Representing and Computing with Types in Dynamically Typed Languages

Extending Dynamic Language Expressivity to Accommodate Rationally Typed Sequences

Jim Edward Newton

Overview of thesis defense for the title: Doctorat de l'Université Sorbonne Université

Directory: Pr. Thierry Géraud, EPITA/LRDE

Advisor: Dr. Didier Verna, EPITA/LRDE

December 11, 2018

Introduction and Context

What is Common Lisp?

- Multi-paradigm: programming language
- ... allow the programmer to express himself.
- Functional, procedural, object-oriented.
- Meta-programming: Meta-object protocol, macros.
- Dynamic approach to typing and reflection

Regular Sequences of Heterogeneous Types

- We can declare types of certain data.
  - `(declare (type integer X) (type list Y))` ; YES

Regular Sequences of Heterogeneous Types

- We can declare types of certain data.
  - `(declare (type integer X) (type list Y))` ; YES
- We can use arbitrary, heterogeneous sequences.
  - `(:a  1 1.0  :b "a" "an" "the" :c 2 22 222 :d 2.3 )` ; YES

Regular Sequences of Heterogeneous Types

- We can declare types of certain data.
  - `(declare (type integer X) (type list Y))` ; YES
- We can use arbitrary, heterogeneous sequences.
  - `(:a  1 1.0  :b "a" "an" "the" :c 2 22 222 :d 2.3 )` ; YES
- However, it is difficult to combine.

Regular Sequences of Heterogeneous Types

- We can declare types of certain data.
  - `(declare (type integer X) (type list Y))` ; YES
- We can use arbitrary, heterogeneous sequences.
  - `(:a  1 1.0  :b "a" "an" "the" :c 2 22 222 :d 2.3 )` ; YES
- However, it is difficult to combine.
  - `(declare (type list[integer] X))` ; NO!

Regular Sequences of Heterogeneous Types

- We can declare types of certain data.
  - `(declare (type integer X) (type list Y))` ; YES
- We can use arbitrary, heterogeneous sequences.
  - `(:a 1 1.0 :b "a" "an" "the" :c 2 22 222 :d 2.3 )` ; YES
- However, it is difficult to combine.
  - `(declare (type list[integer] X))` ; NO!
  - `(declare (type` *regular-pattern* `X))` ; NO!

Regular Sequences of Heterogeneous Types

- We can declare types of certain data.
  - `(declare (type integer X) (type list Y))` ; YES
- We can use arbitrary, heterogeneous sequences.
  - `(:a  1 1.0  :b "a" "an" "the" :c 2 22 222 :d 2.3 )` ; YES
- However, it is difficult to combine.
  - `(declare (type list[integer] X))` ; NO!
  - `(declare (type` *regular-pattern* `X))` ; NO!
- We propose to extend the type system of Common Lisp.

Regular Sequences of Heterogeneous Types

- We can declare types of certain data.
  - `(declare (type integer X) (type list Y))` ; YES
- We can use arbitrary, heterogeneous sequences.
  - `(:a  1 1.0  :b "a" "an" "the" :c 2 22 222 :d 2.3 )` ; YES
- However, it is difficult to combine.
  - `(declare (type list[integer] X))` ; NO!
  - `(declare (type` *regular-pattern* `X))` ; NO!
- We propose to extend the type system of Common Lisp.
- We introduce RTE, regular type expressions, specifying heterogeneous but regular sequences.

Goal: Implement RTEs in Common Lisp

Vaguely: We want to efficiently detect whether a sequence of values matches a regular pattern of types.

Precisely: Given a pattern, at compile-time, generate code, such that given a sequence of values at run-time, we can determine whether the sequence matches the pattern.

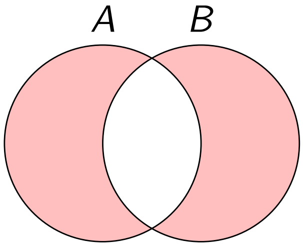
Implementing RTE presents several challenges

1. The representation problem:
   Representing rational type expressions in Common Lisp.
2. The decomposition problem:
   Calculating the Maximal Disjoint Type Decomposition (MDTD).
3. The serialization problem:
   Generating code without redundant type checks.

Overview

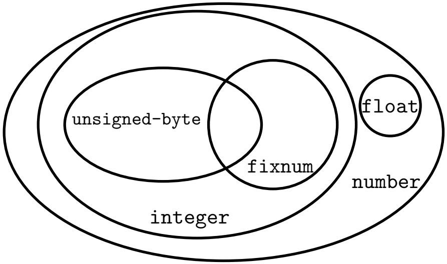
Types, Sequences, and Typed Sequences in Common Lisp

Quick intro to the Common Lisp Type System

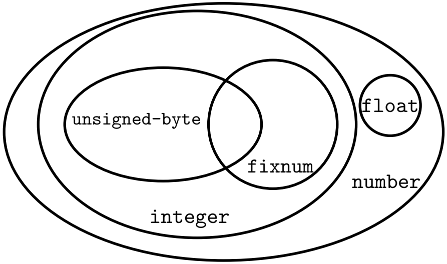


Type operations are set operations: membership, intersection, union, complement, empty-set.

Quick intro to the Common Lisp Type System



Quick intro to the Common Lisp Type System



```
(typep -1 '(or float (and integer (not unsigned-byte))))
```
→ true
```
(subtypep '(and integer fixnum) '(not number))
```
→ false
```
(subtypep '(and float fixnum) nil)
```
→ true

We'd like to recognize sequences with regular patterns.

(1 2.3 9.3 3 1.5 6.5 4.8 5 2 2.3)

We'd like to recognize sequences with regular patterns.
(1   2.3   9.3   3   1.5   6.5   4.8   5   2   2.3)

- We generalize string-based regular expressions to arbitrary sequences.
- To match a string like: `"iFFiFFFiiF"`,
- ... we use a RE such as: $(i \cdot F^*)^+$,
- ... which has surface syntax: `"(iF*)+"`.

We'd like to recognize sequences with regular patterns.
(1   2.3   9.3   3   1.5   6.5   4.8   5   2   2.3)

- We generalize string-based regular expressions to arbitrary sequences.
- To match a string like: `"iFFiFFFiiF"`,
- ... we use a RE such as: $(i \cdot F^*)^+$,
- ... which has surface syntax: `"(iF*)+"`.

We propose Rational Type Expressions (RTEs)

- Rational type expression: $(\ \textit{integer} \cdot \textit{float}\ ^*)^+$
- We need a surface syntax.

We think this:

$$\left(symbol \cdot number^{?} \cdot (ratio^{*} \vee float^{+})\right) \;\wedge\; \overline{t \cdot number \cdot number}$$

And we write this:

```
(:and (:cat symbol
            (:? number)
            (:or (:* ratio)
                 (:+ float)))
      (:not (:cat t number number)))
```

Support for `:and`, `:not`, `:?`, and `:+` is sometimes referred to as *extended* rational expressions. We don't distinguish *extended* and *ordinary* RE.

Using Surface Syntax

With the type *definition* `(rte ...)` we can use rational type expressions just like any other type in the language.

```
(defun set-attributes (object attr)
  (declare (type (rte (:* (:cat keyword number))) ; <-- RTE
                 attr))
  (setf (attributes object) attr))

(deftype plist (type)
  `(rte (:* (:cat keyword ,type)))) ; <-- RTE

(defclass polygon ()
  ((color :type rgb)
   (points :type (rte (:* (:cat fixnum real)))))) ; <-- RTE
```

Efficient Pattern Matching Based on Types

Does:

`(a 1 1.0 b "a" "an" "the" c 2 22 222 d 2.3)`

`follow the pattern:` $(\mathit{symbol} \cdot (\mathit{number}^{+} \ \vee \mathit{string}^{+}))^{+}$ `?`

I.e., is the sequence an element of the specified type?

Does:

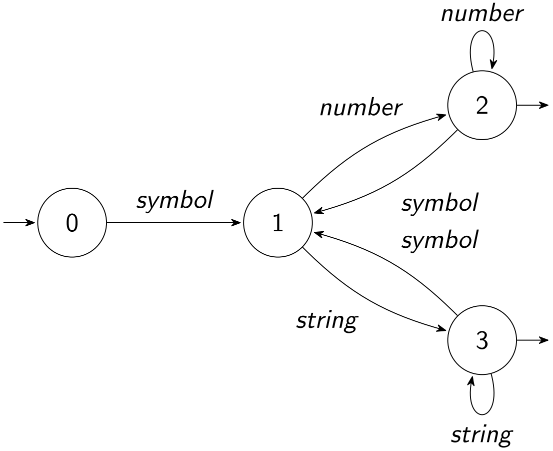
`(a 1 1.0 b "a" "an" "the" c 2 22 222 d 2.3)`

follow the pattern: $(\mathit{symbol} \cdot (\mathit{number}^{+} \lor \mathit{string}^{+}))^{+}$?

I.e., is the sequence an element of the specified type?

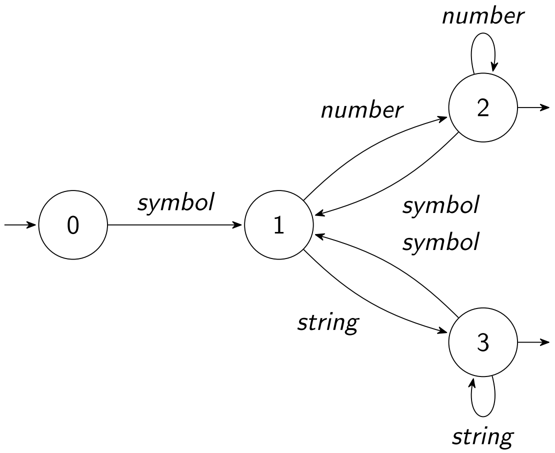
We construct a *deterministic* finite automaton (DFA).

We want to support `:not` and `:and` in our DSL.



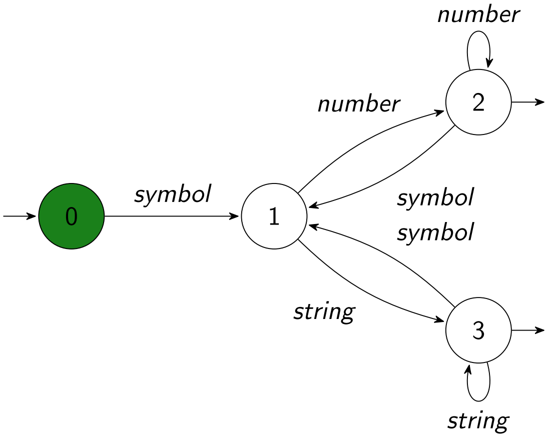
```
(a 1 1.0 b "a" "an" "the" c 2 22 222 d 2.3)
```

How does a DFA work as a type predicate?



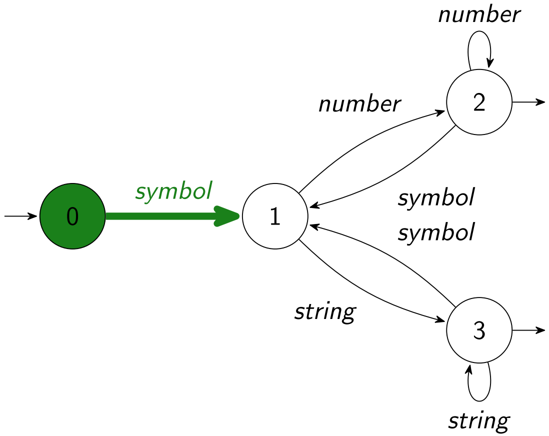
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



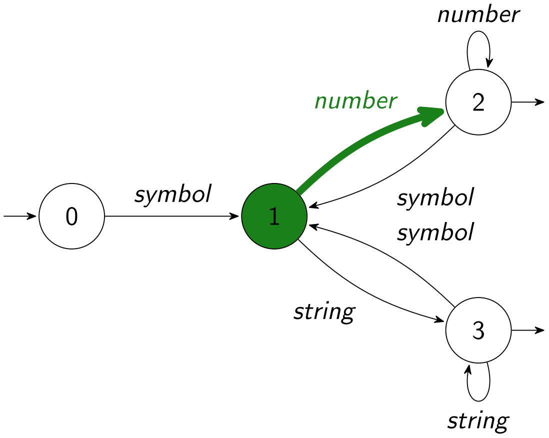
How does a DFA work as a type predicate?

```
(ⓐ 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



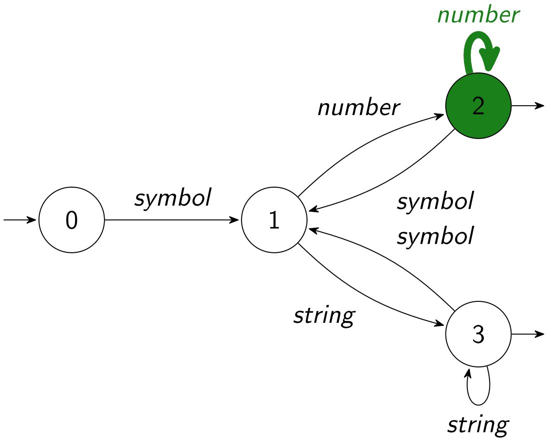
How does a DFA work as a type predicate?

```
(a ① 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



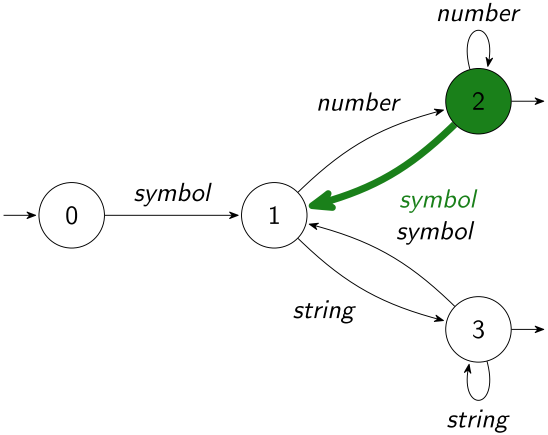
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



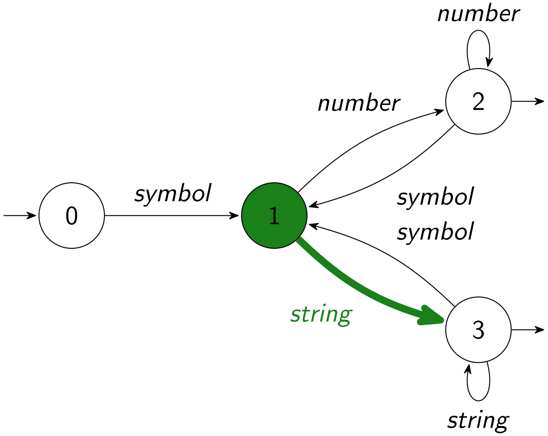
How does a DFA work as a type predicate?

```
(a 1 1.0 ⓑ "a"
"an" "the" c 2
22 222 d 2.3)
```



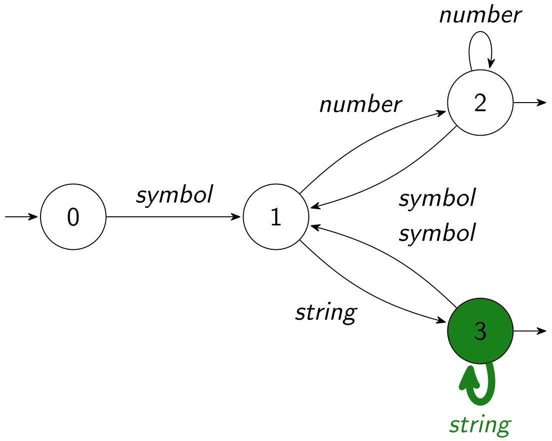
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



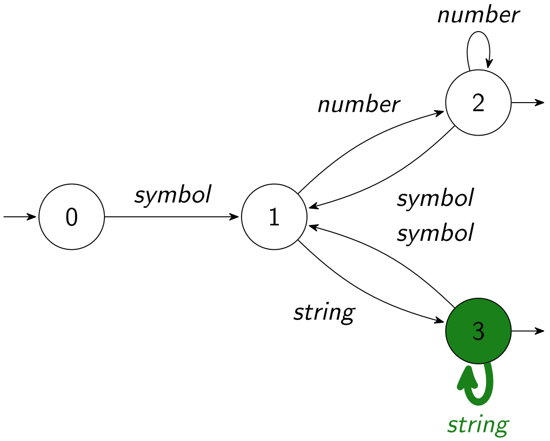
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an"
     "the" c 2
22 222 d 2.3)
```


$$\begin{tikzcd}
\arrow[r] & 0 \arrow[r, "symbol"] & 1 \arrow[r, bend left, "number"] \arrow[rd, bend right, "string"'] & 2 \arrow[l, bend left, "symbol"] \arrow[loop above, "number"] \arrow[r] & {} \\
& & & 3 \arrow[lu, bend right, "symbol"'] \arrow[loop below, "string"] \arrow[r] & {}
\end{tikzcd}$$

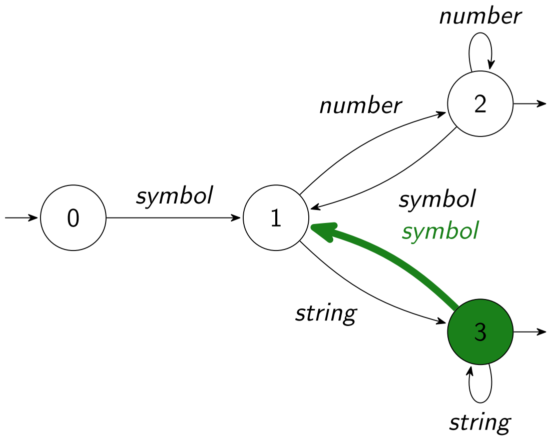
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
        "the"
"an"          c 2
22 222 d 2.3)
```



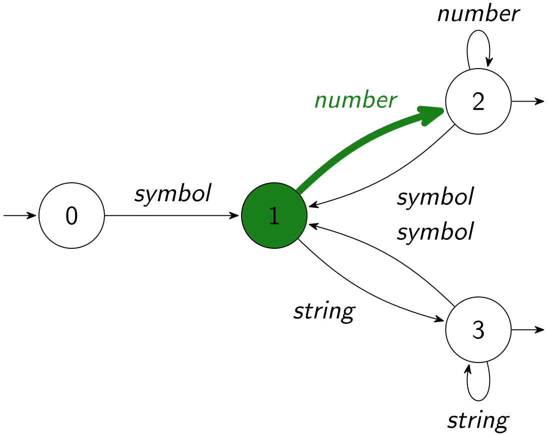
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an" "the" ⓒ 2
22 222 d 2.3)
```



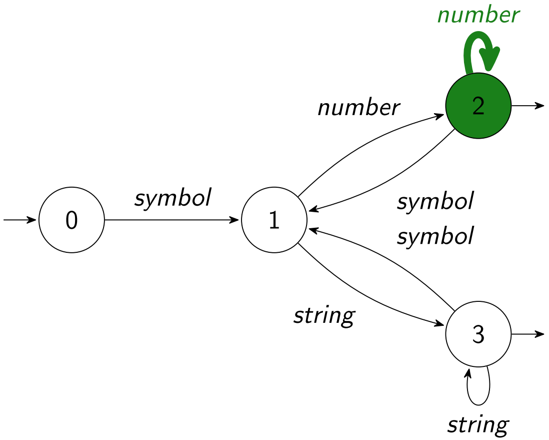
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an" "the" c ②
22 222 d 2.3)
```



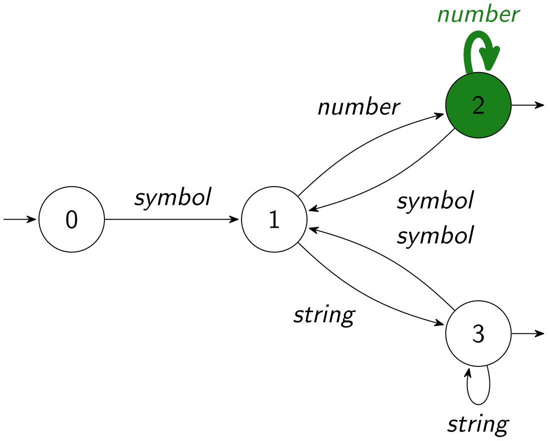
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



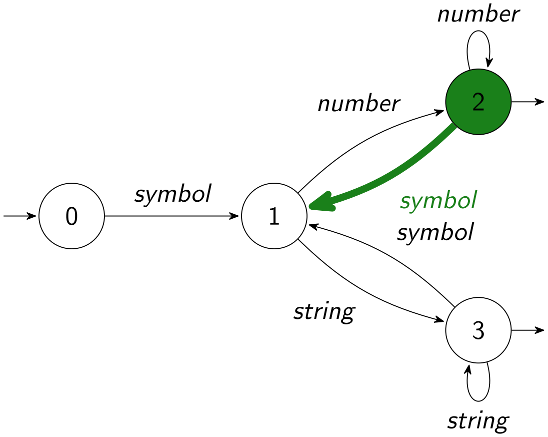
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



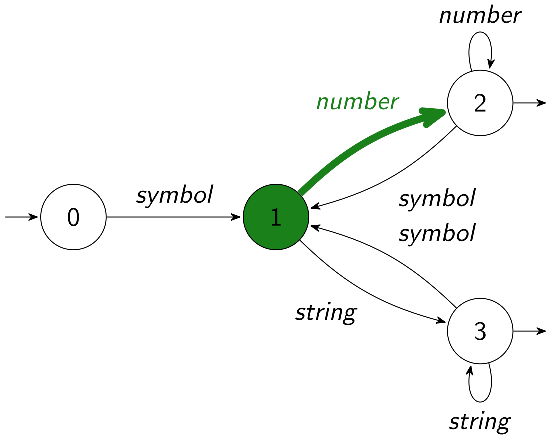
How does a DFA work as a type predicate?

```
(a 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



How does a DFA work as a type predicate?

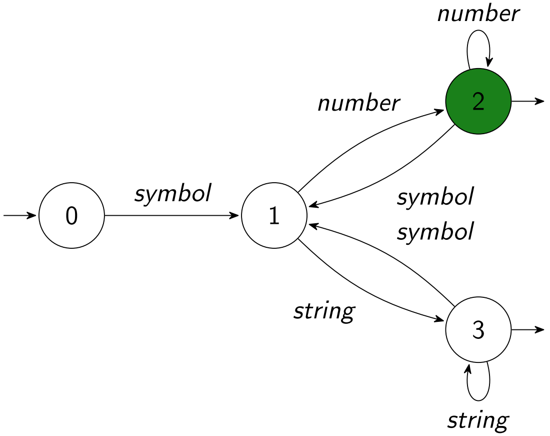
```
(a 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



How does a DFA work as a type predicate?

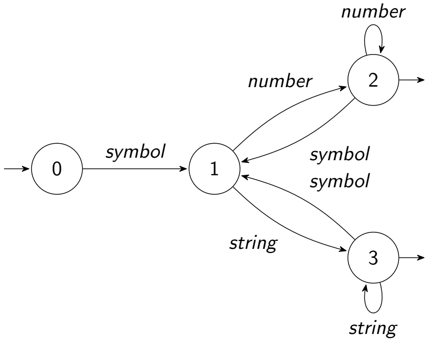
Yes, it's a match!

```
(a 1 1.0 b "a"
"an" "the" c 2
22 222 d 2.3)
```



Code generated from $(symbol \cdot (number^+ \vee string^+))^+$

```
(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)))
```



$$\begin{tikzcd}
 & & 2 \arrow[loop above, "number"] \arrow[dl, bend left, "symbol"] \arrow[r] & {} \\
\arrow[r] & 0 \arrow[r, "symbol"] & 1 \arrow[u, bend left, "number"] \arrow[d, bend right, "string"'] & \\
 & & 3 \arrow[loop below, "string"] \arrow[ul, bend right, "symbol"'] \arrow[r] & {}
\end{tikzcd}$$

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

Lambda-lists characterized by RTEs

A lambda-list in Common Lisp has a fixed part

```
(defun foo (a b)
   ...)

(lambda (a b)
   ...)
```

Lambda-lists characterized by RTEs

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

```
(defun foo (a b &optional c)
  ...)

(lambda (a b &optional c)
  ...)
```

Lambda-lists characterized by RTEs

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

```
(defun foo (a b &optional c &key x y)
  ...)

(lambda (a b &optional c &key x y)
  ...)
```

Lambda-lists characterized by RTEs

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

```
(defun foo (a b &optional c &key x y)
  (declare (type integer a x)
           (type string b c y))
  ...)

(lambda (a b &optional c &key x y)
  (declare (type integer a x)
           (type string b c y))
  ...)
```

Lambda-lists characterized by RTEs

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

```
(defun foo (a b &optional c &key x y)
  (declare (type integer a x)
           (type string b c y))
  ...)

(lambda (a b &optional c &key x y)
  (declare (type integer a x)
           (type string b c y))
  ...)
```

The set of valid argument lists for a function may be characterized by an RTE.

Calling an anonymous function.

```
(apply (lambda (a b &key (x t) (y "") z)
         (declare (type fixnum a b z)
                  (type symbol x)
                  (type string y))
         ...body...)

       DATA)
```

Calling an anonymous function.

```
(apply (lambda (a b &key (x t) (y "") z)
         (declare (type fixnum a b z)
                  (type symbol x)
                  (type string y))
         ...body...)

       DATA)
```

For example:

```
DATA = (2 3 :y "a" :x 'b) ; YES
```

Calling an anonymous function.

```
(apply (lambda (a b &key (x t) (y "") z)
         (declare (type fixnum a b z)
                  (type symbol x)
                  (type string y))
         ...body...)

       DATA)
```

For example:

```
DATA = (2 3 :y "a" :x 'b) ; YES
DATA = (2 3 :y "a" :x 'b :x 42 :y "hello" :y nil) ; YES
```

Calling an anonymous function.

```
(apply (lambda (a b &key (x t) (y "") z)
         (declare (type fixnum a b z)
                  (type symbol x)
                  (type string y))
         ...body...)

       DATA)
```

For example:

```
DATA = (2 3 :y "a" :x 'b) ; YES
DATA = (2 3 :y "a" :x 'b :x 42 :y "hello" :y nil) ; YES
DATA = (2 3 :y "a" :x 42 :x 'b) ; NO
```

An invalid argument list will signal an error at run-time.

QUESTION: Can we select an appropriate lambda-list matching `DATA`, avoiding a run-time error?
We propose `destructuring-case`.

```
(destructuring-case DATA

  ;; Case-1
  ((a b &optional (c ""))
   (declare (type integer a)
            (type string b c))
   ...body...)

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

QUESTION: Can we select an appropriate lambda-list matching `DATA`, avoiding a run-time error?
We propose `destructuring-case`.

```
(destructuring-case DATA

  ;; Case-1
  ((a b &optional (c ""))
   (declare (type integer a)
            (type string b c))
   ...body...)

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

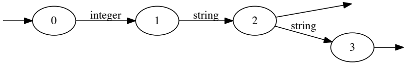
- *integer* · *string* · *string*$^?$

QUESTION: Can we select an appropriate lambda-list matching `DATA`, avoiding a run-time error?

We propose `destructuring-case`.

```
(destructuring-case DATA

  ;; Case-1
  ((a b &optional (c ""))
   (declare (type integer a)
            (type string b c))
   ...body...)

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

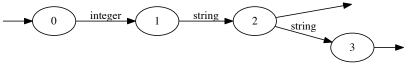
- $integer \cdot string \cdot string^?$



QUESTION: Can we select an appropriate lambda-list matching `DATA`, avoiding a run-time error?
We propose `destructuring-case`.

```
(destructuring-case DATA

  ;; Case-1
  ((a b &optional (c ""))
   (declare (type integer a)
            (type string b c))
   ...body...)

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

- *integer · string · string*$^?$

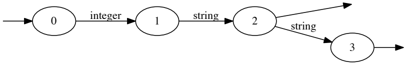


- What is the rational type expression?

QUESTION: Can we select an appropriate lambda-list matching `DATA`, avoiding a run-time error?
We propose `destructuring-case`.

```
(destructuring-case DATA

  ;; Case-1
  ((a b &optional (c ""))
   (declare (type integer a)
            (type string b c))
   ...body...)

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

- *integer · string · string*$^?$

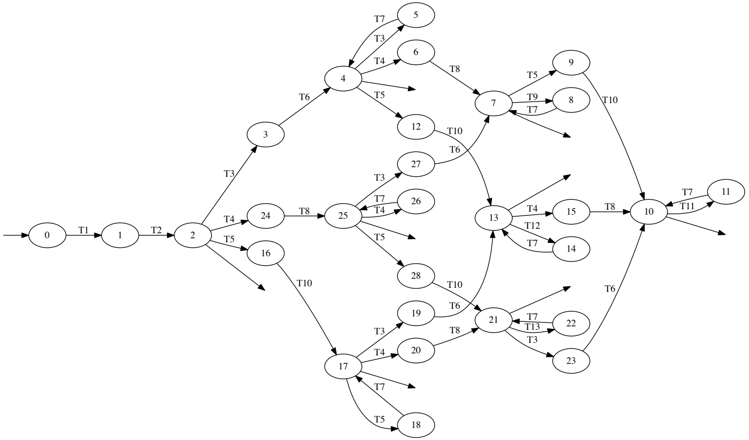


- What is the rational type expression?
- What is the DFA?

RTE auto-generated from destructuring lambda-list

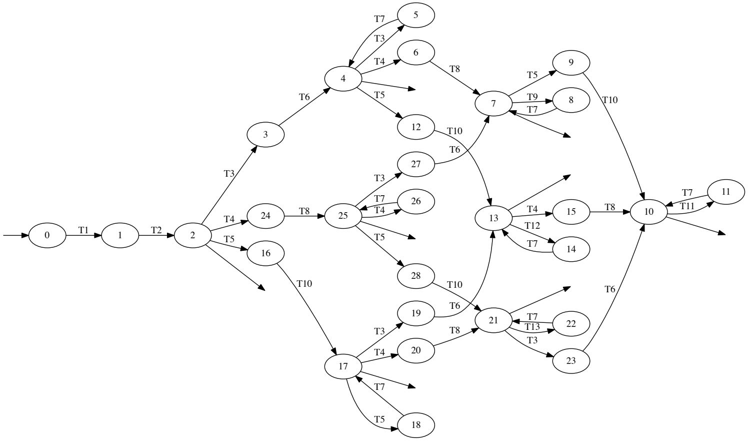
```
(:cat (:cat fixnum
            (:and list (rte (:cat fixnum fixnum))))
      (:and (:* (:cat (:or (eql :x) (eql :y) (eql :z))
                      t))
            (:cat (:* (:cat (not (eql :x))
                            t))
                  (:? (:cat (eql :x)
                            symbol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

DFA corresponding to auto-generated RTE



T_1 = fixnum	T_2 = (and list (rte (:cat fixnum fixnum)))	T_3 = (eql :x)	T_4 = (eql :y)
T_5 = (eql :z)	T_6 = symbol	T_7 = t	T_8 = string
T_9 = (member :x :y)	T_{10} = list	T_{11} = (member :x :y :z)	
T_{12} = (member :x :z)	T_{13} = (member :y :z)		

DFA corresponding to auto-generated RTE

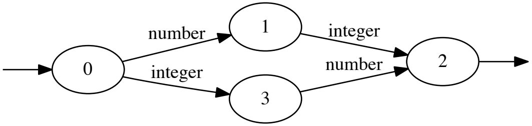


Multiple transitions from states give rise to serialization problem.

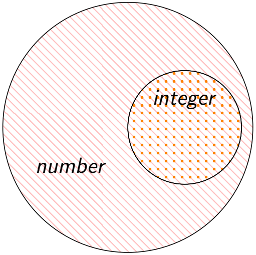
Rational Type Expressions (RTEs) with overlapping types

$$(\mathit{number} \cdot \mathit{integer}) \vee (\mathit{integer} \cdot \mathit{number})$$

We have non-deterministic (NFA).



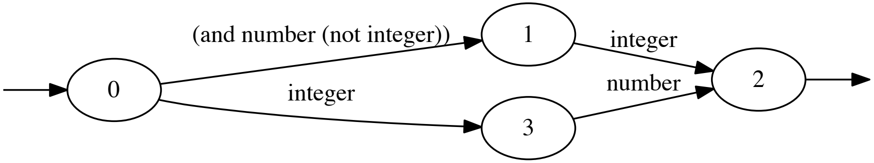
$\mathit{integer} \subset \mathit{number}$



Rational Type Expressions (RTEs) with overlapping types

$$(number \cdot integer) \vee (integer \cdot number)$$

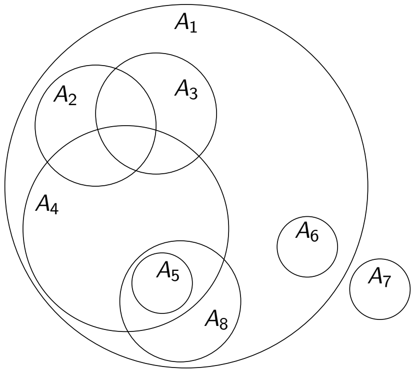
We want deterministic (DFA).



Maximal Disjoint Type Decomposition

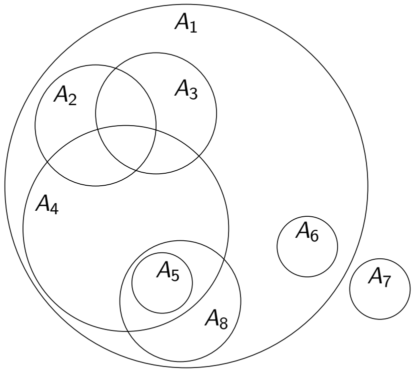
MDTD: decompose a set of types into disjoint types

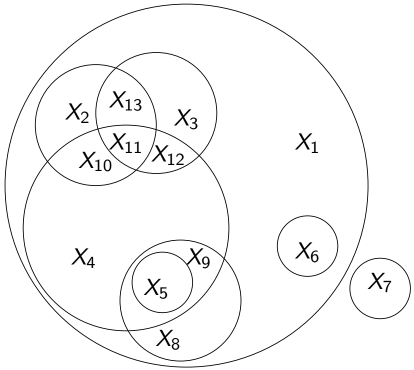
- Given A_i as possibly overlapping regions,



MDTD: decompose a set of types into disjoint types

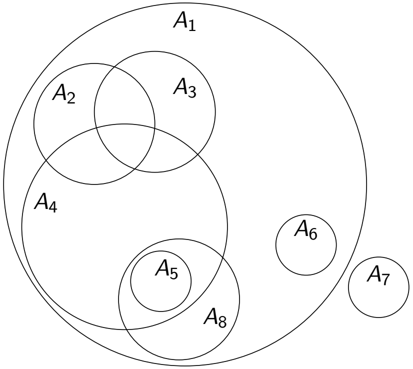
- Given A_i as possibly overlapping regions,
- Calculate X_i as disjoint regions.

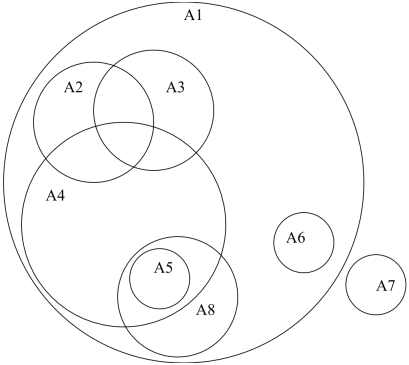




MDTD problem: Baseline algorithm

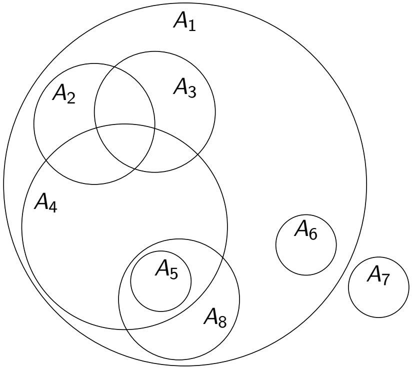
Are there any disjoint sets?

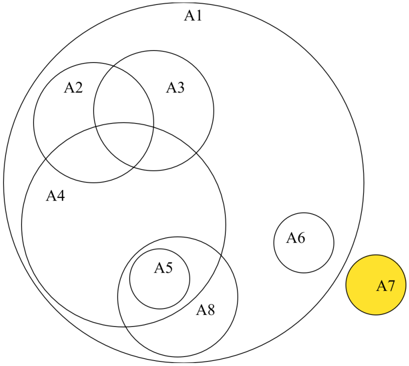




MDTD problem: Baseline algorithm

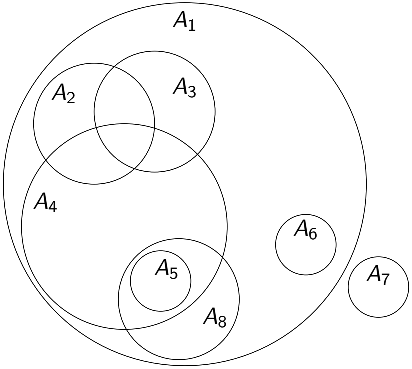
Yes, A_7 intersects no other set.

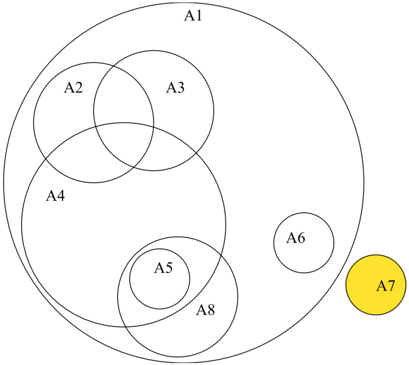




MDTD problem: Baseline algorithm

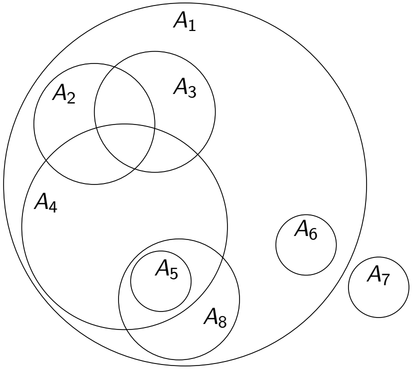
So collect it into D : $D = \{A_7\}$

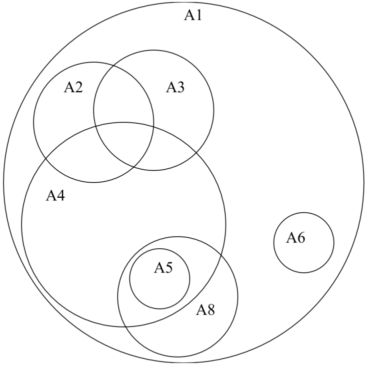




MDTD problem: Baseline algorithm

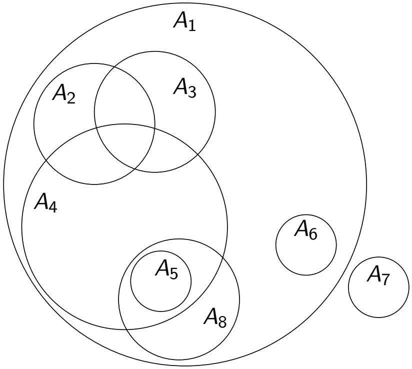
Select any intersecting pair of sets.

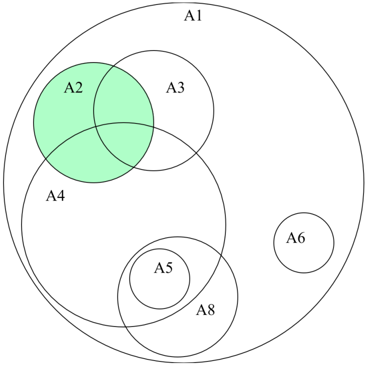




MDTD problem: Baseline algorithm

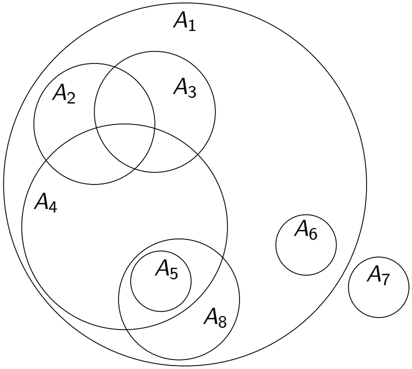
E.g., A_2. Does A_2 intersect anything?

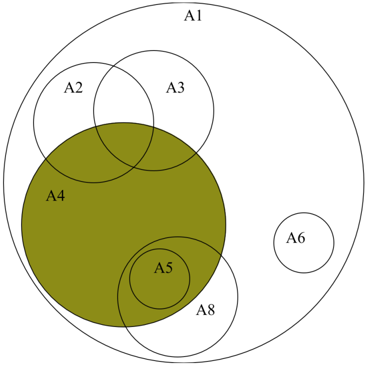




MDTD problem: Baseline algorithm

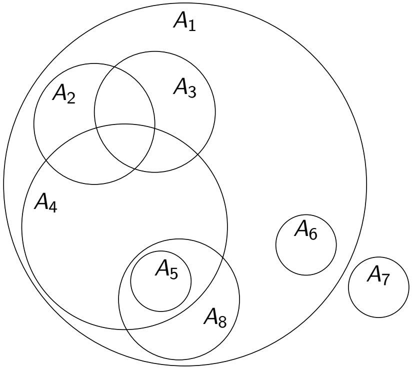
Yes. A_2 intersects A_4.

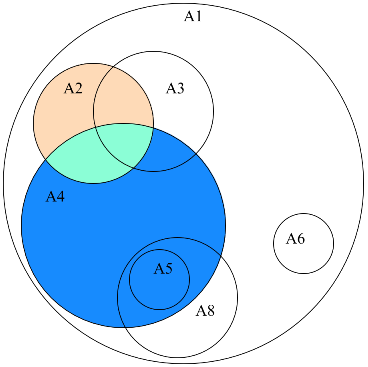




MDTD problem: Baseline algorithm

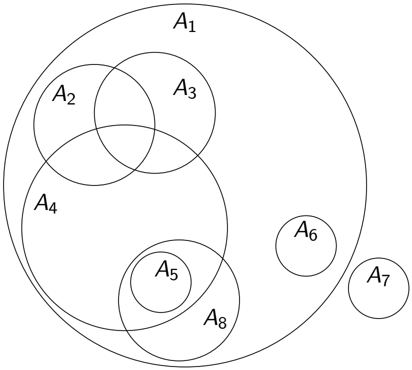
So calculate the standard partition of A_2 and A_4

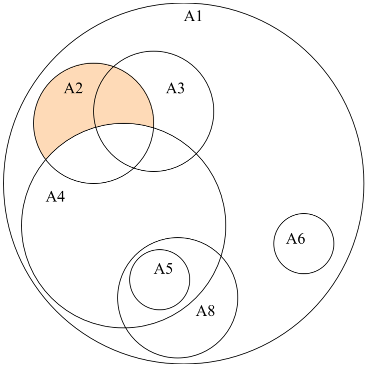




MDTD problem: Baseline algorithm

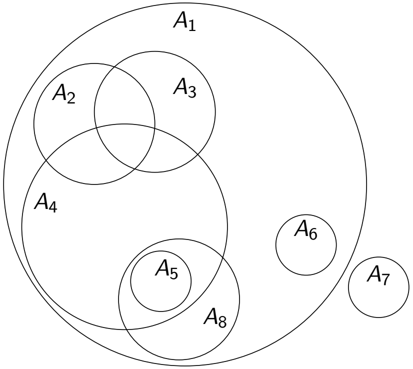
The standard partition is $\{A_2 \cap \overline{A_4}, \ldots\}$

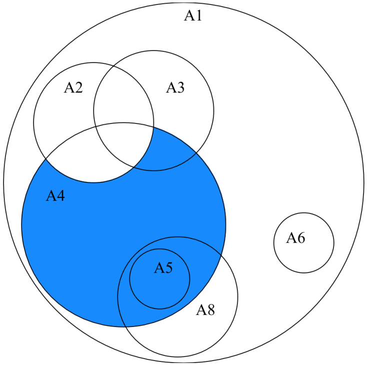




MDTD problem: Baseline algorithm

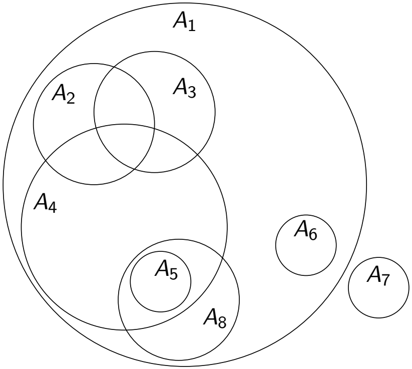
The standard partition is $\{A_2 \cap \overline{A_4}, \quad A_4 \cap \overline{A_2}, \ldots\}$

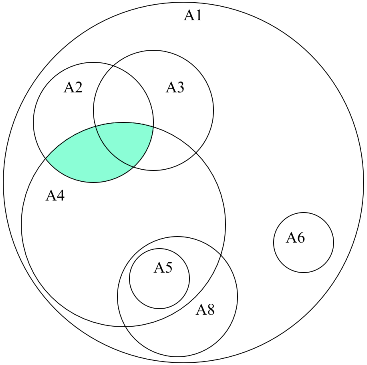




MDTD problem: Baseline algorithm

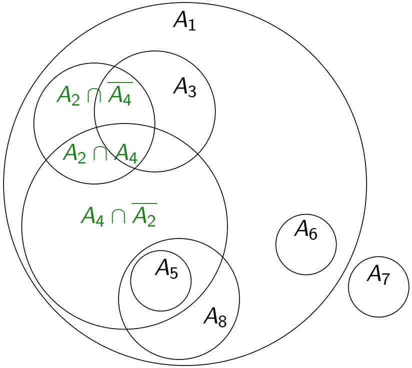
The standard partition is $\{A_2 \cap \overline{A_4}, \quad A_4 \cap \overline{A_2}, \quad A_2 \cap A_4\}$

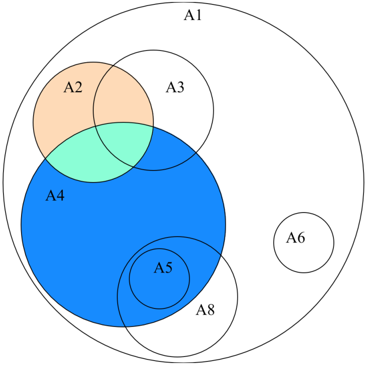




MDTD problem: Baseline algorithm

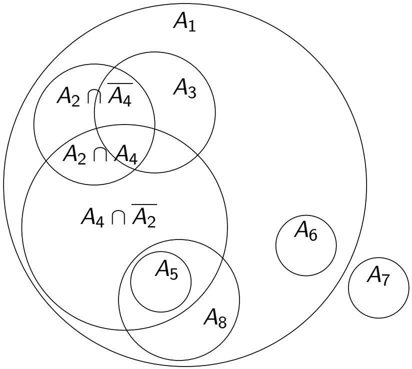
So remove $\{A_2,\ A_4\}$ and add $\{A_2 \cap \overline{A_4},\quad A_4 \cap \overline{A_2},\quad A_2 \cap A_4\}$.

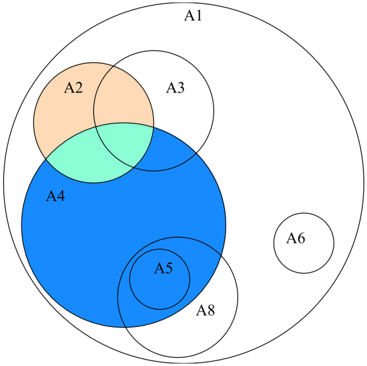




MDTD problem: Baseline algorithm

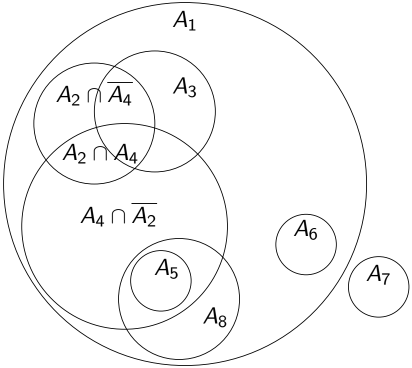
Now, restart. Anything disjoint from everything else? No.

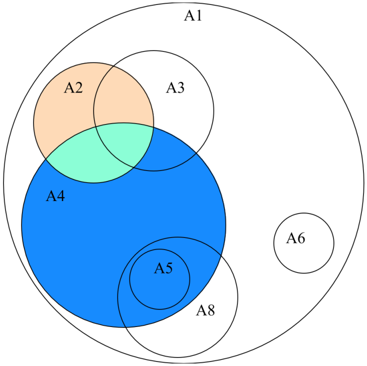




MDTD problem: Baseline algorithm

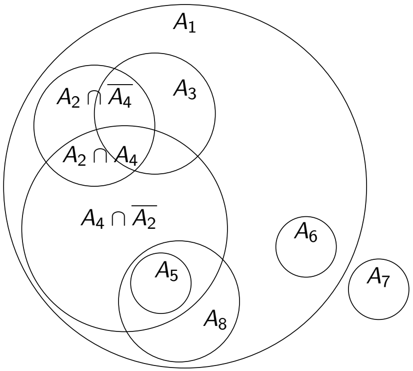
So select any intersecting pair.

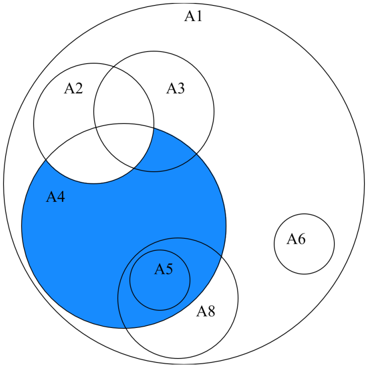




MDTD problem: Baseline algorithm

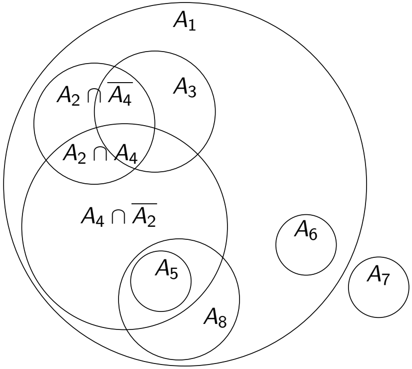
E.g., $A_4 \cap \overline{A_2}$. Does it intersect anything?

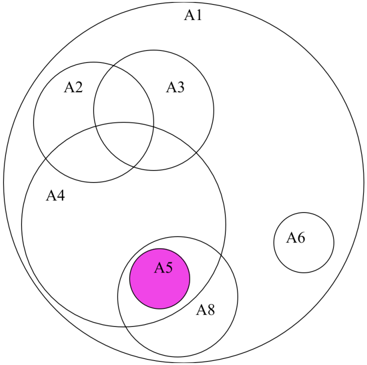




MDTD problem: Baseline algorithm

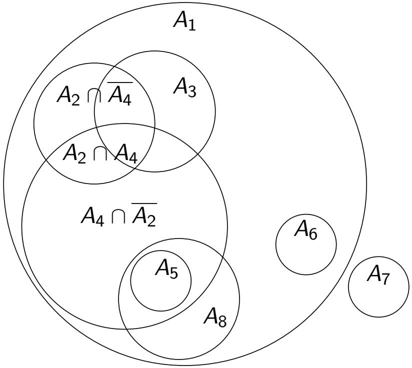
Yes, it intersects A_5.

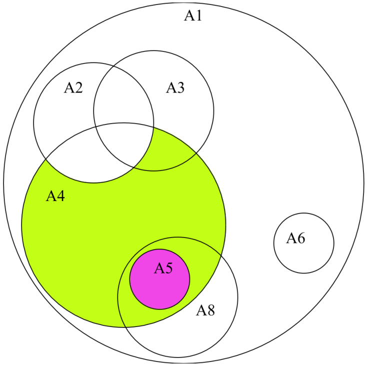




MDTD problem: Baseline algorithm

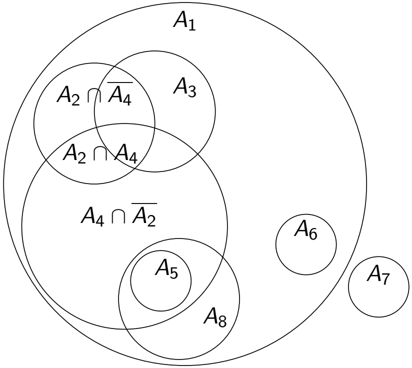
So calculate the standard partition of A_5 and $A_4 \cap \overline{A_2}$.

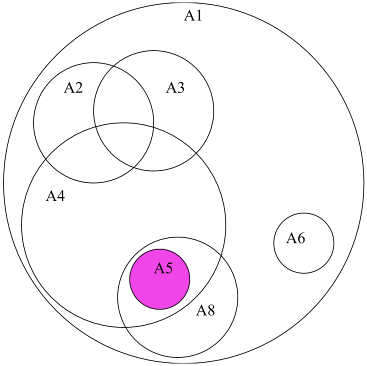




MDTD problem: Baseline algorithm

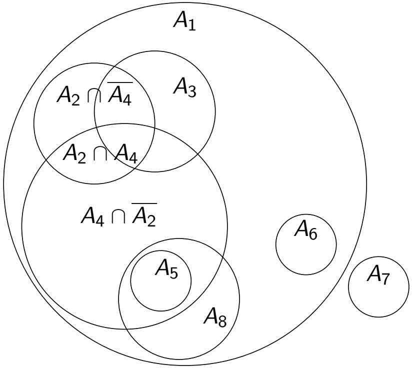
The standard partition is $\{A_5, ...\}$.

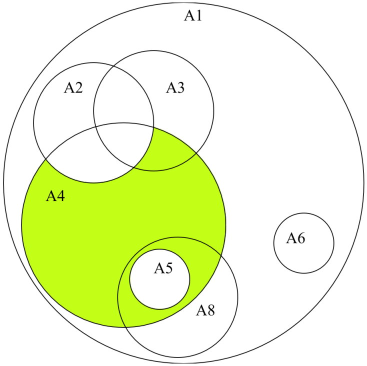




MDTD problem: Baseline algorithm

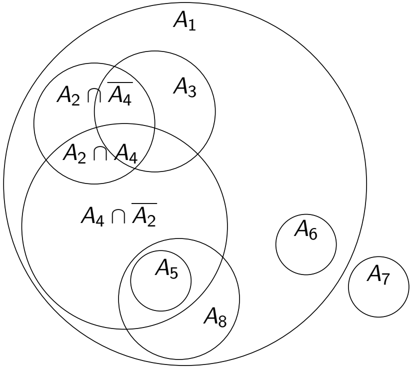
The standard partition is $\{..., A_4 \cap \overline{A_2} \cap \overline{A_5}\}$.

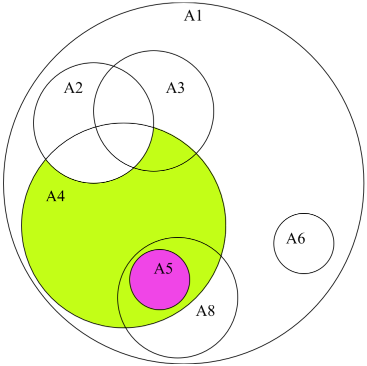




MDTD problem: Baseline algorithm

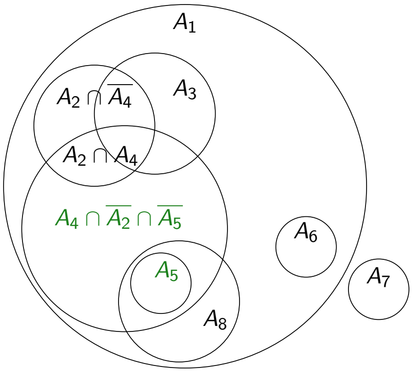
The standard partition is $\{A_5, \quad A_4 \cap \overline{A_2} \cap \overline{A_5}\}$.

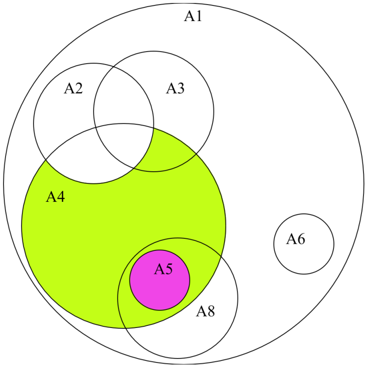




MDTD problem: Baseline algorithm

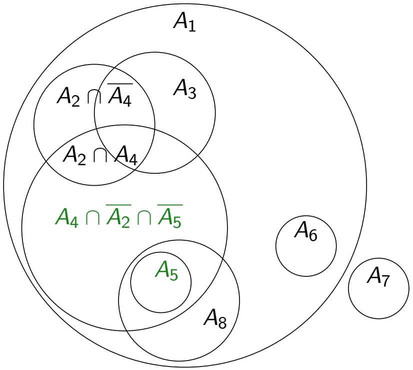
So remove $\{A_5,\ A_4 \cap \overline{A_2}\}$ and add $\{A_5,\ A_4 \cap \overline{A_2} \cap \overline{A_5}\}$.


A_1
$A_2 \cap \overline{A_4}$
A_3
$A_2 \cap A_4$
$A_4 \cap \overline{A_2} \cap \overline{A_5}$
A_6
A_5
A_7
A_8

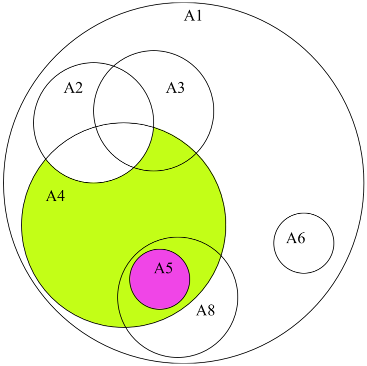


MDTD problem: Baseline algorithm

So remove $\{A_5,\ A_4 \cap \overline{A_2}\}$ and add $\{A_5,\ A_4 \cap \overline{A_2} \cap \overline{A_5}\}$.
A_5 is in both sets. We can optimize, because $A_5 \subset A_4 \cap \overline{A_2}$.

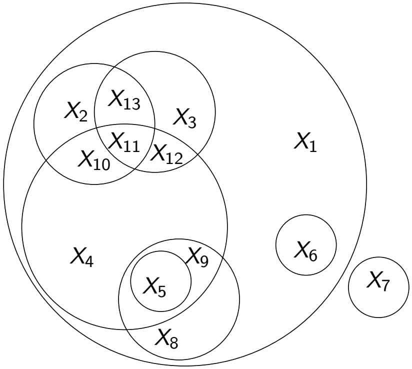


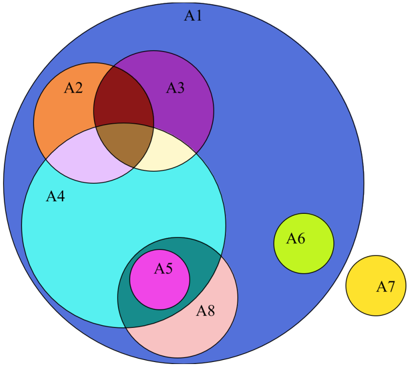
A_1
$A_2 \cap \overline{A_4}$
A_3
$A_2 \cap A_4$
$A_4 \cap \overline{A_2} \cap \overline{A_5}$
A_6
A_5
A_7
A_8



MDTD problem: Baseline algorithm

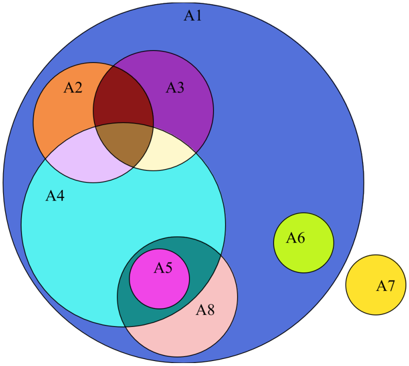
Continue the procedure until collecting all the pairwise disjoint sets.

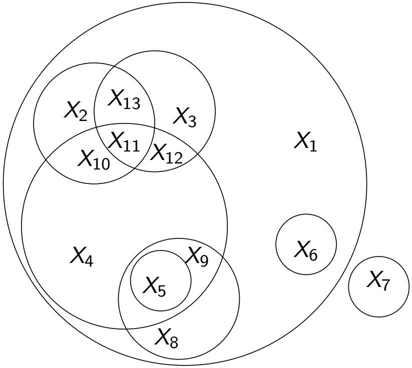


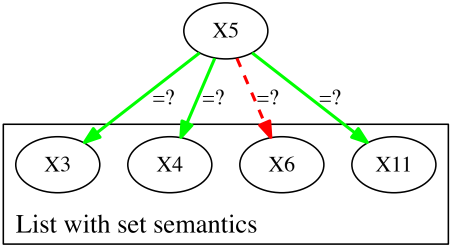


MDTD problem: Baseline algorithm

Calculating all the colored regions as subsets of original overlapping sets.







- Insertion into list with set semantics has linear complexity.
- Uniquify list has quadratic complexity.
- Type equivalence check is $X_i \subset X_j \wedge X_j \subset X_i$?
- And prevents us from using a hash table to implement sets.
- This equivalence function is SLOW!

MDTD result: type specifiers are explosive in size

```
X2: (and (and
            (and A2
                 (not
                   (and (and A4 (not (and (and A1 (not A2)) A3)))
                        (and A3 (not (and A1 (not A2)))))))
            (not (and (and A3 (not (and A1 (not A2))))
                   (not (and A4 (not (and (and A1 (not A2)) A3)))))))
          (not (and (and A4 (not (and (and A1 (not A2)) A3)))
                    (not (and A3 (not (and A1 (not A2))))))))

X3: (and (and (and A1 (not A2)) A3) (not A4))

X10: (and (and
            (and A2
                 (not
                   (and (and A4 (not (and (and A1 (not A2)) A3)))
                        (and A3 (not (and A1 (not A2)))))))
            (not (and (and A3 (not (and A1 (not A2))))
                   (not (and A4 (not (and (and A1 (not A2)) A3)))))))
          (and (and A4 (not (and (and A1 (not A2)) A3)))
               (not (and A3 (not (and A1 (not A2))))))))
```

Problems with baseline algorithm

- Explosive size of type specifiers
- $O(n^2)$ search on each iteration
- Set semantics for lists of types:
  - To uniquify a list: $O(n^2)$.
  - Equivalent types may appear in many different forms.
    ... No canonical form
  - Slow set-equivalence algorithm.
- Many redundant checks
- `subtypep` may return `don't-know`

Strategies to Improving MDTD algorithm

We can do better.

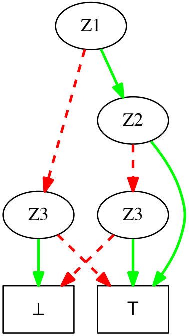
- Optimize current algorithm (caching etc).
- Change the algorithm.
- Change the data structure representing the sets (CL types).

ROBDD: Reduced Ordered Binary Decision Diagrams

What is an ROBDD?

An ROBDD is an EQ-canonical representation for a Boolean function

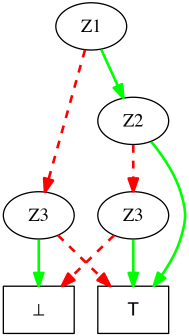
$$\neg(\neg Z_1 \wedge Z_3) \vee (Z_1 \wedge \neg Z_2 \wedge \neg Z_3)$$
$$= (Z_1 \wedge Z_2) \vee (Z_1 \wedge \neg Z_2 \wedge Z_3) \vee (\neg Z_1 \wedge \neg Z_3)$$
$$= \big((Z_1 \vee \neg Z_2) \wedge (Z_1 \vee Z_3) \wedge (\neg Z_1 \vee Z_2) \wedge (Z_2 \vee Z_3)\big) \vee (\neg Z_1 \wedge \neg Z_3)$$



What is an ROBDD?

An ROBDD is an EQ-canonical representation for a Boolean function and an efficient evaluation procedure.

$$\neg(\neg Z_1 \wedge Z_3) \vee (Z_1 \wedge \neg Z_2 \wedge \neg Z_3)$$
$$= (Z_1 \wedge Z_2) \vee (Z_1 \wedge \neg Z_2 \wedge Z_3) \vee (\neg Z_1 \wedge \neg Z_3)$$
$$= \big((Z_1 \vee \neg Z_2) \wedge (Z_1 \vee Z_3) \wedge (\neg Z_1 \vee Z_2) \wedge (Z_2 \vee Z_3)\big) \vee (\neg Z_1 \wedge \neg Z_3)$$



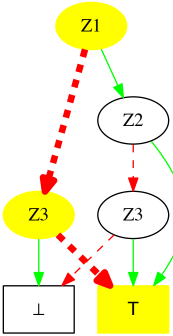
Given assignments for the Boolean variables, trace through the BDD to obtain true or false.

What is an ROBDD?

An ROBDD is an EQ-canonical representation for a Boolean function and an efficient evaluation procedure.

To compute a DNF iteratively, follow all paths from Z_1 to $\top$, noting the green and red arrows.

$$\overbrace{(\neg Z_1 \wedge \neg Z_3)}^{Z_1 \rightarrow Z_2 \rightarrow \top} \vee (Z_1 \wedge \neg Z_2 \wedge Z_3) \vee (Z_1 \wedge Z_2)$$

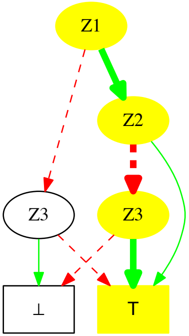


What is an ROBDD?

An ROBDD is an EQ-canonical representation for a Boolean function and an efficient evaluation procedure.

To compute a DNF iteratively, follow all paths from Z_1 to $\top$, noting the green and red arrows.

$$(\neg Z_1 \wedge \neg Z_3) \vee \underbrace{(Z_1 \wedge \neg Z_2 \wedge Z_3)}_{Z_1 \rightarrow Z_2 \rightarrow Z_3 \rightarrow \top} \vee (Z_1 \wedge Z_2)$$

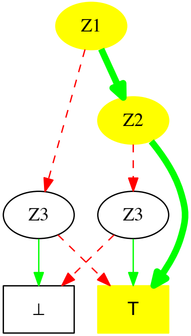


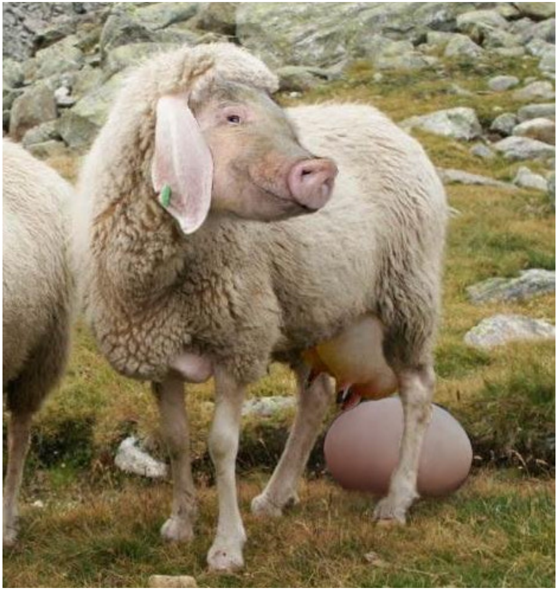
What is an ROBDD?

An ROBDD is an EQ-canonical representation for a Boolean function and an efficient evaluation procedure.

To compute a DNF iteratively, follow all paths from Z_1 to $\top$, noting the green and red arrows.

$$(\neg Z_1 \wedge \neg Z_3) \vee (Z_1 \wedge \neg Z_2 \wedge Z_3) \vee \overbrace{(Z_1 \wedge Z_2)}^{Z_1 \rightarrow Z_2 \rightarrow \top}$$





Creative Commons Attribution ShareAlike, Author: Georg Mittenecker

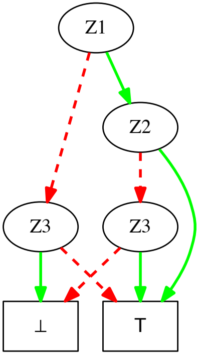
The BDD is the *Eierlegende Wollmilchsau* of Boolean algebra.

BDDs have many (many many..) surprising features and uses.

The same ROBDD also represents the corresponding CL type specifier and type predicate procedure—no duplicate type checks.

$$(Z_1 \wedge Z_2) \vee (Z_1 \wedge \neg Z_2 \wedge Z_3) \vee (\neg Z_1 \wedge \neg Z_3)$$

```
(or (and Z1 Z2)
    (and Z1 (not Z2) Z3)
    (and (not Z1) (not Z3)))
```

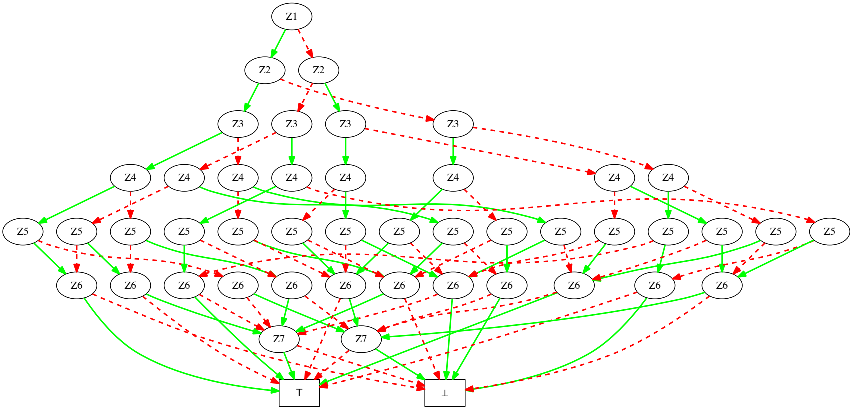


How efficient is ROBDD compression

- What is the worst-case size of an *n*-variable ROBDD?
- What is expected size?

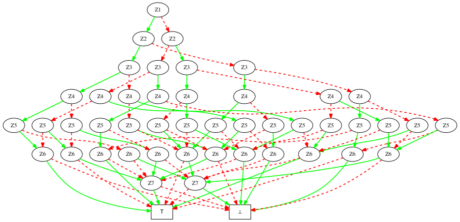
We publish a journal article in *ACM: Transactions on Computational Logic* entitled: *A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams*.

Shape of worst-case ROBDD of *n* Boolean variables?



Worst-case ROBDD has exponential 2^i expansion from top to the *belt*, and double exponential 2^{2^i} decay from the *belt* to bottom.

Shape of worst-case ROBDD of *n* Boolean variables?

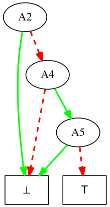
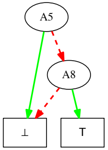


Worst-case ROBDD has exponential 2^i expansion from top to the *belt*, and double exponential 2^{2^i} decay from the *belt* to bottom.

However, the worst-case size of the Common Lisp s-expression form of a type specifier has exponential size, but no double-exponential decay.

We can revisit MDTD algorithms using the ROBDD to represent type specifiers.

We must break the green line joining nodes 4 and 8.

$Node_4 = A_4 \cap \overline{A_5} \cap \overline{A_2}$	$Node_8 = A_8 \cap \overline{A_5}$
	

We must calculate the standard partition:

$$A_4 \cap \overline{A_5} \cap \overline{A_2} \quad \cap \quad A_8 \cap \overline{A_5} \tag{1}$$

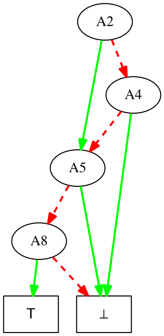
$$A_4 \cap \overline{A_5} \cap \overline{A_2} \quad \cap \quad \overline{A_8 \cap \overline{A_5}} \tag{2}$$

$$\overline{A_4 \cap \overline{A_5} \cap \overline{A_2}} \quad \cap \quad A_8 \cap \overline{A_5} \tag{3}$$

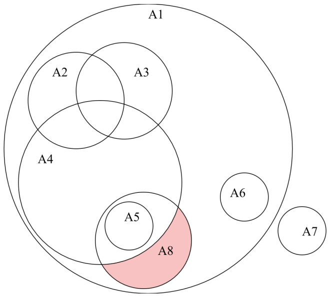
Extending ROBDDs for compatibility with CL type system

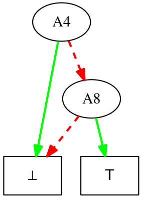
- Traditionally, ROBDDs assume the Boolean variables are independent.
- We propose extending ROBDDs to understand subtype relations.

$$X_8 = \overline{Node_4} \cap Node_8 = \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}} \quad \cap \quad A_8 \cap \overline{A_5}$$



Before

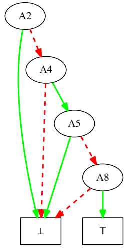
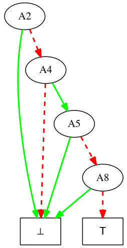
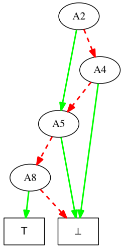
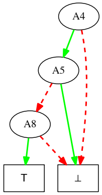
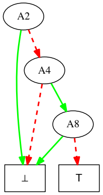
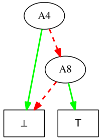




After

We propose simplifying ROBDDs in the presence of subtypes.

The standard partition is sometimes simpler.

	$Node_4 \land Node_8$	$Node_4 \land \neg Node_8$	$\neg Node_4 \land Node_8$
Before			
After			

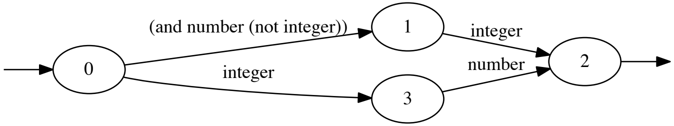
Features of ROBDDs

- Refactor MDTD algorithms to use ROBDDs.
- ROBDDs are algorithmically easy to construct,
- ... especially in a language with garbage collection.
- Systematically manipulate Boolean operations: $\vee$, $\wedge$, $\oplus$, $\neg$.
- Exponential in size, but simplify in presence of subtyping.
- Provide structural equivalence.
  - Uniquify set becomes $O(n \log n)$ rather than $O(n^2)$.
- Serializable to `if`/`then`/`else` code; *Will see shortly.*
  - Redundant checks optimized away.

Optimizing type checking

Recall the DFA problem?

RTE: $(number \cdot integer) \vee (integer \cdot number)$



DFA: leads to inefficient generated code; redundant type checks.

```
X0 (unless seq (return nil))
   (typecase (pop seq)
     (integer
      (go X3))
     ((and number
           (not integer)) ; duplicate type check :-(
      (go X1))
     (t (return nil)))
```

We'd like to build an ROBDD to represent a typecase.

We know how to generate efficient code from an ROBDD.

- Convert typecase into Boolean expression

```
(typecase obj
  (T.1 alternative-1)
  (T.2 alternative-2)
  ...
  (T.n alternative-n))
```

We'd like to build an ROBDD to represent a `typecase`.

We know how to generate efficient code from an ROBDD.

- Convert typecase into Boolean expression

```
(typecase obj
  (T.1 alternative-1)
  (T.2 alternative-2)
  ...
  (T.n alternative-n))
```

- Transform *alternatives* with side-effects into predicates pretending side-effect free.

  `alternative-1` $\to$ *Pseudo.type*-1
  `alternative-2` $\to$ *Pseudo.type*-2
  ...
  `alternative-n` $\to$ *Pseudo.type*-n

Transform typecase into type specifier

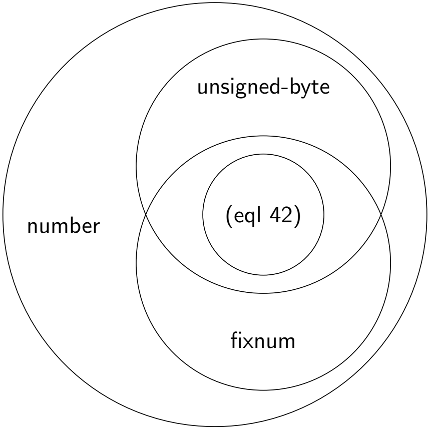
```
(typecase obj
  (T.1 Pseudo.type.1)  ; alternative-1
  (T.2 Pseudo.type.2)  ; alternative-2
  ...
  (T.n Pseudo.type.n)) ; alternative-n
```

Now this pure Boolean expression can be converted to DNF.

```
(or (and T.1
         Pseudo.type.1)
    (and T.2 (not T.1)
         Pseudo.type.2)
    ...
    (and T.n
         (not T.1) (not T.2) ... (not T.n-1)
         Pseudo.type.n))
```

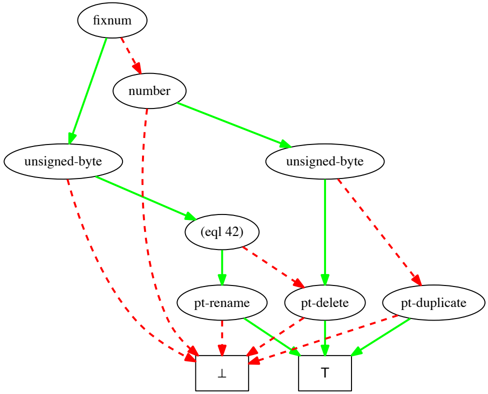
Another bdd-typecase example

```
(bdd-typecase obj
  ((and unsigned-byte
        (not (eql 42)))
   (delete-file))

  ((eql 42)
   (rename-file))

  ((and number
        (not (eql 42))
        (not fixnum))
   (duplicate-file))

  ((and (not fixnum)
        unsigned-byte)
   (launch-missiles)))
```

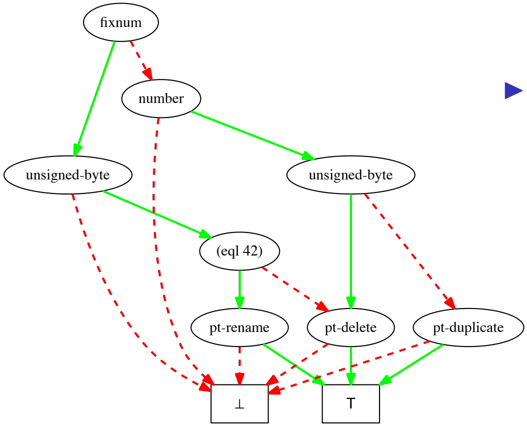


Another bdd-typecase example

```
(bdd-typecase obj
  ((and unsigned-byte
        (not (eql 42)))
   (delete-file))

  ((eql 42)
   (rename-file))

  ((and number
        (not (eql 42))
        (not fixnum))
   (duplicate-file))

  ((and (not fixnum)
        unsigned-byte)
   (launch-missiles)))
```

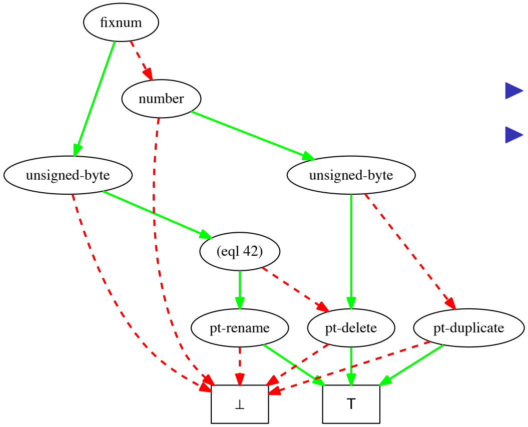


Properties of `bdd-typecase`



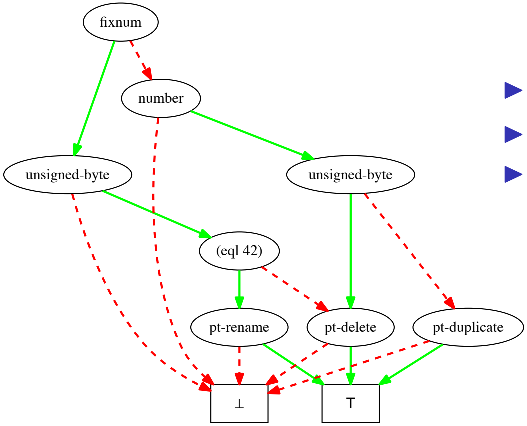
- No duplicate type checks.

Properties of `bdd-typecase`



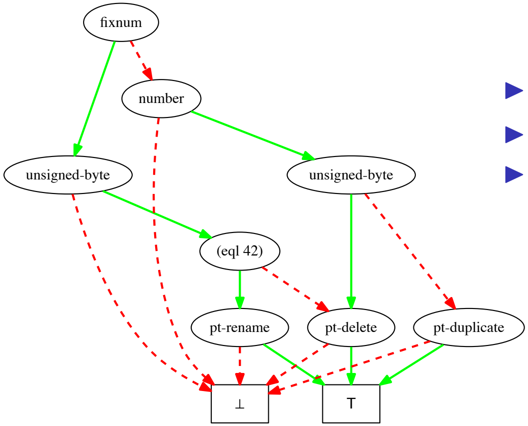
- No duplicate type checks.
- No super-type checks.

Properties of `bdd-typecase`



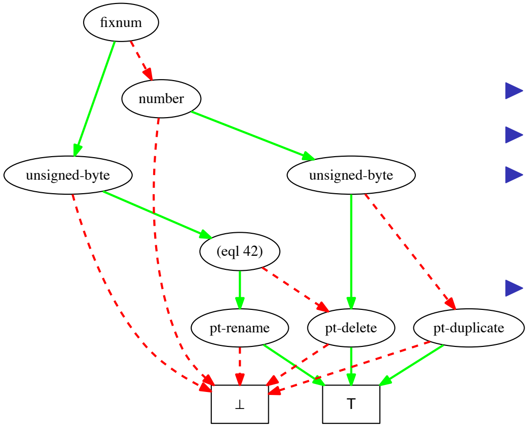
- No duplicate type checks.
- No super-type checks.
- Missing `Pseudo...` implies unreachable code.

Properties of `bdd-typecase`



- No duplicate type checks.
- No super-type checks.
- Missing `Pseudo...` implies unreachable code.
  - No missiles launched!

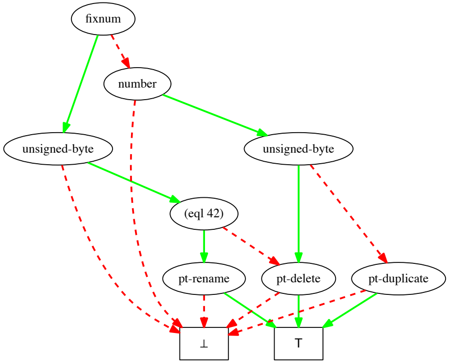
Properties of `bdd-typecase`



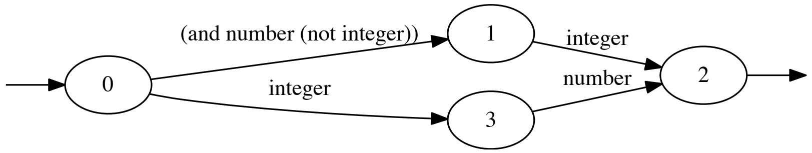
- No duplicate type checks.
- No super-type checks.
- Missing `Pseudo...` implies unreachable code.
  - No missiles launched!
- Serializable to efficient code.

Machine generated code with tagbody/go.

```
(tagbody
 L1 (if (typep obj 'fixnum)
        (go L2)
        (go L4))
 L2 (if (typep obj 'unsigned-byte)
        (go L3)
        (return nil))
 L3 (if (typep obj '(eql 42))
        (go P1)
        (go P2))
 L4 (if (typep obj 'number)
        (go L5)
        (return nil))
 L5 (if (typep obj 'unsigned-byte)
        (go P2)
        (go P3))

 P1 (return (rename-file))
 P2 (return (delete-file))
 P3 (return (duplicate-file)))
```

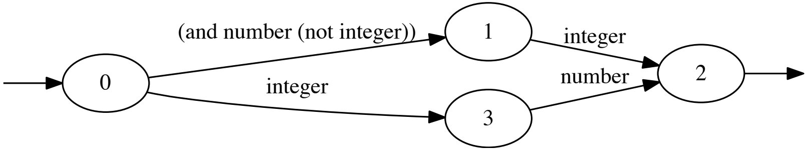


Back to the *deterministic* state machine

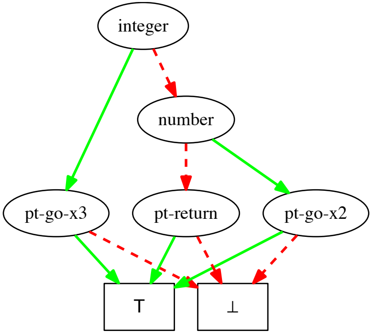


```
X0 (unless seq
     (return nil))

(bdd-typecase (pop seq)
  (integer
   (go X3))

  ((and number
         (not integer))
   (go X1))

  (t
   (return nil)))
```

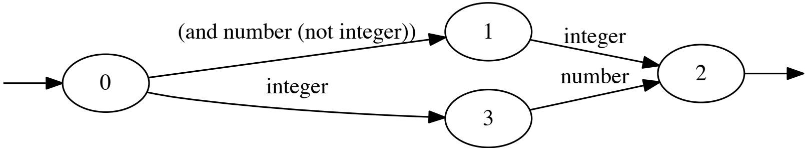
Back to the *deterministic* state machine



```
X0 (unless seq
     (return nil))

(bdd-typecase (pop seq)
  (integer
   (go X3))

  ((and number
        (not integer))
   (go X1))

  (t
   (return nil)))
```



Back to the *deterministic* state machine

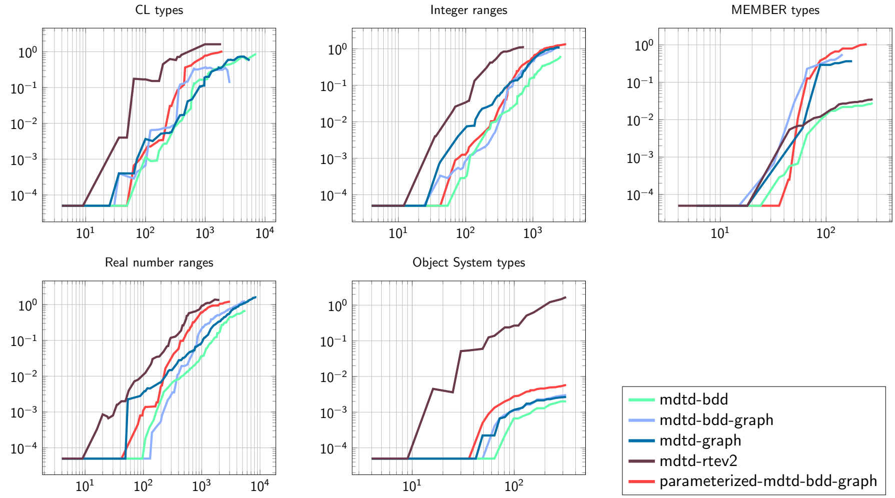


```
X0 (unless seq
     (return nil))

(bdd-typecase (pop seq)
  (integer
   (go X3))

  ((and number
        (not integer))
   (go X1))

  (t
   (return nil)))
```

```
X0 (unless seq
     (return nil))

  (let ((obj (pop seq)))
    (tagbody
     L0 (if (typep obj 'integer)
            (go P0)
            (go L2))
     L2 (if (typep obj 'number)
            (go P1)
            (go P2))

     P0 (go X3)
     P1 (go X1)
     P2 (return nil)))
```

Results and Conclusions

Performance comparison using various algorithms



All plots show $y = time_{computation}$ vs. $x = size_{input} \times size_{output}$.

ROBDD worst case size

N	$\lvert ROBDD_N \rvert$
1	3
2	5
3	7
4	11
5	19
6	31
7	47
8	79
9	143
10	271
11	511
12	767
13	1279
14	2303
15	4351

- Number of labels is number of nodes in the ROBDD.

ROBDD worst case size

N	$\lvert ROBDD_N \rvert$
1	3
2	5
3	7
4	11
5	19
6	31
7	47
8	79
9	143
10	271
11	511
12	767
13	1279
14	2303
15	4351

- Number of labels is number of nodes in the ROBDD.
- Worst case code size for N type checks (including pseudo-predicates), proportional to full ROBDD size for N variables.

ROBDD worst case size

N	$\vert ROBDD_N \vert$
1	3
2	5
3	7
4	11
5	19
6	31
7	47
8	79
9	143
10	271
11	511
12	767
13	1279
14	2303
15	4351

- Number of labels is number of nodes in the ROBDD.
- Worst case code size for N type checks (including pseudo-predicates), proportional to full ROBDD size for N variables.
- But our ROBDD is never worst-case.

Summary of Contributions

- Common Lisp types augmented to support regular type expressions

Summary of Contributions

- Common Lisp types augmented to support regular type expressions
  - ... extending rational language theory and ROBDDs
  - ... to accommodate subtyping.

Summary of Contributions

- Common Lisp types augmented to support regular type expressions
  - ... extending rational language theory and ROBDDs
  - ... to accommodate subtyping.
- Released open source versions of several Common Lisp packages developed for the thesis. Available on Quicklisp and LRDE GitLab.

Summary of Contributions

- Common Lisp types augmented to support regular type expressions
  - ... extending rational language theory and ROBDDs
  - ... to accommodate subtyping.
- Released open source versions of several Common Lisp packages developed for the thesis. Available on Quicklisp and LRDE GitLab.
- Demonstrated use of BDDs to represent and compute with Common Lisp types.

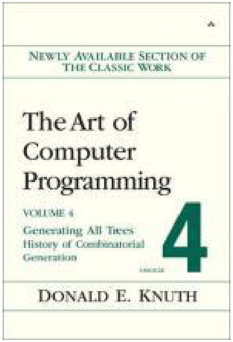
Summary of Contributions

- Common Lisp types augmented to support regular type expressions
  - ... extending rational language theory and ROBDDs
  - ... to accommodate subtyping.
- Released open source versions of several Common Lisp packages developed for the thesis. Available on Quicklisp and LRDE GitLab.
- Demonstrated use of BDDs to represent and compute with Common Lisp types.
- Journal publication: *ACM Transactions on Computational Logic*, A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams.

Summary of Contributions

- Common Lisp types augmented to support regular type expressions
  - ... extending rational language theory and ROBDDs
  - ... to accommodate subtyping.
- Released open source versions of several Common Lisp packages developed for the thesis. Available on Quicklisp and LRDE GitLab.
- Demonstrated use of BDDs to represent and compute with Common Lisp types.
- Journal publication: *ACM Transactions on Computational Logic*, A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams.
- Published and particapted each year (3 times) in European Lisp Symposium

Donald Knuth's new toy.


Newly Available Section of The Classic Work
The Art of Computer Programming
Volume 4
Generating All Trees
History of Combinatorial Generation
4
Donald E. Knuth

Binary decision diagrams (ROBDDs) are wonderful, and the more I play with them the more I love them. For fifteen months I've been like a child with a new toy, being able now to solve problems that I never imagined would be tractable... I suspect that many readers will have the same experience ... there will always be more to learn about such a fertile subject.
[Donald Knuth, *Art of Computer Science, Volume 4*]

Q/A

Questions?

Code available at
`https://gitlab.lrde.epita.fr/jnewton/regular-type-expression`

and also `(ql:quickload :regular-type-expression)`

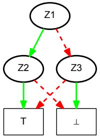
Perspectives

- Better describe (or characterize) which MDTD algorithms are better for which kind of input.
- ... Performance tests with minimal sized ROBDD structures.
- Improve s-expression based manipulation.
- `subtypep` can almost be implemented in terms of ROBDD operations.
- Extend `destructuring-case`, remove duplication, detect vacuity. (ELS 2019?)
- Improve the decision procedure of PCL incorporating SICL technique of inlining constants.
- Extend to other dynamic languages? Possible?

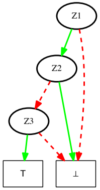
OPTION A
OPTION B
OPTION B IT IS.

Algebra of ROBDDs

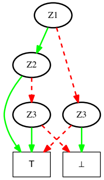
$$(Z_1 \land Z_2 \lor \neg Z_1 \land \neg Z_2) \quad \lor \quad (Z_1 \land \neg Z_2 \land Z_3)$$
$$= Z_1 \land Z_2 \quad \lor \quad \neg Z_1 \land \neg Z_2 \quad \lor \quad Z_1 \land \neg Z_2 \land Z_3$$



$\bigvee$



$=$

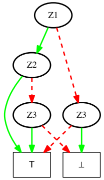


Algebra of ROBDDs

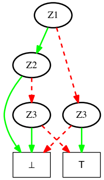
Negation is easy, just swap the true/false nodes.

$$\neg(Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2 \vee Z_1 \wedge \neg Z_2 \wedge Z_3)$$
$$= Z_1 \wedge \neg Z_2 \wedge \neg Z_3 \vee \neg Z_1 \wedge Z_3$$

$\neg$



$=$



Programmatic treatment of an RTE

By homoiconicity we treat the surface syntax as the internal representation.

```
(defun walk-rte (transform pattern)
  (typecase pattern
    ((cons (member :or :and :not :cat :* :+ :?))
     (cons (first pattern)
           (mapcar (lambda (p)
                     (walk-rte transform p))
                   (rest pattern))))
    ...

    (t
     (funcall transform pattern))))
```

Use tail-call optimized local functions, if the target language does not support GOTO?

```
(labels ((L1 () (if (typep obj 'fixnum)
                    (L2)
                    (L4)))
         (L2 () (if (typep obj 'unsigned-byte)
                    (L3)
                    nil))
         (L3 () (if (typep obj '(eql 42))
                    (P1)
                    (P2)))
         (L4 () (if (typep obj 'number)
                    (L5)
                    nil))
         (L5 () (if (typep obj 'unsigned-byte)
                    (P2)
                    (P3)))

         (P1 () (rename-file))
         (P2 () (delete-file))
         (P3 () (duplicate-file)))
  (L1))
```

Common Lisp and types

- Type declarations in structured data and functions.

```
(defclass circle ()
  ((radius :type real)  ; restrict slot to certain type
   (center :type cons)))

(defun cube-root (x)
  (declare (type real x)) ; promise to compiler
  (expt x 1/3))
```

Common Lisp and types

- Type declarations in structured data and functions.
- Arbitrary logic at run-time.

```
(defun stringify (data)
  (typecase data ; priority based type test
    (string data)
    (symbol (symbol-name data))
    (list   (mapcar #'stringify data))))
```

With the type *definition* `(rte ...)` we can use the surface syntax anywhere Common Lisp allows a type specifier.

```
(defclass polygon ()
  ((color)
   (points :type (rte (:* (:cat fixnum real))))))

(defun fun-42 (float-plist)
  (declare (type (rte (:+ (:cat keyword float)))
                 float-plist))
  ...)
```

With the type *definition* `(rte ...)` we can use the surface syntax anywhere Common Lisp allows a type specifier.

```
(defclass polygon ()
  ((color)
   (points :type (rte (:* (:cat fixnum real))))))

(defun fun-42 (float-plist)
  (declare (type (rte (:+ (:cat keyword float)))
                 float-plist))
  ...)
```

By homoiconicity we treat the surface syntax as the internal representation.

Programmatic treatment of an RTE

By homoiconicity we treat the surface syntax as the internal representation.

```
(defun walk-rte (transform pattern)
  (typecase pattern
    ((cons (member :or :and :not :cat :* :+ :?))
     (cons (first pattern)
           (mapcar (lambda (p)
                     (walk-rte transform p))
                   (rest pattern))))
    ...

    (t
     (funcall transform pattern))))
```

Baseline Demo Step 1

We can observe the procedure execution textually as well.

The explosive size of the type specifiers becomes evident.

```
found 1 disjoint:
new-disjoint
  D1
D = (
   1: A7
)
U = (
   1: A1
   2: A2
   3: A3
   4: A4
   5: A5
   6: A6
   7: A8
)
intersecting: U1 U2
```

Baseline Demo Step 2

```
found 0 disjoint:
new-disjoint ()
D = (
   1: A7
)
U = (
   1: (and A1 (not A2))
   2: A2
   3: A3
   4: A4
   5: A5
   6: A6
   7: A8
)
intersecting: U1 U3
```

Baseline Demo Step 3

```
found 0 disjoint:
new-disjoint ()
D = (
  1: A7
)
U = (
  1: (and (and A1 (not A2)) A3)
  2: (and A3 (not (and A1 (not A2))))
  3: (and (and A1 (not A2)) (not A3))
  4: A2
  5: A4
  6: A5
  7: A6
  8: A8
)
intersecting: U1 U4
```

Baseline Demo Step 4

```
found 2 new disjoint:
  D1 D2
D = (
   1: (and (and (and A1 (not A2)) A3) (not A4))
   2: (and (and (and A1 (not A2)) A3) A4)
   3: A7
)
U = (
   1: (and A4 (not (and (and A1 (not A2)) A3)))
   2: (and A3 (not (and A1 (not A2))))
   3: (and (and A1 (not A2)) (not A3))
   4: A2
   5: A5
   6: A6
   7: A8
)
intersecting: U1 U2
```

Baseline Demo Step 5

```
found 0 new disjoint:
D = (
   1: (and (and (and A1 (not A2)) A3) (not A4))
   2: (and (and (and A1 (not A2)) A3) A4)
   3: A7
)
U = (
   1: (and (and A4 (not (and (and A1 (not A2)) A3)))
           (and A3 (not (and A1 (not A2)))))
   2: (and (and A3 (not (and A1 (not A2))))
           (not (and A4 (not (and (and A1 (not A2)) A3)))))
   3: (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2))))))
   4: (and (and A1 (not A2)) (not A3))
   5: A2
   6: A5
   7: A6
   8: A8
)
intersecting: U1 U5
```

Baseline Demo Step 6

```
found 1 new disjoint:
      D1
D = (  1: (and (and A4 (not (and (and A1 (not A2)) A3)))
               (and A3 (not (and A1 (not A2)))))
       2: (and (and (and A1 (not A2)) A3) (not A4))
       3: (and (and (and A1 (not A2)) A3) A4)
       4: A7
)
U = (  1: (and A2
               (not
                (and (and A4 (not (and (and A1 (not A2)) A3)))
                     (and A3 (not (and A1 (not A2)))))))
       2: (and (and A3 (not (and A1 (not A2))))
               (not (and A4 (not (and (and A1 (not A2)) A3)))))
       3: (and (and A4 (not (and (and A1 (not A2)) A3)))
               (not (and A3 (not (and A1 (not A2))))))
       4: (and (and A1 (not A2)) (not A3))
       5: A5
       6: A6
       7: A8
)
intersecting: U1 U2
```

Baseline Demo Step 7

```
found 1 disjoint:
      D1
D = (  1: (and (and A3 (not (and A1 (not A2))))
               (not (and A4 (not (and (and A1 (not A2)) A3)))))
       2: (and (and A4 (not (and (and A1 (not A2)) A3)))
               (and A3 (not (and A1 (not A2)))))
       3: (and (and (and A1 (not A2)) A3) (not A4))
       4: (and (and (and A1 (not A2)) A3) A4)
       5: A7
)
U = (  1: (and
           (and A2
                (not
                 (and (and A4 (not (and (and A1 (not A2)) A3)))
                      (and A3 (not (and A1 (not A2)))))))
           (not
            (and (and A3 (not (and A1 (not A2))))
                 (not (and A4 (not (and (and A1 (not A2)) A3)))))))
       2: (and (and A4 (not (and (and A1 (not A2)) A3)))
               (not (and A3 (not (and A1 (not A2))))))
       3: (and (and A1 (not A2)) (not A3))
       4: A5
       5: A6
       6: A8
)
intersecting:
  U1 U2
```

Baseline Demo Step 8

```
found 2 disjoint:
new-disjoint
  D1 D2
D = (  1: (and
          (and
           (and A2
                (not
                 (and (and A4 (not (and (and A1 (not A2)) A3)))
                      (and A3 (not (and A1 (not A2)))))))
           (not
            (and (and A3 (not (and A1 (not A2))))
                 (not (and A4 (not (and (and A1 (not A2)) A3)))))))
          (not
           (and (and A4 (not (and (and A1 (not A2)) A3)))
                (not (and A3 (not (and A1 (not A2))))))))
       2: (and
          (and
           (and A2
                (not
                 (and (and A4 (not (and (and A1 (not A2)) A3)))
                      (and A3 (not (and A1 (not A2)))))))
           (not
            (and (and A3 (not (and A1 (not A2))))
                 (not (and A4 (not (and (and A1 (not A2)) A3)))))))
          (and (and A4 (not (and (and A1 (not A2)) A3)))
               (not (and A3 (not (and A1 (not A2)))))))
       3: (and (and A3 (not (and A1 (not A2))))
               (not (and A4 (not (and (and A1 (not A2)) A3)))))
       4: (and (and A4 (not (and (and A1 (not A2)) A3)))
               (and A3 (not (and A1 (not A2)))))
       5: (and (and (and A1 (not A2)) A3) (not A4))
       6: (and (and (and A1 (not A2)) A3) A4)
       7: A7
)
U = (  1: (and
           (and (and A4 (not (and (and A1 (not A2)) A3)))
                (not (and A3 (not (and A1 (not A2))))))
           (not
            (and
             (and A2
                  (not
                   (and (and A4 (not (and (and A1 (not A2)) A3)))
                        (and A3 (not (and A1 (not A2)))))))
             (not
              (and (and A3 (not (and A1 (not A2))))
                   (not (and A4 (not (and (and A1 (not A2)) A3)))))))))
       2: (and (and A1 (not A2)) (not A3))
       3: A5
       4: A6
       5: A8
)
intersecting:
  U1 U2
```

Baseline Demo Step 9

```
found 0 disjoint:
D (
  1: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (not
       (and (and A4 (not (and (and A1 (not A2)) A3)))
            (not (and A3 (not (and A1 (not A2))))))))
  2: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2)))))))
  3: (and (and A3 (not (and A1 (not A2))))
          (not (and A4 (not (and (and A1 (not A2)) A3)))))
  4: (and (and A4 (not (and (and A1 (not A2)) A3)))
          (and A3 (not (and A1 (not A2)))))
  5: (and (and (and A1 (not A2)) A3) (not A4))
  6: (and (and (and A1 (not A2)) A3) A4)
  7: A7
)
```

```
U (
  1: (and (and (and A1 (not A2)) (not A3))
          (not
           (and
            (and (and A4 (not (and (and A1 (not A2)) A3)))
                 (not (and A3 (not (and A1 (not A2))))))
            (not
             (and
              (and A2
                   (not
                    (and (and A4 (not (and (and A1 (not A2)) A3)))
                         (and A3 (not (and A1 (not A2)))))))
              (not
               (and (and A3 (not (and A1 (not A2))))
                    (not (and A4 (not (and (and A1 (not A2)) A3)))))))))))
  2: (and
      (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2))))))
      (not
       (and
        (and A2
             (not
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (and A3 (not (and A1 (not A2)))))))
        (not
         (and (and A3 (not (and A1 (not A2))))
              (not (and A4 (not (and (and A1 (not A2)) A3)))))))))
  3: A5
  4: A6
  5: A8
)
intersecting:
  U1 U4
```

Baseline Demo Step 10

```
found 1 disjoint:
  D1
D=8 U=4
D (
  1: A6
  2: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (not
       (and (and A4 (not (and (and A1 (not A2)) A3)))
            (not (and A3 (not (and A1 (not A2))))))))
  3: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2)))))))
  4: (and (and A3 (not (and A1 (not A2))))
          (not (and A4 (not (and (and A1 (not A2)) A3)))))
  5: (and (and A4 (not (and (and A1 (not A2)) A3)))
          (and A3 (not (and A1 (not A2)))))
  6: (and (and (and A1 (not A2)) A3) (not A4))
  7: (and (and (and A1 (not A2)) A3) A4)
  8: A7
)
```

```
U (
  1: (and
      (and (and (and A1 (not A2)) (not A3))
           (not
            (and
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (not (and A3 (not (and A1 (not A2))))))
             (not
              (and
               (and A2
                    (not
                     (and (and A4 (not (and (and A1 (not A2)) A3)))
                          (and A3 (not (and A1 (not A2)))))))
               (not
                (and (and A3 (not (and A1 (not A2))))
                     (not (and A4 (not (and (and A1 (not A2)) A3))))))))))
      (not A6))
  2: (and
      (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2))))))
      (not
       (and
        (and A2
             (not
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (and A3 (not (and A1 (not A2)))))))
        (not
         (and (and A3 (not (and A1 (not A2))))
              (not (and A4 (not (and (and A1 (not A2)) A3)))))))))
  3: A5
  4: A8
)
intersecting:
  U1 U4
```

Baseline Demo Step 11

```
found 2 disjoint:
  D1 D2
D=10 U=3
D (
  1: (and
      (and
       (and (and (and A1 (not A2)) (not A3))
            (not
             (and
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (not (and A3 (not (and A1 (not A2))))))
              (not
               (and
                (and A2
                     (not
                      (and (and A4 (not (and (and A1 (not A2)) A3)))
                           (and A3 (not (and A1 (not A2)))))))
                (not
                 (and (and A3 (not (and A1 (not A2))))
                      (not (and A4 (not (and (and A1 (not A2)) A3)))))))))))
       (not A6))
      (not A8))
  2: (and
      (and
       (and (and (and A1 (not A2)) (not A3))
            (not
             (and
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (not (and A3 (not (and A1 (not A2))))))
              (not
               (and
                (and A2
                     (not
                      (and (and A4 (not (and (and A1 (not A2)) A3)))
                           (and A3 (not (and A1 (not A2)))))))
                (not
                 (and (and A3 (not (and A1 (not A2))))
                      (not (and A4 (not (and (and A1 (not A2)) A3)))))))))))
       (not A6))
      A8)
  3: A6
  4: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (not
       (and (and A4 (not (and (and A1 (not A2)) A3)))
            (not (and A3 (not (and A1 (not A2))))))))
  5: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2))))))))
  6: (and (and A3 (not (and A1 (not A2))))
          (not (and A4 (not (and (and A1 (not A2)) A3)))))
  7: (and (and A4 (not (and (and A1 (not A2)) A3)))
          (and A3 (not (and A1 (not A2)))))
  8: (and (and (and A1 (not A2)) A3) (not A4))
  9: (and (and (and A1 (not A2)) A3) A4)
  10: A7
)
U (
  1: (and A8
          (not
           (and
            (and (and (and A1 (not A2)) (not A3))
                 (not
                  (and
                   (and (and A4 (not (and (and A1 (not A2)) A3)))
                        (not (and A3 (not (and A1 (not A2))))))
                   (not
                    (and
                     (and A2
                          (not
                           (and (and A4 (not (and (and A1 (not A2)) A3)))
                                (and A3 (not (and A1 (not A2)))))))
                     (not
                      (and (and A3 (not (and A1 (not A2))))
                           (not
                            (and A4 (not (and (and A1 (not A2)) A3)))))))))))
            (not A6))))
  2: (and
      (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2))))))
      (not
       (and
        (and A2
             (not
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (and A3 (not (and A1 (not A2)))))))
        (not
         (and (and A3 (not (and A1 (not A2))))
              (not (and A4 (not (and (and A1 (not A2)) A3)))))))))
  3: A5
)
intersecting:
  U1 U2
```

Baseline Demo Step 12

```
found 1 disjoint:
  D1 D2
D (
  1: (and
      (and
       (and (and A4 (not (and (and A1 (not A2)) A3)))
            (not (and A3 (not (and A1 (not A2))))))
       (not
        (and
         (and A2
              (not
               (and (and A4 (not (and (and A1 (not A2)) A3)))
                    (and A3 (not (and A1 (not A2)))))))
         (not
          (and (and A3 (not (and A1 (not A2))))
               (not (and A4 (not (and (and A1 (not A2)) A3)))))))))
      (not
       (and A8
            (not
             (and
              (and (and (and A1 (not A2)) (not A3))
                   (not
                    (and
                     (and (and A4 (not (and (and A1 (not A2)) A3)))
                          (not (and A3 (not (and A1 (not A2))))))
                     (not
                      (and
                       (and A2
                            (not
                             (and (and A4 (not (and (and A1 (not A2)) A3)))
                                  (and A3 (not (and A1 (not A2)))))))
                       (not
                        (and (and A3 (not (and A1 (not A2))))
                             (not
                              (and A4 (not (and (and A1 (not A2)) A3)))))))))))
              (not A6))))))
  2: (and
      (and
       (and (and (and A1 (not A2)) (not A3))
            (not
             (and
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (not (and A3 (not (and A1 (not A2))))))
              (not
               (and
                (and A2
                     (not
                      (and (and A4 (not (and (and A1 (not A2)) A3)))
                           (and A3 (not (and A1 (not A2)))))))
                (not
                 (and (and A3 (not (and A1 (not A2))))
                      (not (and A4 (not (and (and A1 (not A2)) A3)))))))))))
       (not A6))
      (not A8))
  3: (and
      (and
       (and (and (and A1 (not A2)) (not A3))
            (not
             (and
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (not (and A3 (not (and A1 (not A2))))))
              (not
               (and
                (and A2
                     (not
                      (and (and A4 (not (and (and A1 (not A2)) A3)))
                           (and A3 (not (and A1 (not A2)))))))
                (not
                 (and (and A3 (not (and A1 (not A2))))
                      (not (and A4 (not (and (and A1 (not A2)) A3)))))))))))
       (not A6))
      A8)
  4: A6
  5: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (not
       (and (and A4 (not (and (and A1 (not A2)) A3)))
            (not (and A3 (not (and A1 (not A2))))))))
  6: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2)))))))
  7: (and (and A3 (not (and A1 (not A2))))
          (not (and A4 (not (and (and A1 (not A2)) A3)))))
  8: (and (and A4 (not (and (and A1 (not A2)) A3)))
          (and A3 (not (and A1 (not A2)))))
  9: (and (and (and A1 (not A2)) A3) (not A4))
  10: (and (and (and A1 (not A2)) A3) A4)
  11: A7
)
U (
  1: (and A8
          (not
           (and
            (and (and (and A1 (not A2)) (not A3))
                 (not
                  (and
                   (and (and A4 (not (and (and A1 (not A2)) A3)))
                        (not (and A3 (not (and A1 (not A2))))))
                   (not
                    (and
                     (and A2
                          (not
                           (and (and A4 (not (and (and A1 (not A2)) A3)))
                                (and A3 (not (and A1 (not A2)))))))
                     (not
                      (and (and A3 (not (and A1 (not A2))))
                           (not
                            (and A4 (not (and (and A1 (not A2)) A3)))))))))))
            (not A6))))
  2: A5
)
intersecting:
  U1 U2
```

Baseline Demo Step 13

```
found 2 disjoint:
  D1 D2
D (
  1: A5
  2: (and
      (and A8
           (not
            (and
             (and (and (and A1 (not A2)) (not A3))
                  (not
                   (and
                    (and (and A4 (not (and (and A1 (not A2)) A3)))
                         (not (and A3 (not (and A1 (not A2))))))
                    (not
                     (and
                      (and A2
                           (not
                            (and (and A4 (not (and (and A1 (not A2)) A3)))
                                 (and A3 (not (and A1 (not A2)))))))
                      (not
                       (and (and A3 (not (and A1 (not A2))))
                            (not
                             (and A4 (not (and (and A1 (not A2)) A3)))))))))))
             (not A6))))
      (not A5))
  3: (and
      (and
       (and (and A4 (not (and (and A1 (not A2)) A3)))
            (not (and A3 (not (and A1 (not A2))))))
       (not
        (and
         (and A2
              (not
               (and (and A4 (not (and (and A1 (not A2)) A3)))
                    (and A3 (not (and A1 (not A2)))))))
         (not
          (and (and A3 (not (and A1 (not A2))))
               (not (and A4 (not (and (and A1 (not A2)) A3)))))))))
      (not
       (and A8
            (not
             (and
              (and (and (and A1 (not A2)) (not A3))
                   (not
                    (and
                     (and (and A4 (not (and (and A1 (not A2)) A3)))
                          (not (and A3 (not (and A1 (not A2))))))
                     (not
                      (and
                       (and A2
                            (not
                             (and (and A4 (not (and (and A1 (not A2)) A3)))
                                  (and A3 (not (and A1 (not A2)))))))
                       (not
                        (and (and A3 (not (and A1 (not A2))))
                             (not
                              (and A4 (not (and (and A1 (not A2)) A3)))))))))))
              (not A6))))))
  4: (and
      (and
       (and (and (and A1 (not A2)) (not A3))
            (not
             (and
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (not (and A3 (not (and A1 (not A2))))))
              (not
               (and
                (and A2
                     (not
                      (and (and A4 (not (and (and A1 (not A2)) A3)))
                           (and A3 (not (and A1 (not A2)))))))
                (not
                 (and (and A3 (not (and A1 (not A2))))
                      (not (and A4 (not (and (and A1 (not A2)) A3)))))))))))
       (not A6))
      (not A8))
  5: (and
      (and
       (and (and (and A1 (not A2)) (not A3))
            (not
             (and
              (and (and A4 (not (and (and A1 (not A2)) A3)))
                   (not (and A3 (not (and A1 (not A2))))))
              (not
               (and
                (and A2
                     (not
                      (and (and A4 (not (and (and A1 (not A2)) A3)))
                           (and A3 (not (and A1 (not A2)))))))
                (not
                 (and (and A3 (not (and A1 (not A2))))
                      (not (and A4 (not (and (and A1 (not A2)) A3)))))))))))
       (not A6))
      A8)
  6: A6
  7: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (not
       (and (and A4 (not (and (and A1 (not A2)) A3)))
            (not (and A3 (not (and A1 (not A2))))))))
  8: (and
      (and
       (and A2
            (not
             (and (and A4 (not (and (and A1 (not A2)) A3)))
                  (and A3 (not (and A1 (not A2)))))))
       (not
        (and (and A3 (not (and A1 (not A2))))
             (not (and A4 (not (and (and A1 (not A2)) A3)))))))
      (and (and A4 (not (and (and A1 (not A2)) A3)))
           (not (and A3 (not (and A1 (not A2)))))))
  9: (and (and A3 (not (and A1 (not A2))))
          (not (and A4 (not (and (and A1 (not A2)) A3)))))
  10: (and (and A4 (not (and (and A1 (not A2)) A3)))
           (and A3 (not (and A1 (not A2)))))
  11: (and (and (and A1 (not A2)) A3) (not A4))
  12: (and (and (and A1 (not A2)) A3) A4)
  13: A7
)
U (
)
```

Baseline MDTD algorithm

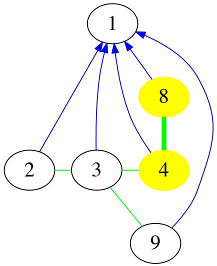
Algorithm 1: Finds the maximal disjoint type decomposition

Input: A finite non-empty set U of sets

Output: A finite set D of disjoint sets

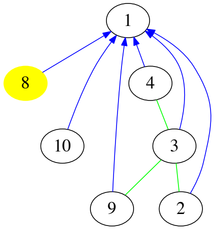
```
1  $D \leftarrow \emptyset$
2  while true do
3      $D' \leftarrow \{u \in U \mid u' \in U \setminus \{u\} \implies u \cap u' = \emptyset\}$
4      $D \leftarrow D \cup D'$
5      $U \leftarrow U \setminus D'$
6      if $U = \emptyset$ then
7          return $D$
8      else
9          Find $\alpha \in U$ and $\beta \in U$ such that $\alpha \cap \beta \neq \emptyset$
10         $U \leftarrow U \setminus \{\alpha, \beta\} \quad \cup \quad$ standard-partition
```

Step 3 using s-expressions



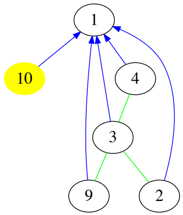
Node	Boolean	Standard
	expression	partition
1	$A_1 \cap \overline{A_5} \cap \overline{A_6}$	
2	$A_2 \cap \overline{A_4 \cap \overline{A_5}}$	
3	A_3	
4	$A_4 \cap \overline{A_5} \cap \overline{A_2}$	$\rightarrow A_4 \cap \overline{A_5} \cap \overline{A_2} \cap \overline{A_8 \cap \overline{A_5}}$
8	$A_8 \cap \overline{A_5}$	$\rightarrow A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}$
9	$A_2 \cap A_4 \cap \overline{A_5}$	
10		$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$
X_5	A_5	
X_6	A_6	
X_7	A_7	

Step 4 using s-expressions



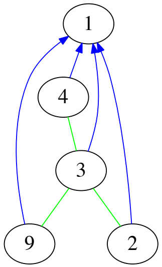
Node	Boolean expression	Standard partition
1	$A_1 \cap \overline{A_5} \cap \overline{A_6}$	$\rightarrow A_1 \cap \overline{A_5} \cap \overline{A_6}$ $\cap\ \overline{A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}}$
2	$A_2 \cap \overline{A_4 \cap \overline{A_5}}$	
3	A_3	
4	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap \overline{A_8 \cap \overline{A_5}}$	
8	$A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}$	collect
9	$A_2 \cap A_4 \cap \overline{A_5}$	
10	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$	
X_5	A_5	
X_6	A_6	
X_7	A_7	

Step 5 using s-expressions



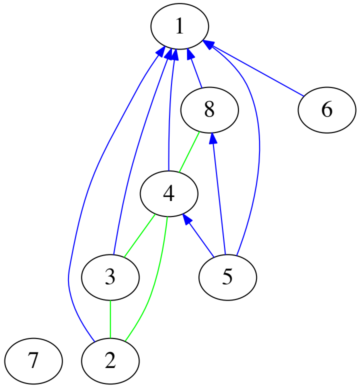
Node	Boolean expression	Standard partition
1	$A_1 \cap \overline{A_5} \cap \overline{A_6} \cap \overline{A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}}$	$\to A_1 \cap \overline{A_5} \cap \overline{A_6} \cap \overline{A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}}$
2	$A_2 \cap \overline{A_4 \cap \overline{A_5}}$	
3	A_3	
4	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap \overline{A_8 \cap \overline{A_5}}$	
9	$A_2 \cap A_4 \cap \overline{A_5}$	
10	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$	collect
X_5	A_5	
X_6	A_6	
X_7	A_7	
X_8	$A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}$	

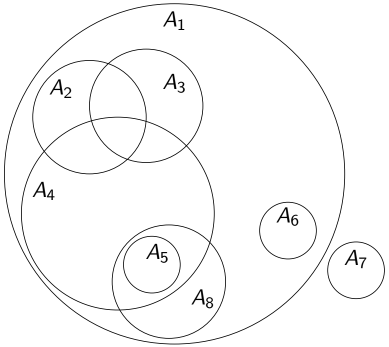
Step 6 using s-expressions



Node	Boolean expression
1	$A_1 \cap \overline{A_6}$
	$\cap \overline{A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}}$
	$\cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}}$
2	$A_2 \cap \overline{A_4 \cap \overline{A_5}}$
3	A_3
4	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap \overline{A_8 \cap \overline{A_5}}$
9	$A_2 \cap A_4 \cap \overline{A_5}$
X_5	A_5
X_6	A_6
X_7	A_7
X_8	$A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}$
X_{10}	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$

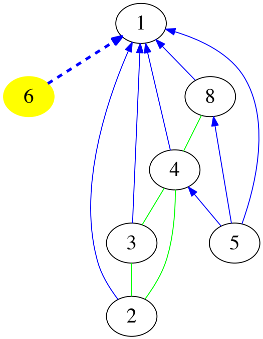
Graph-based MDTD





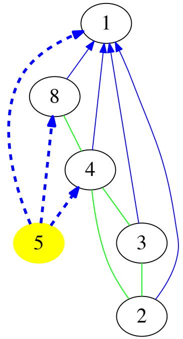
Topology graph representing type hierarchy and intersections. We find MDTD by controlled breaking and *re-wiring* of this graph.

Step 0



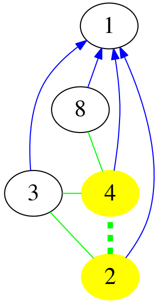
Node	Boolean expression	Standard partition
1	A_1	$\to A_1 \cap \overline{A_6}$
2	A_2	
3	A_3	
4	A_4	
5	A_5	
6	A_6	A_6 collect into D
8	A_8	
X_7	A_7	

Step 1



Node	Boolean expression	Standard partition
1	A_1	$\rightarrow A_1 \cap \overline{A_6} \cap \overline{A_5}$
2	A_2	
3	A_3	
4	A_4	$\rightarrow A_4 \cap \overline{A_5}$
5	A_5	A_5 collect into D
8	A_8	$\rightarrow A_8 \cap \overline{A_5}$
X_6	A_6	
X_7	A_7	

Step 2 using s-expressions

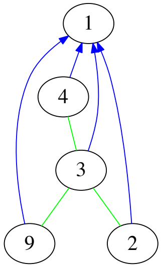


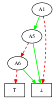
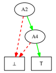
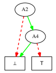
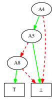
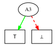
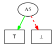
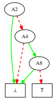
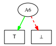
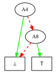
Node	Boolean expression	Standard partition
1	$A_1 \cap \overline{A_5} \cap \overline{A_6}$	
2	A_2	$\rightarrow A_2 \cap \overline{A_4 \cap \overline{A_5}}$
3	A_3	
4	$A_4 \cap \overline{A_5}$	$\rightarrow A_4 \cap \overline{A_5} \cap \overline{A_2}$
8	$A_8 \cap \overline{A_5}$	
9		$A_2 \cap A_4 \cap \overline{A_5}$
X_5	A_5	
X_6	A_6	
X_7	A_7	

Summary of MDTD algorithms

- Baseline algorithm suffers from several problems.
  - Set semantics
  - Slow loops
  - Explosive size
- Graph algorithm fixes some of these problems.
  - Better loops
  - Fewer redundant checks
- Still a problem:
  - Set semantics of type specifiers.
  - Type equivalence
  - Initial graph construction is $\Omega(n^2)$
- We can consider a smarter data structure to represent types.

After Step 2 using ROBDDs



Node	type	Node	type
1		9	
2		10	
3		X_5	
4		X_6	
8		X_7