Implementing Baker's `SUBTYPEP` decision procedure

Léo Valais

April 1st, 2019

European Lisp Symposium

Introduction

Common Lisp type system, `subtypep`
& Baker's decision procedure

The Common Lisp type system

- Types $\rightarrow$ sets, subtypes $\rightarrow$ subsets
- S-expression based, inductive Domain-Specific Language $\rightarrow$ *type specifiers*
- Examples
  - Atomic $\rightarrow$ `string`, `integer`, `my-class`, ...
  - Compound form
    - `(or string number)` $\equiv$ `string` $\cup$ `number`
    - `(unsigned-byte 10)` $\equiv \{0, 1, \cdots, 2^{10} - 1\}$
    - `(array real (3 3))` $\equiv \mathcal{M}_{3,3}(\mathbb{R})$
    - Many more!

Use case

$$\forall M \in \mathcal{M}_{3,3}(\mathbb{R}), tr(M) = \sum_{i=1}^{3} M_{i,j}$$

λ Common Lisp

```lisp
(defun tr (M)
  (declare (type (array real (3 3)) M))
  (+ (aref M 0 0)
     (aref M 1 1)
     (aref M 2 2)))
```

Use case

$$\forall M \in \mathcal{M}_{3,3}(\mathbb{R}), tr(M) = \sum_{i=1}^{3} M_{i,j}$$

λ Common Lisp

```lisp
(defun tr (M)
  (declare (type (array real (3 3)) M))
  (+ (aref M 0 0)
     (aref M 1 1)
     (aref M 2 2)))
```

- Type checking
- Value checking
- Compiler optimization
- Documentation

What about subtyping?

- (`subtypep` $\langle A \rangle$ $\langle B \rangle$) $\equiv A \subseteq B$?
- Predicate function

What about subtyping?

- (subtypep ⟨A⟩ ⟨B⟩) $\equiv A \subseteq B$?
- Predicate function

λ Quite easy

```
(subtypep '(and integer
                (not float))
          '(or number string))
```

What about subtyping?

- (`subtypep` $\langle A \rangle$ $\langle B \rangle$) $\equiv A \subseteq B$?
- Predicate function

λ Not that easy after all...

```
(subtypep '(or my-class string (integer 0 (1024)))
          '(or super-class
               (array * 1)
               (unsigned-byte 10)))
```

What about subtyping?

- (subtypep ⟨A⟩ ⟨B⟩) $\equiv A \subseteq B$?
- Predicate function

λ *“Oh dear, we are in trouble”* 😢

```
(subtypep '(or string
               my-class
               (and integer
                    (not (unsigned-byte 10)))
               (member 3.14 2.71))
          '(and (array * (* * 8 *))
                bit-vector
                (not (eql :some-keyword))))
```

What about subtyping?

- `(subtypep` $\langle A \rangle$ $\langle B \rangle$`)` $\equiv A \subseteq B$?
- Predicate function

- Type specifiers arbitrarily deep
- May take a while to answer...

Problem #1 — complex input

Arbitrarily complex input type specifiers

```
bit-vector
(not (eql :some-keyword))))
```

`satisfies` type specifier

- `(satisfies` $\langle \mathit{predicate} \rangle$`)` $\equiv \{x \mid \mathit{predicate}(x)\}$
- `(satisfies oddp)` $\rightarrow$ all odd numbers

`satisfies` type specifier

- `(satisfies` $\langle \mathit{predicate} \rangle$`)` $\equiv \{x \mid \mathit{predicate}(x)\}$
- `(satisfies oddp)` $\rightarrow$ all odd numbers
- `(subtypep '(satisfies oddp) '(satisfies evenp))`

`satisfies` type specifier

- `(satisfies` $\langle \mathit{predicate} \rangle$`)` $\equiv \{x \mid \mathit{predicate}(x)\}$
- `(satisfies oddp)` $\rightarrow$ all odd numbers
- `(subtypep '(satisfies oddp) '(satisfies evenp))`
- `(subtypep '(satisfies` $\langle F \rangle$`) '(satisfies` $\langle G \rangle$`))`
  - arbitrary predicates F and G
- halting problem $\rightarrow$ `subtypep` *cannot* even answer 😱

`satisfies` type specifier

- `(satisfies` $\langle predicate \rangle$`)` $\equiv \{x \mid predicate(x)\}$
- `(satisfies oddp)` $\rightarrow$ all odd numbers
- `(subtypep '(satisfies oddp) '(satisfies evenp))`
- `(subtypep '(satisfies` $\langle F \rangle$`) '(satisfies` $\langle G \rangle$`))`
  - arbitrary predicates F and G
- halting problem $\rightarrow$ `subtypep` *cannot* even answer 😱

Problem #2 — undecidability

Subtypep cannot answer for some type specifiers

`subtypep`

$$(\texttt{subtypep}\ \langle A\rangle\ \langle B\rangle) = \begin{cases} \texttt{(T T)} & \rightarrow A \subseteq B \\ \texttt{(NIL T)} & \rightarrow A \not\subseteq B \\ \texttt{(NIL NIL)} & \rightarrow \text{“I can't answer”} \end{cases}$$

- `(NIL NIL)` encodes undecidability

subtypep

$$(\texttt{subtypep}\ \langle A\rangle\ \langle B\rangle) = \begin{cases} \texttt{(T T)} & \to A \subseteq B \\ \texttt{(NIL T)} & \to A \not\subseteq B \\ \texttt{(NIL NIL)} & \to \text{“I give up, sorry 🤕”} \end{cases}$$

- `(NIL NIL)` encodes ~~undecidability~~“input too complex”

subtypep

$$\texttt{(subtypep}\ \langle A\rangle\ \langle B\rangle\texttt{)} = \begin{cases} \texttt{(T T)} & \rightarrow A \subseteq B \\ \texttt{(NIL T)} & \rightarrow A \not\subseteq B \\ \texttt{(NIL NIL)} & \rightarrow \text{“I give up, sorry 🤕”} \end{cases}$$

- `(NIL NIL)` encodes ~~undecidability~~“input too complex”
- Lack of reliability
- Painful limit for some applications
  - Newton’s regular type expressions
  - Newton’s optimized `typecase` implementation

Baker's decision procedure

- \+ focus on result accuracy
- \+ *never* returns `(NIL NIL)` uselessly
- − paper difficult to read
- − not exhaustive
- − very few solutions about `satisfies`

- \+ efficiency
- − *not* open source
- \+ Divide and Conquer
- − exponential complexity (theoretical)

Baker's decision procedure

- + focus on result accuracy
- + *never* returns `(NIL NIL)` uselessly
- − paper difficult to read
- − not exhaustive
- − very few solutions about `satisfies`

- + efficiency
- − *not* open source
- + Divide and Conquer
- − exponential complexity (theoretical)

Baker's decision procedure

- \+ focus on result accuracy
- \+ *never* returns `(NIL NIL)` uselessly
- − paper difficult to read
- − not exhaustive
- − very few solutions about `satisfies`

- \+ efficiency
- − *not* open source
- \+ Divide and Conquer
- − exponential complexity (theoretical)

Baker's decision procedure

- + focus on result accuracy
- + *never* returns `(NIL NIL)` uselessly
- − paper difficult to read
- − not exhaustive
- − very few solutions about `satisfies`

- + efficiency
- − *not* open source
- + Divide and Conquer
- − exponential complexity (theoretical)

Application

Use case of `subtypep`

The problem

- Serialize CLOS instances → JSON
- Automatic JSON object construction

λ Common Lisp

```lisp
(defclass point ()
  ((x :type number
      :initarg :x)
   (y :type number
      :initarg :y)
   (name :type string
         :initarg :name))
  (:metaclass json-serializable))

(json-serialize (make-instance 'point :x -10 :y 3.2 :name "A1"))
```

The problem

- Serialize CLOS instances → JSON
- Automatic JSON object construction

λ Common Lisp

```lisp
(defclass point ()
  ((x :type number
      :initarg :x)
   (y :type number
      :initarg :y)
   (name :type string
         :initarg :name))
  (:metaclass json-serializable))

(json-serialize (make-instance
↪  'point :x -10 :y 3.2 :name
↪  "A1"))
```

JSON serialization

```json
{
  "X": -10,
  "Y": 3.2,
  "NAME": "A1"
}
```

CLOS setup

λ Common Lisp

```lisp
(defclass json-serializable (standard-class)
  ())

(defmethod validate-superclass
    ((class json-serializable) (superclass standard-class))
  t)
(defmethod validate-superclass
    ((class standard-class) (superclass json-serializable))
  t)

(defgeneric json-serialize (instance))
```

CLOS setup

λ Common Lisp

```lisp
(defclass json-serializable (standard-class)
  ())

(defmethod validate-superclass
    ((class json-serializable) (superclass standard-class))
  t)
(defmethod validate-superclass
    ((class standard-class) (superclass json-serializable))
  t)

(defgeneric json-serialize (instance))
```

- No restriction on subclassing
- But restrictions on existence!

`json-serializable` existence condition

- Slots
  - 👍 names $\rightarrow$ symbols
  - 👎 values $\rightarrow$ virtually *any type*

`json-serializable` existence condition

- Slots
  - names → symbols
  - values → virtually *any type*

λ Common Lisp

```lisp
(deftype json ()
  '(or number
       string
       (member :true
               :false
               :null)
       (and symbol
            (not keyword))
       list
       hash-table))
```

`json-serializable` existence condition

- Slots
  - 👍 names $\rightarrow$ symbols
  - 👎 values $\rightarrow$ virtually *any type*
- Types of slots $\rightarrow u_1, u_2, \ldots, u_n$
  - $u_i \subseteq$ `json`
  - $\Rightarrow$ `(subtypep` u_i `'json)`
- Trigger compile-time error

λ Common Lisp

```lisp
(deftype json ()
  '(or number
       string
       (member :true
               :false
               :null)
       (and symbol
            (not keyword))
       list
       hash-table))
```

json-serializable existence check

λ Common Lisp

```lisp
(defun json-compatible-class-p (class)
  (let* ((slots (class-slots class))
         (types (mapcar #'slot-definition-type slots)))
    (every (lambda (slot-type)
             (subtypep slot-type 'json))
           types)))

(defmethod initialize-instance ((class json-serializable)
                                &rest args)
  (let ((class (call-next-method)))
    (closer-mop:ensure-finalized class nil)
    (unless (json-compatible-class-p class)
      (error "class ~a is not JSON-compatible" class))
    cls))
```

employee class

λ Common Lisp

```lisp
(defclass employee ()
  ((name :type (or string
                   (and symbol
                        (not keyword))
                   unsigned-byte))
   (half-time-p (or boolean
                    (member :true
                            :false))))
  (:metaclass json-serializable))
```

employee class

λ Common Lisp

```
(defclass employee ()
  ((name :type (or string
                   (and symbol
                        (not keyword))
                   unsigned-byte))
   (half-time-p (or boolean
                    (member :true
                            :false))))
  (:metaclass json-serializable))
```

- 2 subtypep calls → one per slot

Pre-processing

Simplifying the problem

Pre-processing steps

λ name's type verification

```
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          'json)
```

Pre-processing steps

λ name's type verification

```
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          '(or number
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not keyword))
               list
               hash-table))
```

- alias expansion
  - implementation dependant feature
  - `sb-ext:typexpand`

Pre-processing steps

λ name's type verification

```
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          '(or number
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not keyword))
               list
               hash-table))
```

- alias expansion
  - implementation dependant feature
  - `sb-ext:typexpand`
- more preprocessing!
  - syntactic sugar elimination
  - numeric types specific actions

Pre-processing steps

λ name's type verification

```
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          '(or number
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not keyword))
               list
               hash-table))
```

- alias expansion
  - implementation dependant feature
  - `sb-ext:typexpand`
- more preprocessing!
  - syntactic sugar elimination
  - ~~numeric types specific actions~~

Pre-processing steps

λ name's type verification

```lisp
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          '(or number
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not keyword))
               list
               hash-table))
```

- alias expansion
  - implementation dependant feature
  - `sb-ext:typexpand`
- more preprocessing!
  - syntactic sugar elimination
  - ~~numeric types specific actions~~
- splitting
  - "litteral" types
  - "numeric" types

Pre-processing steps

λ name's type verification

```lisp
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          '(or number
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not keyword))
               list
               hash-table))
```

- alias expansion
  - implementation dependant feature
  - `sb-ext:typexpand`
- more preprocessing!
  - syntactic sugar elimination
  - ~~numeric types specific actions~~
- splitting
  - "litteral" types
  - "numeric" types
- subtyping equivalence

Problem reformulation

Litteral types splitting

```
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          '(or number
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not keyword))
               list
               hash-table))
```

Numeric types splitting

```
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          '(or number
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not keyword))
               list
               hash-table))
```

Problem reformulation

Litteral types splitting

```
(subtypep '(or string
              (and symbol
                   (not keyword))
              NIL)
          '(or NIL
              string
              (member :true
                      :false
                      :null)
              (and symbol
                   (not keyword))
              list
              hash-table))
```

Numeric types splitting

```
(subtypep '(or string
              (and symbol
                   (not keyword))
              unsigned-byte)
          '(or number
              string
              (member :true
                      :false
                      :null)
              (and symbol
                   (not keyword))
              list
              hash-table))
```

Problem reformulation

Litteral types splitting

```
(subtypep '(or string
               (and symbol
                    (not keyword))
               NIL)
          '(or NIL
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not keyword))
               list
               hash-table))
```

Numeric types splitting

```
(subtypep '(or NIL
               (and NIL
                    (not NIL))
               unsigned-byte)
          '(or number
               NIL
               NIL
               (and NIL
                    (not NIL))
               NIL
               NIL))
```

Problem reformulation

```
(and (subtypep '(or string
                    (and symbol
                         (not keyword))
                    NIL)
               '(or NIL
                    string
                    (member :true :false :null)
                    (and symbol
                         (not keyword))
                    list
                    hash-table))
     (subtypep '(or NIL
                    (and NIL
                         (not NIL))
                    unsigned-byte)
               '(or number
                    NIL
                    NIL
                    (and NIL
                         (not NIL))
                    NIL
                    NIL)))
```

Problem reformulation

❝ Litte… …ing

```
(subtype…
…NIL))
```

```
(and (subtypep '(or string
                    (and symbol
                         (not keyword))
                    NIL)
               '(or NIL
                    string
                    (member :true :false :null)
                    (and symbol
...
                    unsigned-byte)
               '(or number
                    NIL
                    NIL
                    (and NIL
                         (not NIL))
                    NIL
                    NIL)))
```

Subtyping equivalence

- $A \subseteq B \Leftrightarrow A \cap \overline{B} = \emptyset$
- (subtypep $\langle A \rangle$ $\langle B \rangle$) $\equiv$ (subtypep '(and $\langle A \rangle$ (not $\langle B \rangle$)) nil)

```
(and (subtypep '(AND (or string
                         (and symbol
                              (not keyword))
                         NIL)
                     (NOT (or NIL
                              string
                              (member :true :false :null)
                              (and symbol
                                   (not keyword))
                              list
                              hash-table)))
               NIL)
     (subtypep '(AND (or NIL
                         (and NIL
                              (not NIL))
                         unsigned-byte)
                     (NOT (or number
                              NIL
                              NIL
                              (and NIL
                                   (not NIL))
                              NIL
                              NIL)))
               NIL))
```

Problem reformulation

```
(and (NULL-LITERAL-TYPE-P '(AND (or string
                                    (and symbol
                                         (not keyword))
                                    NIL)
                                (NOT (or NIL
                                         string
                                         (member :true :false
                                          ↪   :null)
                                         (and symbol
                                              (not keyword))
                                         list
                                         hash-table))))
     (NULL-NUMERIC-TYPE-P '(AND (or NIL
                                    (and NIL
                                         (not NIL))
                                    unsigned-byte)
                                (NOT (or number
                                         NIL
                                         NIL
                                         (and NIL
                                              (not NIL))
                                         NIL
                                         NIL)))))
```

Problem reformulation

```
(and (NULL-LITERAL-TYPE-P '(AND (or string
                                   (and symbol
                                        (not keyword))
                                   NIL)
                               (NOT (or NIL
                                        string
                                        (member :true :false
```

Lit

```
(su
```

Preprocessing summary

- alias (`deftype`) expansion
- splitting across “type kingdoms”
- $A \subseteq B \Leftrightarrow A \cap \overline{B} = \emptyset$
- *specialized sub-procedures*
  - `null-literal-type-p`
  - ~~`null-numeric-type-p`~~

```
                                        (and NIL
                                             (not NIL))
                                        NIL
                                        NIL)))))
```

Literal types specialized sub-procedure

Primitive types, `member` type specifiers, CLOS classes

The Matrix™

Some assumptions

- We can enumerate all Common Lisp values $\rightarrow e_1, e_2, \ldots, e_\omega$

Some assumptions

- We can enumerate all Common Lisp values $\rightarrow e_1, e_2, \ldots, e_\omega$
- We can enumerate all combinations of these $\rightarrow u_1, u_2, \ldots, u_\omega$

Some assumptions

- We can enumerate all Common Lisp values $\rightarrow e_1, e_2, \ldots, e_\omega$
- We can enumerate all combinations of these $\rightarrow u_1, u_2, \ldots, u_\omega$
  - all Common Lisp types!

Some (unrealistic 🙈) assumptions

- We can enumerate all Common Lisp values $\rightarrow e_1, e_2, \ldots, e_\omega$
- We can enumerate all combinations of these $\rightarrow u_1, u_2, \ldots, u_\omega$
  - all Common Lisp types!

The Matrix™

Some (unrealistic 🙈) assumptions

- We can enumerate all Common Lisp values → $e_1, e_2, \ldots, e_\omega$
- We can enumerate all combinations of these → $u_1, u_2, \ldots, u_\omega$
  - all Common Lisp types!

$$\begin{aligned}
\emptyset &= \{\} \quad \text{(a.k.a. } \texttt{nil}\text{)} \\
u_1 &= \{e_1, e_3, e_4\} \\
u_2 &= \{e_1\} \\
&\vdots \\
u_\omega &= \{e_2, e_4, \cdots\}
\end{aligned}$$

The Matrix™

Some (unrealistic 🙈) assumptions

- We can enumerate all Common Lisp values → $e_1, e_2, \ldots, e_\omega$
- We can enumerate all combinations of these → $u_1, u_2, \ldots, u_\omega$
  - all Common Lisp types!

$$e_i \in u_j \Leftrightarrow \mathcal{M}_{j,i} = 1$$

$$\emptyset = \{\} \quad \text{(a.k.a. } \texttt{nil}\text{)}$$

$$u_1 = \{e_1, e_3, e_4\}$$

$$u_2 = \{e_1\}$$

$$\vdots$$

$$u_\omega = \{e_2, e_4, \cdots\}$$

$$\begin{array}{c c}
 & \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & \cdots & e_\omega \end{array} \\
\begin{array}{c} \mathcal{B}_\emptyset \\ \mathcal{B}_1 \\ \mathcal{B}_2 \\ \vdots \\ \mathcal{B}_\omega \end{array} &
\begin{pmatrix}
0 & 0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 1 & 1 & \cdots & 0 \\
1 & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 1 & 0 & 1 & \cdots & 0
\end{pmatrix}
\end{array}$$

The Matrix™

Properties

- $\mathcal{B}_i \to$ bit-vector representing the type u_i
- $u_i \cup u_j \to \mathcal{B}_i \vee \mathcal{B}_j$ (bitwise)
- $u_i \cap u_j \to \mathcal{B}_i \wedge \mathcal{B}_j$ (bitwise)
- $\overline{u_i} \to \neg\mathcal{B}_i$ (bitwise)

$$\emptyset = \{\} \quad \text{(a.k.a. } \texttt{nil}\text{)}$$

$$u_1 = \{e_1, e_3, e_4\}$$

$$u_2 = \{e_1\}$$

$$\vdots$$

$$u_\omega = \{e_2, e_4, \cdots\}$$

$$\begin{array}{c c}
 & \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & \cdots & e_\omega \end{array} \\
\begin{array}{c} \mathcal{B}_\emptyset \\ \mathcal{B}_1 \\ \mathcal{B}_2 \\ \vdots \\ \mathcal{B}_\omega \end{array} &
\begin{pmatrix}
0 & 0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 1 & 1 & \cdots & 0 \\
1 & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 1 & 0 & 1 & \cdots & 0
\end{pmatrix}
\end{array}$$

null-literal-type-p

$$
\begin{aligned}
\texttt{(subtypep } \langle u_i \rangle \ \langle u_j \rangle\texttt{)} &\equiv u_i \subseteq u_j \\
&\Leftrightarrow u_i \cap \overline{u_j} = \emptyset \\
&\equiv \texttt{(null-literal-type-p '(and } \langle u_i \rangle \texttt{ (not } \langle u_j \rangle\texttt{)))} \\
&\Leftrightarrow \mathcal{B}_i \wedge \neg \mathcal{B}_j = \mathcal{B}_\emptyset
\end{aligned}
$$

`null-literal-type-p`

$$\begin{aligned}
(\texttt{subtypep}\ \langle u_i \rangle\ \langle u_j \rangle) &\equiv u_i \subseteq u_j \\
&\Leftrightarrow u_i \cap \overline{u_j} = \emptyset \\
&\equiv (\texttt{null-literal-type-p '(and}\ \langle u_i \rangle\ \texttt{(not}\ \langle u_j \rangle\texttt{)))} \\
&\Leftrightarrow \mathcal{B}_i \wedge \neg \mathcal{B}_j = \mathcal{B}_\emptyset
\end{aligned}$$

- All about matrix lookups & bitwise operations on bit-vectors
- Very fast, but…

null-literal-type-p

$$\begin{aligned}
\texttt{(subtypep } \langle u_i \rangle\ \langle u_j \rangle\texttt{)} &\equiv u_i \subseteq u_j \\
&\Leftrightarrow u_i \cap \overline{u_j} = \emptyset \\
&\equiv \texttt{(null-literal-type-p '(and } \langle u_i \rangle\ \texttt{(not } \langle u_j \rangle\texttt{)))} \\
&\Leftrightarrow \mathcal{B}_i \wedge \neg \mathcal{B}_j = \mathcal{B}_\emptyset
\end{aligned}$$

- All about matrix lookups & bitwise operations on bit-vectors
- Very fast, but...
- ...still an *infinite* matrix! 😔

Getting finite

$$
\begin{array}{c|c}
 & \begin{array}{ccccccc} e_1 & e_2 & e_3 & e_4 & \cdots & e_N & \cdots \end{array} \\
\begin{array}{c} \mathcal{B}_\emptyset \\ \mathcal{B}_1 \\ \mathcal{B}_2 \\ \vdots \\ \mathcal{B}_M \\ \vdots \end{array} &
\begin{pmatrix}
0 & 0 & 0 & 0 & \cdots & 0 & \cdots \\
1 & 0 & 1 & 1 & \cdots & 0 & \cdots \\
1 & 0 & 0 & 0 & \cdots & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots \\
0 & 1 & 0 & 1 & \cdots & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots
\end{pmatrix}
\end{array}
$$

Getting finite

$$\begin{array}{c|c} & \begin{array}{ccccccc} e_1 & e_2 & e_3 & e_4 & \cdots & e_N & \cdots \end{array} \\ \begin{array}{c} \mathcal{B}_\emptyset \\ \mathcal{B}_1 \\ \mathcal{B}_2 \\ \vdots \\ \mathcal{B}_M \end{array} & \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 & \cdots \\ 1 & 0 & 1 & 1 & \cdots & 0 & \cdots \\ 1 & 0 & 0 & 0 & \cdots & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots \\ 0 & 1 & 0 & 1 & \cdots & 0 & \cdots \end{pmatrix} \end{array}$$

- Types (u_k and associated $\mathcal{B}_k$)
  - only those in `subtypep` call
  - `nil` is always involved (somehow)

Getting finite

$$\begin{array}{c c}
 & \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & \cdots & e_N \end{array} \\
\begin{array}{c} \mathcal{B}_\emptyset \\ \mathcal{B}_1 \\ \mathcal{B}_2 \\ \vdots \\ \mathcal{B}_M \end{array} &
\begin{pmatrix}
0 & 0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 1 & 1 & \cdots & 0 \\
1 & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 1 & 0 & 1 & \cdots & 0
\end{pmatrix}
\end{array}$$

- Types (u_k and associated $\mathcal{B}_k$)
  - only those in `subtypep` call
  - `nil` is always involved (somehow)
- Values (e_k)
  - only *sufficiently many*
  - distinguish each type from the others

Getting finite

$$
\begin{array}{c} \\ \mathcal{B}_\emptyset \\ \mathcal{B}_1 \\ \mathcal{B}_2 \\ \vdots \\ \mathcal{B}_M \end{array}
\begin{array}{c}
\begin{array}{cccccc} r_1 & r_2 & r_3 & r_4 & \cdots & r_N \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 1 & 1 & \cdots & 0 \\
1 & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 1 & 0 & 1 & \cdots & 0
\end{pmatrix}
\end{array}
$$

- Types (u_k and associated $\mathcal{B}_k$)
  - only those in `subtypep` call
  - `nil` is always involved (somehow)
- Values (e_k)
  - only *sufficiently many*
  - distinguish each type from the others
- Not just “values” $\rightarrow$ *representative* elements

Back to our problem

λ Common Lisp

```
(null-literal-type-p
 '(and (or string
           (and symbol
                (not keyword))
           nil)
       (not (or nil
                string
                (member :true
                        :false
                        :null)
                (and symbol
                     (not keyword))
                list
                hash-table))))
```

Back to our problem

λ Common Lisp

```lisp
(null-literal-type-p
 '(and (or string
           (and symbol
                (not keyword))
           nil)
       (not (or nil
                string
                (member :true
                        :false
                        :null)
                (and symbol
                     (not keyword))
                list
                hash-table))))
```

- Matrix setup
  - primitive types known at compile-time
  - manual representative choice

Back to our problem

The Matrix

$$\begin{array}{cc}
 & \begin{array}{cccccc} \texttt{t} & \texttt{nil} & \texttt{sym} & \texttt{"str"} & \cdots & \texttt{(l i s t)} \end{array} \\
\begin{array}{c} \mathcal{B}_{\texttt{nil}} \\ \mathcal{B}_{\texttt{t}} \\ \mathcal{B}_{\texttt{null}} \\ \mathcal{B}_{\texttt{symbol}} \\ \mathcal{B}_{\texttt{string}} \\ \vdots \\ \mathcal{B}_{\texttt{list}} \end{array} & \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & 0 & 0 & \cdots & 1 \end{pmatrix}
\end{array}$$

Back to our problem

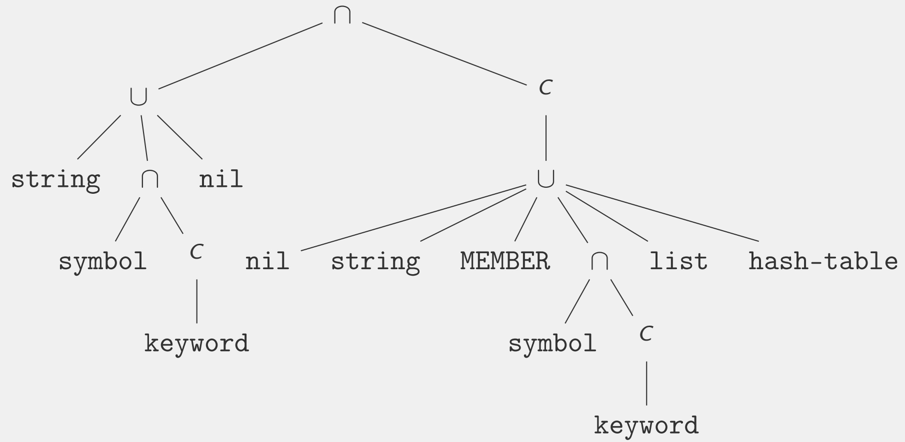
λ Common Lisp

```
(null-literal-type-p
 '(and (or string
           (and symbol
                (not keyword))
           nil)
       (not (or nil
                string
                (member :true
                        :false
                        :null)
                (and symbol
                     (not keyword))
                list
                hash-table))))
```

- Matrix setup
  - primitive types known at compile-time
  - manual representative choice
- Lookup bit-vectors & translate logic operators

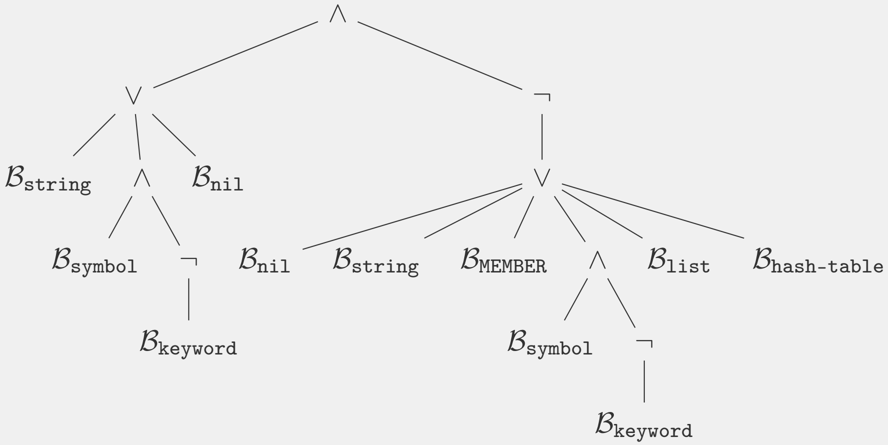
Back to our problem

Bit-vector expression reduction



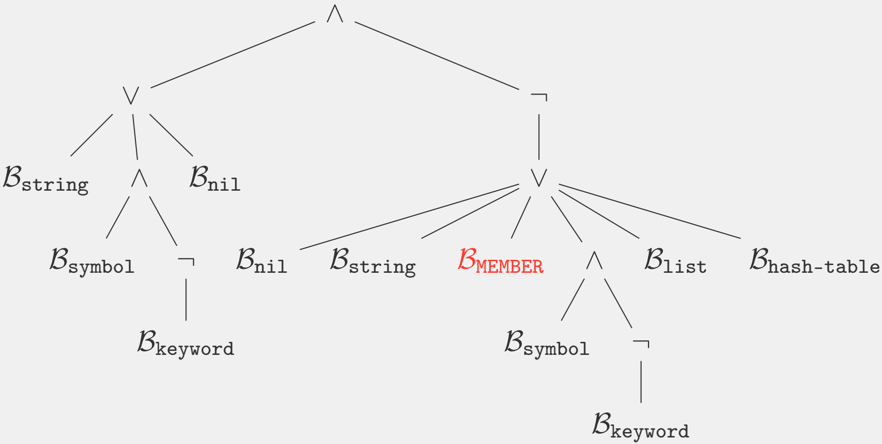
Back to our problem

Bit-vector expression reduction



Back to our problem

Bit-vector expression reduction



Back to our problem

Bit-vector expression reduction

The Matrix

$$
\begin{array}{c c}
 & \begin{array}{cccccc} \texttt{t} & \texttt{nil} & \texttt{sym} & \texttt{"str"} & \cdots & \texttt{(l i s t)} \end{array} \\
\begin{array}{c} \mathcal{B}_{\texttt{nil}} \\ \mathcal{B}_{\texttt{t}} \\ \mathcal{B}_{\texttt{null}} \\ \mathcal{B}_{\texttt{symbol}} \\ \mathcal{B}_{\texttt{string}} \\ \vdots \\ \mathcal{B}_{\texttt{list}} \end{array} &
\begin{pmatrix}
0 & 0 & 0 & 0 & \cdots & 0 \\
1 & 1 & 1 & 1 & \cdots & 1 \\
0 & 1 & 0 & 0 & \cdots & 0 \\
1 & 1 & 1 & 0 & \cdots & 0 \\
0 & 0 & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 1 & 0 & 0 & \cdots & 1
\end{pmatrix}
\end{array}
$$

Back to our problem

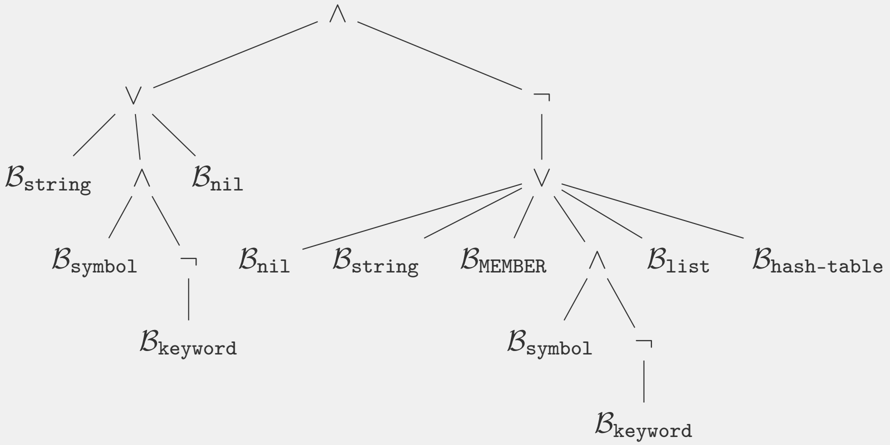
Bit-vector expression reduction

The Matrix

$$
\begin{array}{cc}
 & \begin{array}{ccccccccc} \texttt{t} & \texttt{nil} & \texttt{sym} & \texttt{"str"} & \cdots & \texttt{(l i s t)} & \texttt{:true} & \texttt{:false} & \texttt{:null} \end{array} \\
\begin{array}{c} \mathcal{B}_{\texttt{nil}} \\ \mathcal{B}_{\texttt{t}} \\ \mathcal{B}_{\texttt{null}} \\ \mathcal{B}_{\texttt{symbol}} \\ \mathcal{B}_{\texttt{string}} \\ \vdots \\ \mathcal{B}_{\texttt{list}} \\ \mathcal{B}_{\texttt{MEMBER}} \end{array} &
\begin{pmatrix}
0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & \cdots & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & \cdots & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 1 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \cdots & 0 & 1 & 1 & 1
\end{pmatrix}
\end{array}
$$

Back to our problem

Bit-vector expression reduction



Back to our problem

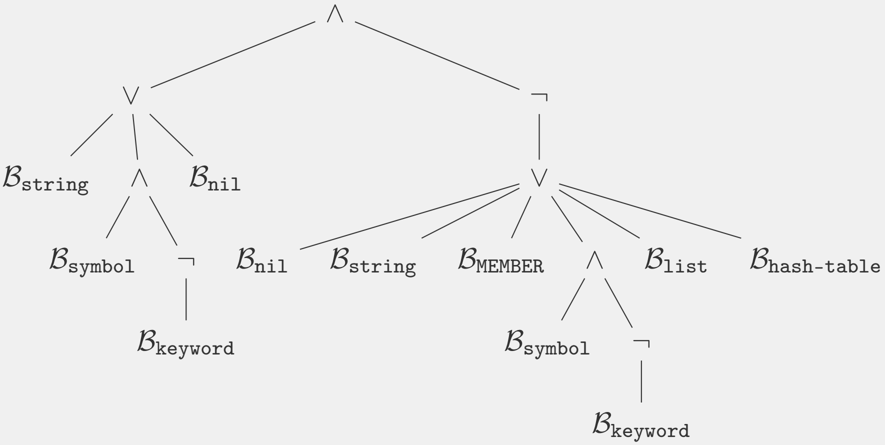
λ Common Lisp

```
(null-literal-type-p
 '(and (or string
           (and symbol
                (not keyword))
           nil)
       (not (or nil
                string
                (member :true
                        :false
                        :null)
                (and symbol
                     (not keyword))
                list
                hash-table))))
```

- Matrix setup
  - primitive types known at compile-time
  - manual representative choice
- Lookup bit-vectors & translate logic operators
- `(member` $\langle a \rangle$ $\langle b \rangle$`)` type specifier bit-vector
  1. register a and b as representatives
  2. $\mathcal{B}_{(\texttt{member}\ \langle a \rangle\ \langle b \rangle)} = \mathcal{B}_{\{a\}} \vee \mathcal{B}_{\{b\}}$

Back to our problem

Bit-vector expression reduction



Back to our problem

λ Common Lisp

```lisp
(null-literal-type-p
 '(and (or string
           (and symbol
                (not keyword))
          nil)
      (not (or nil
               string
               (member :true
                       :false
                       :null)
               (and symbol
                    (not
                     ↪ keyword))
               list
               hash-table))))
```

- Matrix setup
  - primitive types known at compile-time
  - manual representative choice
- Lookup bit-vectors & translate logic operators
- `(member ⟨a⟩ ⟨b⟩)` type specifier bit-vector
  1. register a and b as representatives
  2. $\mathcal{B}_{(\texttt{member}\ \langle a\rangle\ \langle b\rangle)} = \mathcal{B}_{\{a\}} \vee \mathcal{B}_{\{b\}}$
- Eventually reduces to $\mathcal{B}_{\texttt{nil}}$
- `null-literal-type-p` returns true

employee **verification**

```
(defclass employee ()
  ((name :type (or string
                   (and symbol
                        (not keyword))
                   unsigned-byte))
   (half-time-p (or boolean
                    (member :true
                            :false))))
  (:metaclass json-serializable))
```

```
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          'json)
```

employee verification

```
(and (NULL-LITERAL-TYPE-P '(AND (or string
                                    (and symbol
                                         (not keyword))
                                    NIL)
                                (NOT (or NIL
                                         string
                                         (member :true :false :null)
                                         (and symbol
                                              (not keyword))
                                         list
                                         hash-table))))
     (NULL-NUMERIC-TYPE-P '(AND (or NIL
                                    (and NIL
                                         (not NIL))
                                    unsigned-byte)
                                (NOT (or number
                                         NIL
                                         NIL
                                         (and NIL
                                              (not NIL))
                                         NIL
                                         NIL)))))
```

employee **verification**

```lisp
(defclass employee ()
  ((name :type (or string
                   (and symbol
                        (not keyword))
                   unsigned-byte))
   (half-time-p (or boolean
                    (member :true
                            :false))))
  (:metaclass json-serializable))
```

```lisp
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          'json)
```

✔ employee.name

employee **verification**

```lisp
(defclass employee ()
  ((name :type (or string
                   (and symbol
                        (not keyword))
                   unsigned-byte))
   (half-time-p (or boolean
                    (member :true
                            :false))))
  (:metaclass json-serializable))
```

```lisp
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          'json)
```

- employee.name
- employee.half-time-p

employee **verification**

```lisp
(defclass employee ()
  ((name :type (or string
                   (and symbol
                        (not keyword))
                   unsigned-byte))
   (half-time-p (or boolean
                    (member :true
                            :false))))
  (:metaclass json-serializable))
```

```lisp
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          'json)
```

- ✔ employee.name
- ✔ employee.half-time-p

employee verification

```lisp
(defclass employee ()
  ((name :type (or string
                   (and symbol
                        (not keyword))
                   unsigned-byte))
   (half-time-p (or boolean
                    (member :true
                            :false))))
  (:metaclass json-serializable))
```

```lisp
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          'json)
```

- ✔ `employee.name`
- ✔ `employee.half-time-p`

Conclusion

employee is JSON-compatible! 🎉

Going further

CLOS classes & `null-numeric-type-p`

CLOS classes

- Issue → find a representative instance
- Cannot use `make-instance` → possible side-effects
- Baker's solution
  - *hook into `defclass`/`defstruct` implementation*
  - – not portable
  - – maybe not trivial
- Our solution → the Meta Object Protocol
  - *register class prototypes → "fake" instances*
  - + portable
  - + easier to implement
  - + packageable

CLOS classes

- Issue → find a representative instance
- Cannot use `make-instance` → possible side-effects
- Baker's solution
  - *hook into `defclass`/`defstruct` implementation*
  - – not portable
  - – maybe not trivial
- Our solution → the Meta Object Protocol
  - *register class prototypes → "fake" instances*
  - + portable
  - + easier to implement
  - + packageable

CLOS classes

- Issue → find a representative instance
- Cannot use `make-instance` → possible side-effects
- Baker's solution
  - *hook into `defclass`/`defstruct` implementation*
  - – not portable
  - – maybe not trivial
- Our solution → the Meta Object Protocol
  - *register class prototypes → "fake" instances*
  - + portable
  - + easier to implement
  - + packageable

`null-numeric-type-p`

- Many representations of numerical data types
- Range representation available → `(integer (12) *)`
- Subtyping problem ⇒ interval combination canonicalization
- Exponential theoretical complexity
  - acceptable in practice

Conclusion

- Baker's decision procedure
- Implementation
  - Pre-processing
  - Primitive types, CLOS classes, `member` type specifiers
  - Numeric ranges
- Still a work in progress
- Intuitively more accurate
- More efficient

Conclusion

- Baker's decision procedure
- Implementation
  - Pre-processing
  - Primitive types, CLOS classes, `member` type specifiers
  - Numeric ranges
- Still a work in progress
- Intuitively more accurate
- ~~More efficient~~ *April fools* 😸

Thanks for listening! 😃

Any question?