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

The Common Lisp type system

- Types $\rightarrow$ sets, subtypes $\rightarrow$ subsets

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)))
```

The Common Lisp type system

- Types $\rightarrow$ sets, subtypes $\rightarrow$ subsets
- Types $\rightarrow$ *first class* values

The Common Lisp type system

- Types $\rightarrow$ sets, subtypes $\rightarrow$ subsets
- Types $\rightarrow$ *first class* values
- `(subtypep` $\langle A \rangle$ $\langle B \rangle$`)` $\equiv A \subseteq B$?
- Predicate function

The Common Lisp type system

- Types $\rightarrow$ sets, subtypes $\rightarrow$ subsets
- Types $\rightarrow$ *first class* values
- `(subtypep` $\langle A \rangle$ $\langle B \rangle$`)` $\equiv A \subseteq B$?
- Predicate function

λ Common Lisp

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

The Common Lisp type system

- Types → sets, subtypes → subsets
- Types → *first class values*
- (
- P

```
1
2
3
4        (unsigned-byte 10)))
```

- Type specifiers arbitrarily deep
- May take a while to retrun

Problem #1 — complex input

Arbitrarily complex input type specifiers

`subtypep` cannot always answer

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

`subtypep` cannot always answer

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

`subtypep` cannot always answer

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

`subtypep` cannot always answer

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

Problem #2 — undecidability

Subtypep cannot answer for some type specifiers

`subtypep` return values

$$
\texttt{(subtypep}\ \langle A\rangle\ \langle B\rangle\texttt{)} = \begin{cases} \texttt{(T T)} & \rightarrow A \subseteq B \\ \texttt{(NIL T)} & \rightarrow A \not\subseteq B \\ \texttt{(NIL NIL)} & \rightarrow \text{``undecidable''} \end{cases}
$$

- `(NIL NIL)` encodes undecidability

`subtypep` return values

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

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

`subtypep` **return values**

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

- `(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)` when it is possible to answer
- − paper difficult to read
- − not exhaustive
- − very few solutions about `satisfies`

- − no implementation available
- − exponential complexity (theoretical)
- ? efficiency

Baker's decision procedure

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

- − no implementation available
- − exponential complexity (theoretical)
- ? efficiency

Baker's decision procedure

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

- − no implementation available
- − exponential complexity (theoretical)
- ? efficiency

Content

1. **Application using `subtypep`**
2. **Baker's decision procedure**
   2.1 *Pre-processing*
   2.2 *Types as bit-vectors*
   2.3 *Type specifier $\rightarrow$ bit-vector expression*
3. **Going further**

The problem

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

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"
}
```

The problem

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"
}
```

λ Common Lisp

```lisp
(deftype json ()
  '(or number
       string
       (and symbol
            (not keyword))
       list
       hash-table))
```

Our `employee` class

- 2 slots ⇒ 2 calls to `subtypep`
- Trigger error if one fails

λ Common Lisp

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

Baker's decision procedure

Application of our implementation to check

`employee.name` $\subseteq$ `json`

Pre-processing steps

λ Common Lisp

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

Pre-processing steps

λ Common Lisp

```lisp
(subtypep '(or string
               (and symbol
                    (not keyword))
               unsigned-byte)
          '(or number
               string
               (and symbol
                    (not keyword))
               list
               hash-table))
```

- Alias expansion

Pre-processing steps

λ Common Lisp

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

- Alias expansion
- $P \subseteq Q \Rightarrow P \cap \neg Q = \emptyset$

Bit-vector type representation

Bit-vector type representation

Types represented as bit-vectors $\mathcal{B}_P$

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

Bit-vector type representation

Types represented as bit-vectors $\mathcal{B}_P$

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

Properties (bitwise)

$$\mathcal{B}_{P \cup Q} = \mathcal{B}_P \vee \mathcal{B}_Q$$

$$\mathcal{B}_{P \cap Q} = \mathcal{B}_P \wedge \mathcal{B}_Q$$

$$\mathcal{B}_{\overline{P}} = \neg \mathcal{B}_P$$

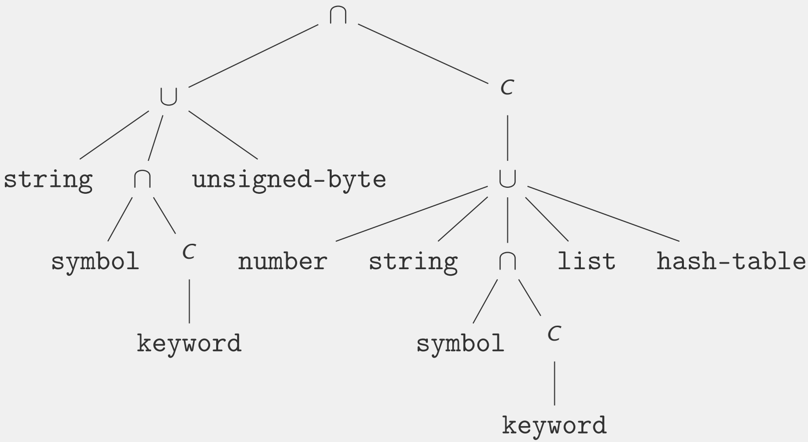
Back to our problem

λ Common Lisp

```lisp
(subtypep '(and (or string
                    (and symbol
                         (not keyword))
                    unsigned-byte)
                (not (or number
                         string
                         (and symbol
                              (not keyword))
                         list
                         hash-table)))
          nil)
```

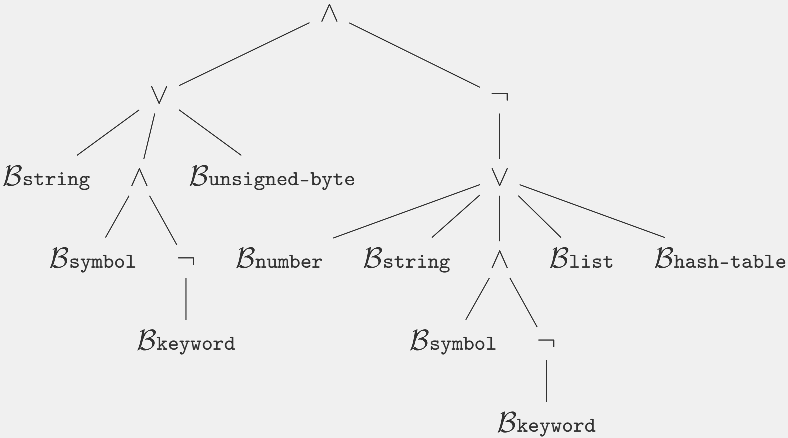
Back to our problem

Bit-vector expression reduction



Back to our problem

Bit-vector expression reduction



`employee` verification

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

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

✔ `employee.name`

`employee` verification

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

```
(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 boolean))
  (: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 boolean))
  (: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! 🎉

CLOS classes & `member` type specifiers

Choosing representative elements right

CLOS classes

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

CLOS classes

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

CLOS classes

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

`member` type specifiers

- Explicitly provide type's elements
- $(\texttt{member}\ \langle A\rangle\ \langle B\rangle\ \langle C\rangle) \equiv \{A, B, C\}$
- "Anonymous" types
- Bit-vector $\mathcal{B}_{(\texttt{member}\ \langle A\rangle\ \langle B\rangle\ \langle C\rangle)}$
  1. add A, B, C as representatives
  2. $\mathcal{B}_{(\texttt{member}\ \langle A\rangle\ \langle B\rangle\ \langle C\rangle)} = \mathcal{B}_{\{A\}} \vee \mathcal{B}_{\{B\}} \vee \mathcal{B}_{\{C\}}$

Conclusion

- `subtypep` unreliability
- Baker's decision procedure
  - no implementation given
  - many details missing
  - seems elegant and powerful
- Our implementation
  - incomplete & experimental
  - motivating accuracy & performance measures
- Future work
  - implement missing type specifiers (`array` & `complex`)
  - find solutions for `cons` & `satisfies`
  - open source the implementation!

Thanks for listening! 😃

Any question?