 1 to 2 on symbol
  - From 1 to 3 on symbol
  - From 2 to 2 on number
  - From 3 to 3 on string

- **Symbols**:
  - number
  - symbol
  - string
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "an" "the" © 8 88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "an" "the" c 8
88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)
How does a DFA work as a type predicate?

(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)

Diagram:

- Initial state: 0
- Final states: 1, 2, 3
- Transitions:
  - From 0: symbol to 1
  - From 1: symbol to 3, number to 2
  - From 2: number to 2
  - From 3: string to 3
How does a DFA work as a type predicate?

Yes, it’s a match!

(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
Code generated from \((symbol \cdot (number^+ \lor string^+))^+)\)

\[
\text{tagbody}
\]

0

\[
\begin{align*}
\text{unless seq } & (\text{return nil}) \\
\text{typecase } (\text{pop seq}) & \\
(symbol & (\text{go 1})) \\
(t & (\text{return nil}))
\end{align*}
\]

1

\[
\begin{align*}
\text{unless seq } & (\text{return nil}) \\
\text{typecase } (\text{pop seq}) & \\
(number & (\text{go 2})) \\
(string & (\text{go 3})) \\
(t & (\text{return nil}))
\end{align*}
\]

2

\[
\begin{align*}
\text{unless seq } & (\text{return t}) \\
\text{typecase } (\text{pop seq}) & \\
(number & (\text{go 2})) \\
(symbol & (\text{go 1})) \\
(t & (\text{return nil}))
\end{align*}
\]

3

\[
\begin{align*}
\text{unless seq } & (\text{return t}) \\
\text{typecase } (\text{pop seq}) & \\
(string & (\text{go 3})) \\
(symbol & (\text{go 1})) \\
(t & (\text{return nil}))
\end{align*}
\]
Introducing Regular Type Expression

A Regular Type Expression (RTE) is a surface syntax DSL expressing regular type patterns in sequences.

\[
(symbol \cdot (rational^* \lor float^+) ) \land t \cdot ratio^? \cdot number
\]

RTE DSL notation:

\[
(: and (: cat symbol
\quad (: or (: * rational)
\quad (: + float ))
\quad (: not (: cat t (: ? ratio ) number )))
\]

Regular type expressions express components:
required, optional, repeating, and typed.
Destructuring Lambda lists as Patterns
Lambda-lists characterized by regular patterns

A lambda-list in Common Lisp has a fixed part

(destructuring-bind (a b)
    DATA
    ...)

Jim Newton (12th European Lisp Symposium: Finite Automata Theory Based Optimization) 1,2 April 2019 12 / 32
Lambda-lists characterized by regular patterns

A lambda-list in Common Lisp has a fixed part, an optional part

(destructuring-bind (a b &optional c)
    DATA
    ...)

Jim Newton (12th European Lisp Symposium) Finite Automata Theory Based Optimization 1,2 April 2019 12 / 32
A lambda-list in Common Lisp has a fixed part, an optional part, and a repeating part.

```
(destructuring-bind (a b &optional c &key x y) DATA ...
```

Lambda-lists characterized by regular patterns

A lambda-list in Common Lisp has a fixed part, an optional part, and a repeating part. Any of the variables may be restricted by type declarations.

(destructuring-bind (a b &optional c &key x y) DATA
    (declare (type integer a x)
             (type string b c y))
  ...)

Efficiently implementing destructuring-case
Macro: destructuring-case

(destructuring-case expression
 ((X Y)
  (declare (type fixnum X Y))
  :clause-1)
 ((X Y)
  (declare (type fixnum X)
           (type integer Y))
  :clause-2)
 ((X Y)
  (declare (type (or string fixnum) X)
           (type number Y))
  :clause-3))
Expansion of destructuring-case

\[
\text{(rte-case expression} \\
\left(\text{(:cat fixnum fixnum)}\right) \\
\text{(destructuring-bind } (X \ Y) \ \text{expression} \\
\left(\text{declare (type fixnum } X \ Y)\right) \\
: \text{clause-1})
\]

\[
\left(\text{(:cat fixnum integer)}\right) \\
\text{(destructuring-bind } (X \ Y) \ \text{expression} \\
\left(\text{declare (type fixnum } X) \\
\left(\text{type integer } Y)\right) \\
: \text{clause-2})
\]

\[
\left(\text{(:cat (or string fixnum) number)}\right) \\
\text{(destructuring-bind } (X \ Y) \ \text{expression} \\
\left(\text{declare (type (or string fixnum) } X) \\
\left(\text{type number } Y)\right) \\
: \text{clause-3})\right)
\]
Simplified \texttt{rte-case} expansion

\begin{verbatim}
(rte-case expression
  ((:cat fixnum fixnum )
   :clause-1)
  ((:cat fixnum integer)
   :clause-2)
  ((:cat (or string fixnum) number)
   :clause-3))
\end{verbatim}
Automata for clauses of `rte-case`

\[
\text{rte-case expression} \\
\text{((:cat fixnum fixnum) :clause-1)}
\]

\[
\text{((:cat fixnum integer) :clause-2)}
\]

\[
\text{((:cat (or string fixnum) number) :clause-3))}
\]
We could select the appropriate clause by executing the three automata in turn at run-time.
Automata for clauses of `rte-case`

\[\text{rte-case expression} \]
\[
(( :\text{cat fixnum fixnum} ) \colon\text{clause-1})
\]

\[
(( :\text{cat fixnum integer} ) \colon\text{clause-2})
\]

\[
(( :\text{cat (or string fixnum) number} ) \colon\text{clause-3})
\]

We can do better.
DFAs for disjoined clause-1, clause-2, and clause-3

(rte-case expression
  ((:cat fixnum fixnum) :clause-1))

((and (:cat fixnum integer)
      (:not ... T1...)) :clause-2)

((:and (:cat (or string fixnum) number)
       (:not ... T1...)
       (:not ... T2...)) :clause-3)
We can *merge* the three disjoint automata into one single automata. Worst-case run-time is divided by 3.
Easy, because fixnum transition is found on each input DFA.
Easy, because string and fixnum are disjoint transitions of state 3.0.
Challenging, because `fixnum` is not found on DFA 3.

(subtypep `fixnum` `integer`)?
Consequence of `subtypep` returning `nil, nil`

Every time `subtypep` returns `nil, nil` the risk is that the remaining automata size doubles.
DFA representing synchronized-cross-product of rte-case
Macro DEFMETHOD

Syntax:

defmethod function-name {method-qualifier}* specialized-lambda-list [[declaration* | documentation]] form*

=> new-method

function-name ::= {symbol | (setf symbol)}

method-qualifier ::= non-list

specialized-lambda-list ::= ({var | (var parameter-specializer-name)}*
                         [&optional {var | (var [initform [supplied-p-parameter] ])}]*
                         [&rest var]
                         [&key {var | ({var | (keywordvar)} [initform [supplied-p-parameter] ])}*
                                [&allow-other-keys] ]
                         [&aux {var | (var [initform] )}] )

parameter-specializer-name ::= symbol | (eql eql-specializer-form)
All the valid `defmethod` forms which are unaccounted for.
All the remaining ways a valid defmethod form can appear, some accounted for in the destructuring-case and some accounted for.
Our implementation of an $N$-clause destructuring-case reduces the number of traversals of the sequence in question from $N + 1$ to 2, once for discrimination, and one for binding.

The code is available from quicklisp via package :rte.
Lots more to be done: benchmarking, connection to method dispatch...

There are two Clojure libraries seqspec and spec which seem very related. According to the author of seqspec, seqspec does not optimize using finite automata because of some annoying limitations of the JVM.

Thanks to Didier Verna for begin my PhD advisor for the past 3 years. Also thanks to Robert Strandh, Pascal Costanza, and Christophe Rhodes for serving on my PhD defense committee.
Questions?

EPITA
ÉCOLE D’INGÉNIEURS EN INFORMATIQUE

LRDE