

Type-checking on Heterogeneous Sequences in Common Lisp

Jim Newton

EPITA/LRDE

May 9, 2016



Overview

1 Introduction

- Common Lisp Types

2 Type Sequences

- Limitations
- Rational Type Expression
- Generated Code
- Example: destructuring-case
- Overlapping Types

3 Conclusion

- Difficulties Encountered
- Summary
- Questions

Common Lisp Types

What is a *type* in Common Lisp?

Definition (from CL specification)

A (possibly infinite) set of objects.

Definition (type specifier)

An expression that denotes a type.

Atomic examples

`t`, `integer`, `number`, `asdf:component`

Type specifiers come in several forms.

- Compound type specifiers
 - `(eql 12)`
 - `(member :x :y :z)`
 - `(satisfies oddp)`
 - `(and (or number string) (not (satisfies MY-FUN)))`

Type specifiers come in several forms.

- Compound type specifiers
 - `(eql 12)`
 - `(member :x :y :z)`
 - `(satisfies oddp)`
 - `(and (or number string) (not (satisfies MY-FUN)))`
- Specifiers for the empty type
 - `nil`
 - `(and number string)`
 - `(and (satisfies evenp) (satisfies oddp))`

Using types with sequences

Compile time

```
(lambda (x y)
  (declare (type (vector float) x y))
  (list x y))
```

Run time

```
(typep my-list '(cons t (cons t (cons string))))
```

Limitations

Limited capability for specifying heterogeneous sequences.
You can't specify the following.

- An arbitrary length, non-empty, list of floats:
(1.0 2.0 3.0)

Limited capability for specifying heterogeneous sequences.
You can't specify the following.

- An arbitrary length, non-empty, list of floats:
`(1.0 2.0 3.0)`
- A plist such as:
`(:x 0 :y 2 :z 3)`

The Rational Type Expression

Introducing the RTE type

Rational type expression vs. RTE type specifier

- $number^+$
 - (RTE `(:+ number)`)
 - Example: `(1.0 2.0 3.0)`

Introducing the RTE type

Rational type expression vs. RTE type specifier

- $number^+$
 - (RTE `(:+ number)`)
 - Example: `(1.0 2.0 3.0)`
- $(keyword \cdot integer)^*$
 - (RTE `(:* (:cat keyword integer))`)
 - Example: `(:x 0 :y 2 :z 3)`

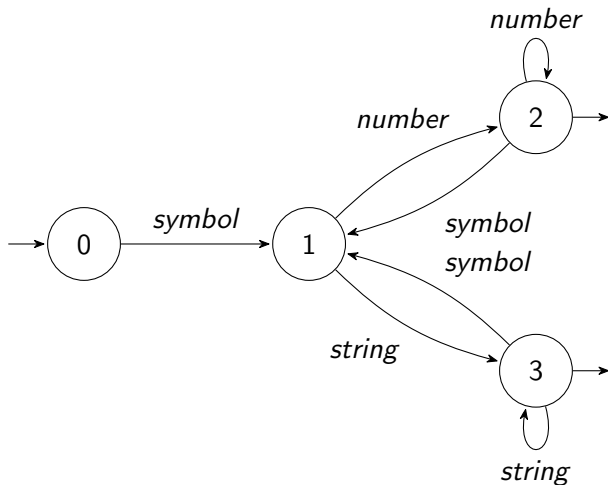
Use RTE anywhere CL expects a type specifier.

```
(typedef plist (type)
  '(and list
    (RTE (:* keyword ,type))))

(defun foo (A B)
  (declare (type (RTE (:+ number)) A)
    (type (plist float) B))
  ...)
```

An RTE can be expressed as a finite state machine.

$$(symbol \cdot (number^+ \cup string^+))^+$$

$$(:+ symbol (:or (:+ number) (:+ string)))$$


Generated Code

State machine can be expressed in CL code

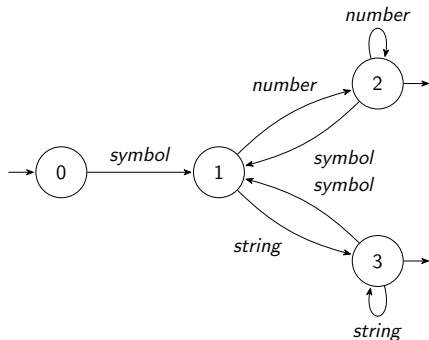
```
(lambda (seq)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (typecase seq
    (list
     ...)
    (simple-vector
     ...)
    (vector
     ...)
    (sequence
     ...)
    (t nil)))
```

Code generating implementing state machine

```

(tagbody
0
  (unless seq (return nil))
  (typecase (pop seq)
    (symbol (go 1))
    (t (return nil))))
1
  (unless seq (return nil))
  (typecase (pop seq)
    (number (go 2))
    (string (go 3))
    (t (return nil))))
2
  (unless seq (return t))
  (typecase (pop seq)
    (number (go 2))
    (symbol (go 1))
    (t (return nil))))

```



```

3
  (unless seq (return t))
  (typecase (pop seq)
    (string (go 3))
    (symbol (go 1))
    (t (return nil))))))

```

destructuring-case

Example of destructuring-case

```
(destructuring-case DATA
  ;; Case-1
  ((a b &optional (c ""))
   (declare (type integer a)
             (type string b c))
   ...))

;; Case-2
((a (b c)
  &key (x t) (y "") z
  &allow-other-keys)
 (declare (type fixnum a b c)
          (type symbol x)
          (type string y)
          (type list z))
 ...))
```

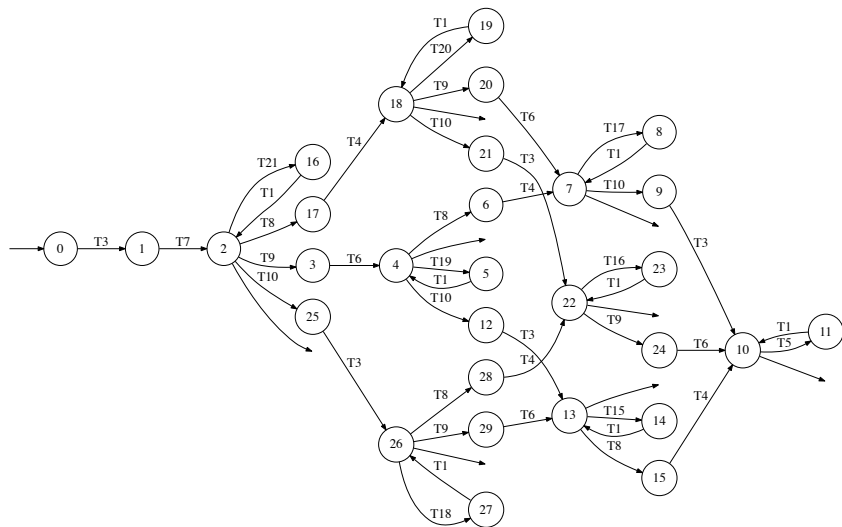
```
(typecase DATA
  ;; Case-1
  ((rte (:cat integer
            string
            (:? string)))
   ...destructuring-bind...))

;; Case-2
((rte ...complicated...
  ...destructuring-bind...
  ...))
```

Regular type expression denoting Case-2

```
(:cat
  (:cat fixnum (:and list (rte (:cat fixnum fixnum))))
  (:and (:* keyword t)
    (:cat
      (:* (not (member :x :y :z)) t)
      (:or :empty-word
        (:cat (eql :z) fixnum (:* (not (member :x :y)) t)
          (:? (eql :y) string (:* (not (eql :x)) t)
            (:? (eql :x) symbol (:* t t))))
          (:cat (eql :z) fixnum (:* (not (member :x :y)) t)
            (:? (eql :x) symbol (:* (not (eql :y)) t)
              (:? (eql :y) string (:* t t))))
          (:cat (eql :y) string (:* (not (member :x :z)) t)
            (:? (eql :z) fixnum (:* (not (eql :x)) t)
              (:? (eql :x) symbol (:* t t))))
          (:cat (eql :x) symbol (:* (not (member :y :z)) t)
            (:? (eql :z) fixnum (:* (not (eql :y)) t)
              (:? (eql :y) string (:* t t))))
          (:cat (eql :y) string (:* (not (member :x :z)) t)
            (:? (eql :x) symbol (:* (not (eql :z)) t)
              (:? (eql :z) fixnum (:* t t))))
          (:cat (eql :x) symbol (:* (not (member :y :z)) t)
            (:? (eql :y) string (:* (not (eql :z)) t)
              (:? (eql :z) fixnum (:* t t)))))))))))))
```

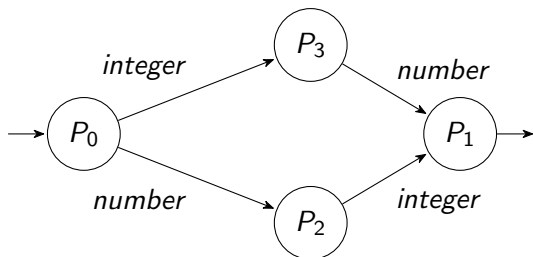
Finite State Machine of Case-2 of destructuring-case



Overlapping Types

Rational type expression with *overlapping* types
$$((integer \cdot number) \cup (number \cdot integer))$$

```
(:or (:cat integer number)
      (:cat number integer))
```

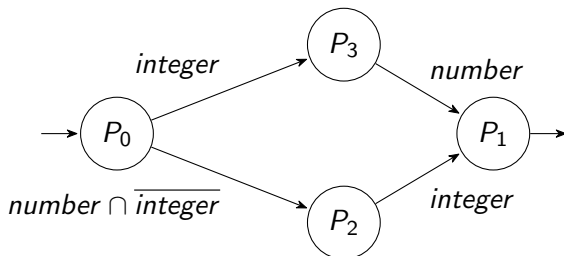


Overlapping types must be decomposed into *disjoint* types

$$((integer \cdot number) \cup ((number \cap \overline{integer}) \cdot integer))$$

```
(:or (:cat integer number)
```

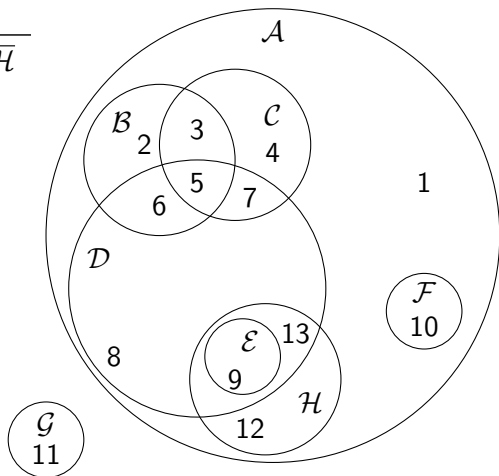
```
  (:cat (and number (not integer))
        integer))
```



Overlapping types considered harmful

Disjoint Set	Decomposed Expression
--------------	-----------------------

{ 1 }	$A \cap \bar{B} \cap \bar{C} \cap \bar{D} \cap \bar{F} \cap \bar{H}$
{ 2 }	$B \cap \bar{C} \cap \bar{D}$
{ 3 }	$B \cap C \cap \bar{D}$
{ 4 }	$C \cap \bar{B} \cap \bar{D}$
{ 5 }	$B \cap C \cap D$
{ 6 }	$B \cap D \cap \bar{C}$
{ 7 }	$C \cap D \cap \bar{B}$
{ 8 }	$D \cap \bar{B} \cap \bar{C} \cap \bar{H}$
{ 9 }	\mathcal{E}
{ 10 }	\mathcal{F}
{ 11 }	\mathcal{G}
{ 12 }	$\mathcal{H} \cap \bar{D}$
{ 13 }	$D \cap \mathcal{H} \cap \bar{\mathcal{E}}$



How to calculate type disjoint-ness and equivalence.

```
(defun type-intersection (T1 T2)
  '(and ,T1 ,T2))
```

```
(defun types-disjoint-p (T1 T2)
  (subtypep (type-intersection T1 T2) nil))
```

```
(defun types-equivalent-p (T1 T2)
  (multiple-value-bind (T1<=T2 okT1T2) (subtypep T1 T2)
    (multiple-value-bind (T2<=T1 okT2T2) (subtypep T2 T1)
      (values (and T1<=T2 T2<=T1) (and okT1T2 okT2T2))))))
```

Interesting Difficulties Encountered

Performance and correctness problems with SUBTYPEP

```
(subtypep '(and integer (or (eql 1) (satisfies F)))  
          '(and integer (or (eql 0) (satisfies G))))
```

\implies *NIL, T (should be NIL, NIL)*

```
(subtypep 'compiled-function nil)
```

\implies *NIL, NIL (should be NIL, T)*

```
(subtypep '(eql :x) 'keyword)
```

\implies *NIL, NIL (should be T, T)*

Recursive types forbidden

Neither the CL type system nor the RTE extension are expressive enough to specify recursive types such as:

```
(deftype singleton (type)
  '(or (cons ,type nil)
        (cons (singleton ,type))))
```

```
(deftype proper-list (type)
  '(cons ,type (or null
                 (proper-list ,type))))
```

Missing CL API for type reflection and extension

- Can't ask whether a particular type exists? *I.e.*, is there a type `foo` ?
- *E.g.*, Given two RTE type specifiers, we can calculate whether one is a subtype of the other. Unfortunately, CL provides no `SUBTYPE` hook allowing me to make this calculation.

Future Research

- Static analysis of destructuring-case to detect unreachable code or overlapping cases.
- Investigate performance of type decomposition (disjoint-izing).
- Apply to other dynamic languages (e.g., Python, Scala/JVM, Julia/LLVM).

Summary

- Regular expression style type-based pattern matching on CL sequences.
- RTE type allows $\mathcal{O}(n)$ type checking of CL sequences.
- Non-linear complexity moved to compile time.
- Source-code available at <https://www.lrde.epita.fr/wiki/Publications/newton.16.els>

Q/A

Questions?

