Programmatic Manipulation of Type Specifiers
in Common Lisp

Jim Newton

10th European Lisp Symposium

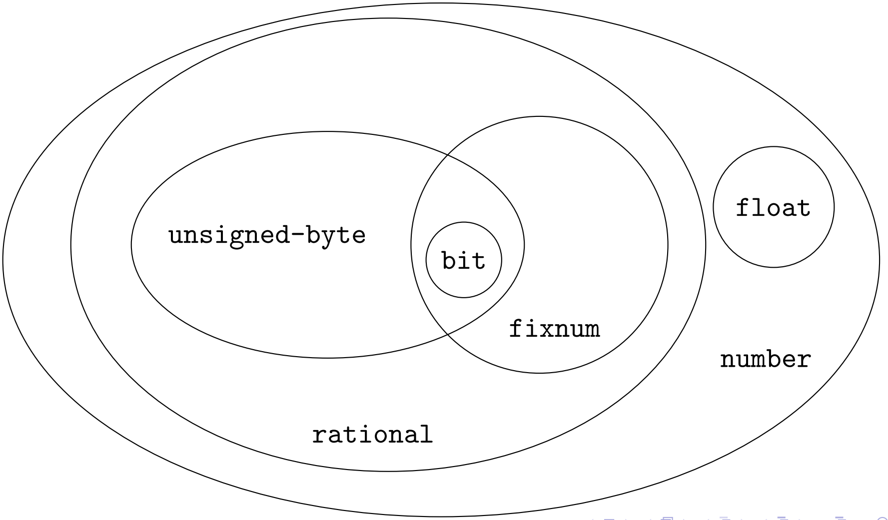
3-4 April 2017

Overview

1. Common Lisp Types
   - Native type specifiers
   - Type calculus with type specifiers
2. Reduced Ordered Binary Decision Diagrams (ROBDDs)
   - Representing CL types as ROBDDs
   - Reductions to accommodate CL subtypes
   - Type calculus using ROBDDs
   - Type checking and code generation with BDDs
3. Conclusion
   - Summary
   - Questions

Table of Contents

Types are sets. Subtypes are subsets. Intersecting types are intersecting sets. Disjoint types are disjoint sets.



Type specifiers are powerful and intuitive

Type specifiers can be extremely intuitive thanks to homoiconicity.

- Simple
  - `integer`

Type specifiers are powerful and intuitive

Type specifiers can be extremely intuitive thanks to homoiconicity.

- Simple
  - `integer`
- Compound type specifiers
  - `(satisfies oddp)`
  - `(and (or number string) (not (satisfies MY-FUN)))`

Type specifiers are powerful and intuitive

Type specifiers can be extremely intuitive thanks to homoiconicity.

- Simple
  - `integer`
- Compound type specifiers
  - `(satisfies oddp)`
  - `(and (or number string) (not (satisfies MY-FUN)))`
- Specifiers for the empty type
  - `nil`
  - `(and number string)`
  - `(and (satisfies evenp) (satisfies oddp))`

Type specifiers are powerful and intuitive

Type specifiers can be extremely intuitive thanks to homoiconicity.

- Simple
  - `integer`
- Compound type specifiers
  - `(satisfies oddp)`
  - `(and (or number string) (not (satisfies MY-FUN)))`
- Specifiers for the empty type
  - `nil`
  - `(and number string)`
  - `(and (satisfies evenp) (satisfies oddp))`

There are many type specifiers for the same type.

We can ask questions with CL type specifiers.

- Type membership? `(typep x T1)`

$$x \in T_1$$

We can ask questions with CL type specifiers.

- Type membership? `(typep x T1)`
- Type inclusion? `(subtypep T1 T2)`

$$T_1 \subset T_2$$

We can ask questions with CL type specifiers.

- Type membership? `(typep x T1)`
- Type inclusion? `(subtypep T1 T2)`
- Type equivalence? `(and (subtypep T1 T2) (subtypep T2 T1))`

$$(T_1 \subset T_2) \wedge (T_2 \subset T_1)$$

We can ask questions with CL type specifiers.

- Type membership? `(typep x T1)`
- Type inclusion? `(subtypep T1 T2)`
- Type equivalence? `(and (subtypep T1 T2) (subtypep T2 T1))`
- Type disjointness? `(subtypep ‘(and ,T1 ,T2) nil)`

$$T_1 \cap T_2 \subset \emptyset$$

We can ask questions with CL type specifiers.

- Type membership? `(typep x T1)`
- Type inclusion? `(subtypep T1 T2)`
- Type equivalence? `(and (subtypep T1 T2) (subtypep T2 T1))`
- Type disjointness? `(subtypep ‘(and ,T1 ,T2) nil)`

Sometimes, `subtypep` returns don’t know.

Type expressions can be barely human readable.

```
(setf T1 '(not (or (and fixnum unsigned-byte)
                   (and number float)
                   (and fixnum float))))

(setf T2 '(or (and fixnum
                   (not rational)
                   (or (and number (not float))
                       (not number)))
              (and (not fixnum)
                   (or (and number (not float))
                       (not rational)))))
```

Type expressions can be barely human readable.

```
(setf T1 '(not (or (and fixnum unsigned-byte)
                   (and number float)
                   (and fixnum float))))

(setf T2 '(or (and fixnum
                   (not rational)
                   (or (and number (not float))
                       (not number)))
              (and (not fixnum)
                   (or (and number (not float))
                       (not rational)))))
```

The same type may be checked multiple times.

Type expressions can be barely human readable.

```
(setf T1 '(not (or (and fixnum unsigned-byte)
                   (and number float)
                   (and fixnum float))))

(setf T2 '(or (and fixnum
                   (not rational)
                   (or (and number (not float))
                       (not number)))
              (and (not fixnum)
                   (or (and number (not float))
                       (not rational)))))
```

The same type may be checked multiple times. We can do better.

Table of Contents

Type specifier viewed as a Boolean expression of variables

A CL type specifier has a dual in Boolean algebra notation.

Type specifier: `(not (or (and A C) (and B C) (and B D)))`
Boolean Expression: $\neg\,((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$

Type specifier viewed as a Boolean expression of variables

Forget about the CL type system for the moment,
and just concentrate on Boolean algebra with binary variables.

Boolean Expression: $\neg\left((A \wedge C) \vee (B \wedge C) \vee (B \wedge D)\right)$

Type specifier viewed as a Boolean expression of variables

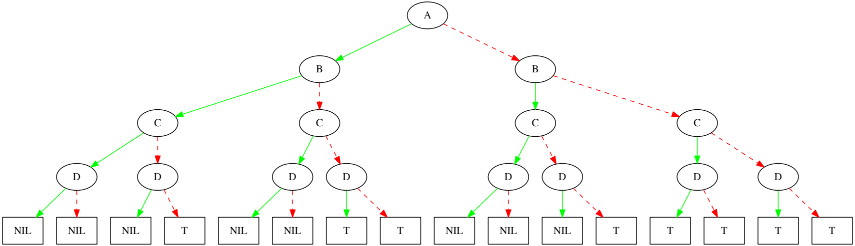
If we *order* the variables,
then every Boolean expression has a unique truth table.

Boolean Expression: $\neg((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$

Type specifier viewed as a Boolean expression of variables

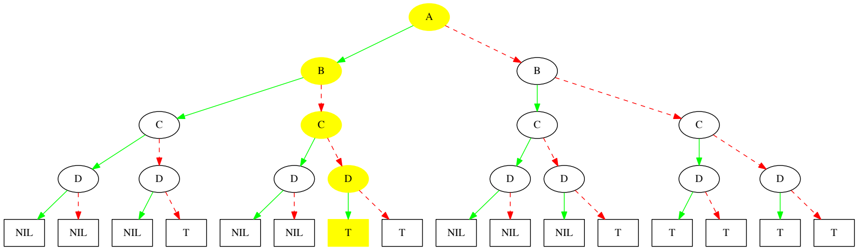
The truth table can be represented as an OBDD, ordered binary decision diagram. A green arrow a variable being true; a red arrow represents the variable being false.

Boolean Expression: $\neg\,((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$



Type specifier viewed as a Boolean expression of variables

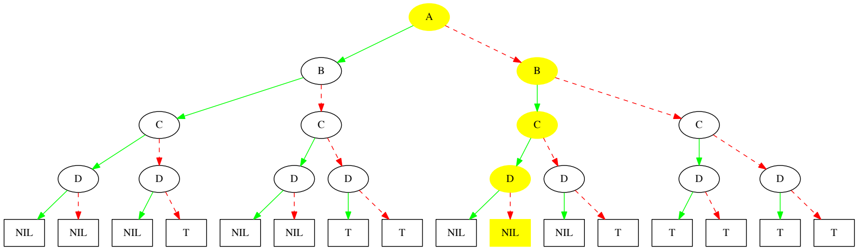
Every path from root to leaf corresponds to one row of the truth table.



A	B	C	D	$\neg((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$
$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\bot$

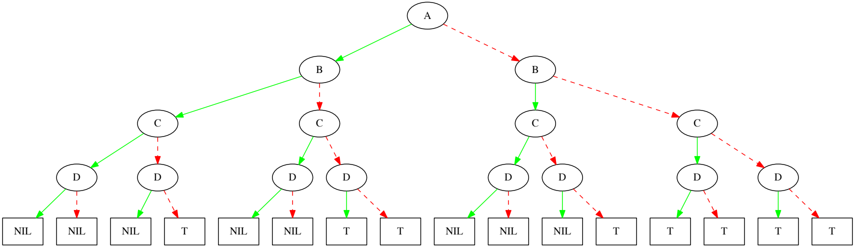
Type specifier viewed as a Boolean expression of variables

Every path from root to leaf corresponds to one row of the truth table.



A	B	C	D	$\neg((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$
$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\bot$

Type specifier viewed as a Boolean expression of variables



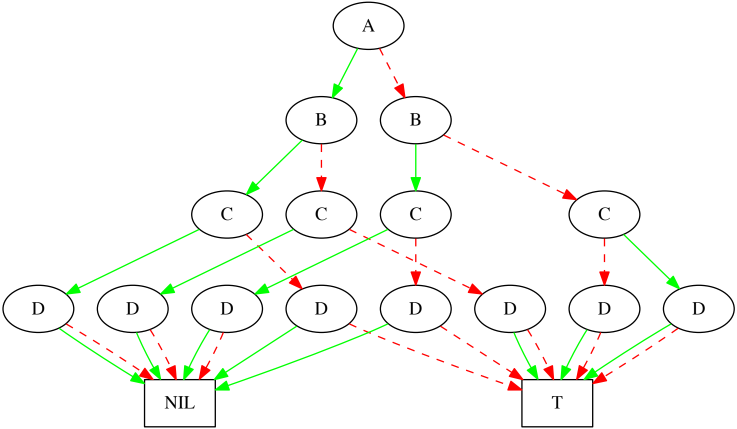
4 variables $\implies 2^{4+1} - 1 = 31$ nodes

The graph size grows exponentially with number of variables.

We can do better.

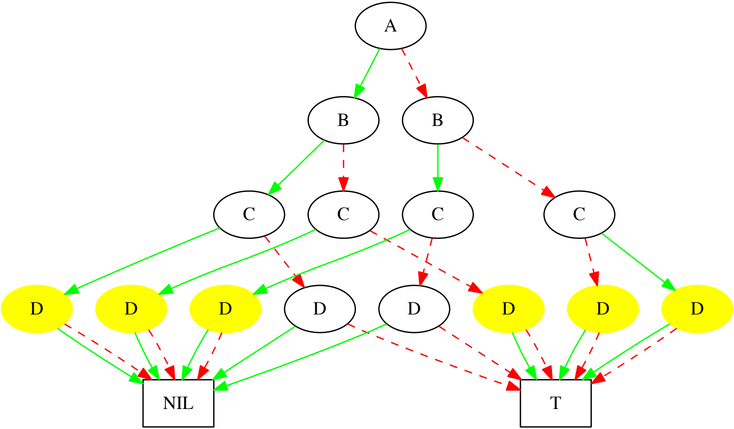
Standard Rule 1: the *terminal* rule

There are 3 standard reduction rules. The terminal rule allows us to replace leaf nodes with singleton objects, `NIL` and `T`. Divides size by 2.

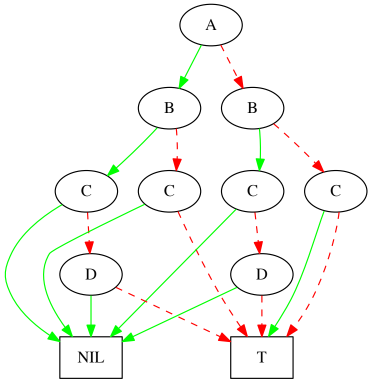


Standard Rule 2: the *deletion* rule

The deletion rule allows us to remove nodes which have the same red (false) and green (true) pointer.

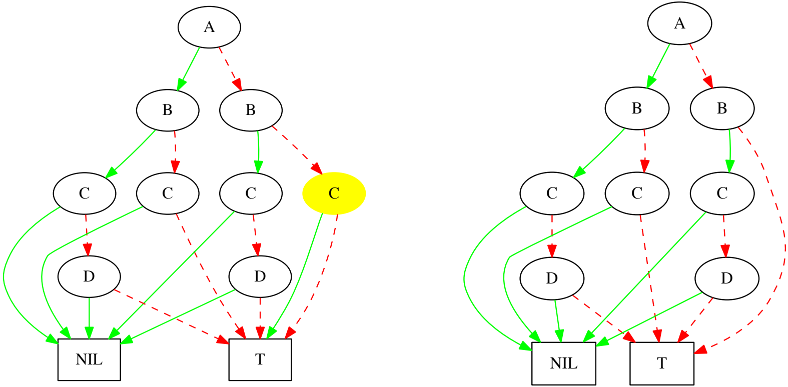


Reducing to 11 nodes



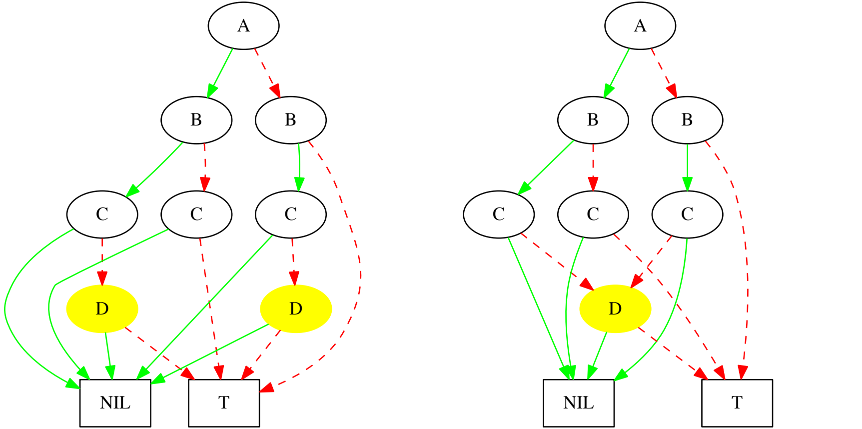
More reduction

The deletion rule can be applied multiple times.



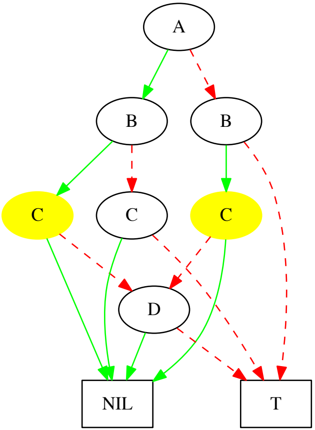
Standard Rule 3: the *merging* rule

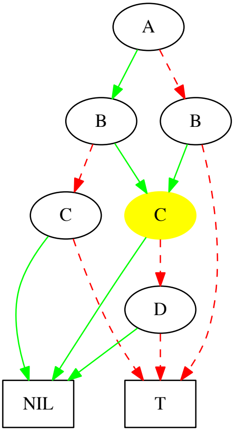
The merging rule allows us to merge structurally congruent nodes, *i.e.*, with same children, and same label.



More congruent nodes

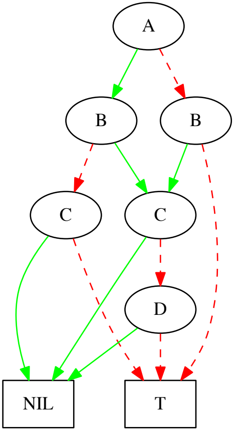
The merging rule can be applied multiple times.





ROBDD: Reduced ordered binary decision diagram

Started with 31 nodes, we can represent the CL type specifier with only 8 nodes.

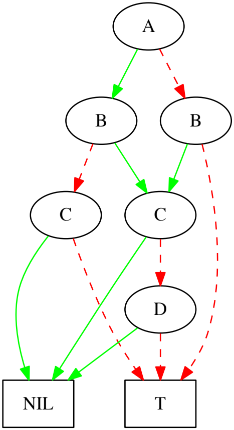



```
(not (or (and A C)
         (and B C)
         (and B D)))
```

Standard algorithm to serialize to a canonical disjunctive form.

```
(or (and A (not B) (not C))
    (and A B (not C) (not D))
    (and (not A) B (not C) (not D))
    (and (not A) (not B)))
```

ROBDD: Reduced ordered binary decision diagram

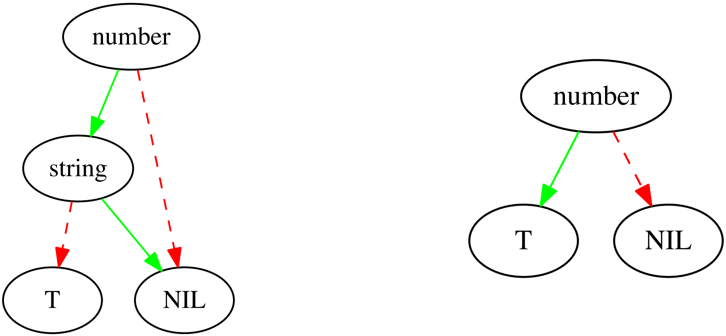



```
(not (or (and A C)
         (and B C)
         (and B D)))

(or (and A (not B) (not C))
    (and A B (not C) (not D))
    (and (not A) B (not C) (not D))
    (and (not A) (not B)))
```

This serialization is in no sense minimum in form.

Standard ROBDD reduction rules are insufficient for CL type system.

`(and number (not string))` = `number` are equivalent types, but the BDDs are different!



Brief Recap

We would like to use ORBDDs to programmatically represent and manipulate CL types.

- We have used the ORBDD developed for Boolean algebra of binary variables,

Brief Recap

We would like to use ORBDDs to programmatically represent and manipulate CL types.

- We have used the ORBDD developed for Boolean algebra of binary variables,
- Applying: the (1) terminal rule, (2) deletion rule, and (3) merging rule.

Brief Recap

We would like to use ORBDDs to programmatically represent and manipulate CL types.

- We have used the ORBDD developed for Boolean algebra of binary variables,
- Applying: the (1) terminal rule, (2) deletion rule, and (3) merging rule.
- We unfortunately lack unique ORBDD representations for equivalent CL types.

Brief Recap

We would like to use ORBDDs to programmatically represent and manipulate CL types.

- We have used the ORBDD developed for Boolean algebra of binary variables,
- Applying: the (1) terminal rule, (2) deletion rule, and (3) merging rule.
- We unfortunately lack unique ORBDD representations for equivalent CL types.
- We find that it does not quite work for reasoning about CL types.

Brief Recap

We would like to use ORBDDs to programmatically represent and manipulate CL types.

- We have used the ORBDD developed for Boolean algebra of binary variables,
- Applying: the (1) terminal rule, (2) deletion rule, and (3) merging rule.
- We unfortunately lack unique ORBDD representations for equivalent CL types.
- We find that it does not quite work for reasoning about CL types.
- A solution is needed.

Brief Recap

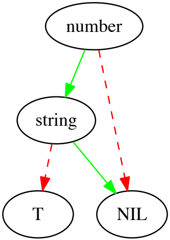
We would like to use ORBDDs to programmatically represent and manipulate CL types.

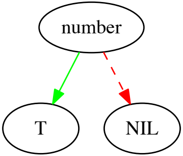
- We have used the ORBDD developed for Boolean algebra of binary variables,
- Applying: the (1) terminal rule, (2) deletion rule, and (3) merging rule.
- We unfortunately lack unique ORBDD representations for equivalent CL types.
- We find that it does not quite work for reasoning about CL types.
- A solution is needed.
- We introduce a 4th reduction rule: the *subtype rule*. Our contribution.

Subtype rule (4), CL type system compatibility

The types `number` and `string` are disjoint, therefore, $string \subset \overline{number}$.

Child to search	Relation	Reduction
$P.green$	$P \subset C$	$C \to C.green$
$P.green$	$P \subset \overline{C}$	$C \to C.red$
$P.red$	$\overline{P} \subset C$	$C \to C.green$
$P.red$	$\overline{P} \subset \overline{C}$	$C \to C.red$
$P.red$	$P \supset C$	$C \to C.red$
$P.red$	$P \supset \overline{C}$	$C \to C.green$
$P.green$	$\overline{P} \supset C$	$C \to C.red$
$P.green$	$\overline{P} \supset \overline{C}$	$C \to C.green$

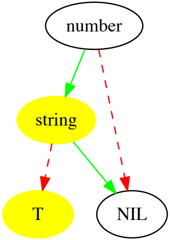


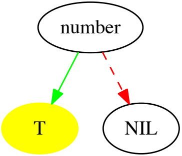


Subtype rule (4), CL type system compatibility

The types `number` and `string` are disjoint; therefore, $string \subset \overline{number}$.

Child to search	Relation	Reduction
$P.green$	$P \subset C$	$C \to C.green$
$P.green$	$P \subset \overline{C}$	$C \to C.red$
$P.red$	$\overline{P} \subset C$	$C \to C.green$
$P.red$	$\overline{P} \subset \overline{C}$	$C \to C.red$
$P.red$	$P \supset C$	$C \to C.red$
$P.red$	$P \supset \overline{C}$	$C \to C.green$
$number.green$	$\overline{number} \supset string$	$string \to string.red$
$P.green$	$\overline{P} \supset \overline{C}$	$C \to C.green$





Type calculus using ROBDDs

As before, we can ask questions with ROBDDs.

Type calculus using ROBDDs

As before, we can ask questions with ROBDDs.

Questions

Are two types the same? Or disjoint? Or is one a subtype of the other?

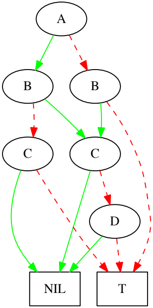
Functions

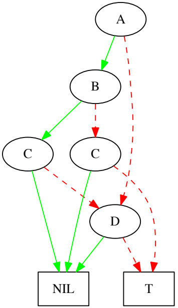
`bdd-and`, `bdd-or`, `bdd-and-not`.

Are the two types the same? No, BDDs are different.

```
(not (or (and A C)
         (and B C)
         (and B D)))
```

```
(or (and A
         (not C)
         (or (and B (not D))
             (not B)))
    (and (not A)
         (or (and B (not C) (not D))
             (not D))))
```

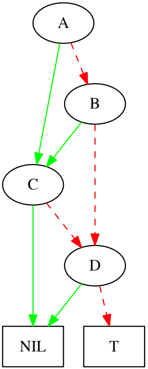




Are two types disjoint? No, the intersection is non-nil.

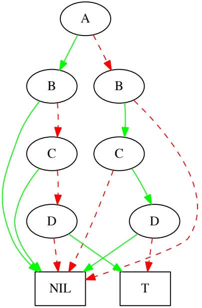
```
(setf T1
  (bdd '(and (not (and (not A) D))
             (not (or (and A C)
                      (and B C)
                      (and B D))))))

(setf T2
   (bdd '(or (and A
                  (not C)
                  (or (and B (not D))
                      (not B)))
             (and (not A)
                  (or (and B
                           (not C)
                           (not D))
                      (not D))))))
```

(bdd−and T1 T2)



Is one a subtype of the other? Yes. $T_1 \subset T_2$.

```
(bdd-and-not T2 T1)
```




```
(bdd-and-not T1 T2)
```

Run-time calls to bdd-type-p

```
(defun bdd-type-p (obj bdd)
  (etypecase bdd
    (bdd-false
     nil)
    (bdd-true
     t)
    (bdd-node
     (bdd-type-p obj
                 (if (typep obj (bdd-label bdd))
                     (bdd-left bdd)
                     (bdd-right bdd))))))
```

Guarantees that each base-type is checked maximum of once.

Compile time call to bdd-typep, via compiler-macro

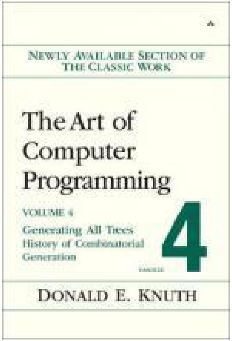
```
(bdd-typep X '(or (and sequence (not array))
                  number
                  (and (not sequence) array)))
```

Compile time call to bdd-typep, via compiler-macro

```
(bdd-typep X '(or (and sequence (not array))
                 number
                 (and (not sequence) array)))

(funcall (lambda (obj)
           (block nil
             (tagbody
              1 (if (typep obj 'array)
                    (go 2)
                    (go 3))
              2 (return (not (typep obj 'sequence)))
              3 (if (typep obj 'number)
                    (return t)
                    (go 4))
              4 (return (typep obj 'sequence)))))
         X)
```

Table of Contents

Donald Knuth's new toy.



Binary decision diagrams (BDDs) 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*]

Summary

- Native CL type specifiers are
  - Powerful and intuitive
  - But may suffer performance issues
  - Missing capability (subtypep)
- ROBDDs offer an interesting alternative
  - We have extended Standard ROBDD theory to CL types
  - Shown type calculus operations, equality, intersection, relative complement, etc
  - Demonstrated efficient compile time code generation for type checking.
  - Competitive performance
- Lots more work to do.
- For more information see the LRDE website:
  - https://www.lrde.epita.fr/wiki/User:Jnewton

Questions/Answers

Questions?

ROBDD: Reduced Ordered Binary Design Diagrams

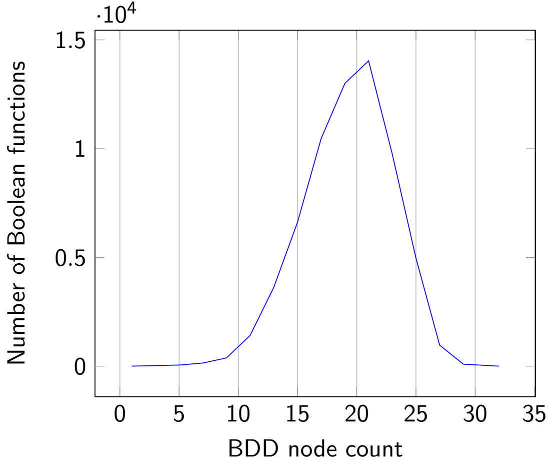
Having as few nodes as possible has advantages in:

- Correctness in presence of subtypes,
- Memory allocation,
- Execution time of graph-traversal related operations, and
- Generated code size (as we'll see later).

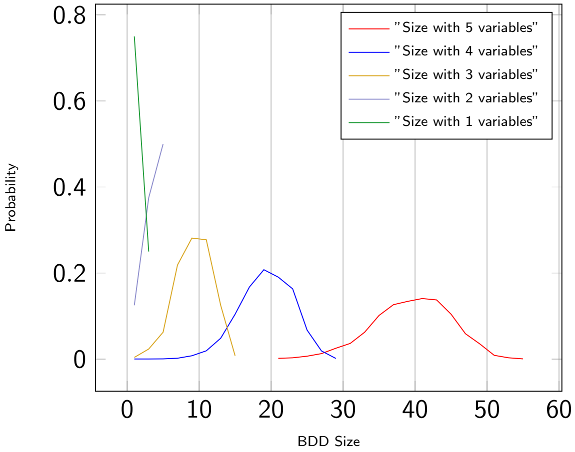
Possible ROBDD sizes for 4 variables

Of the $2^{2^4} = 65,536$ different Boolean functions of 4 variables, various sizes of reduced BDDs are possible.

Worst case size is 32 nodes. Average size is approximately 20 nodes.

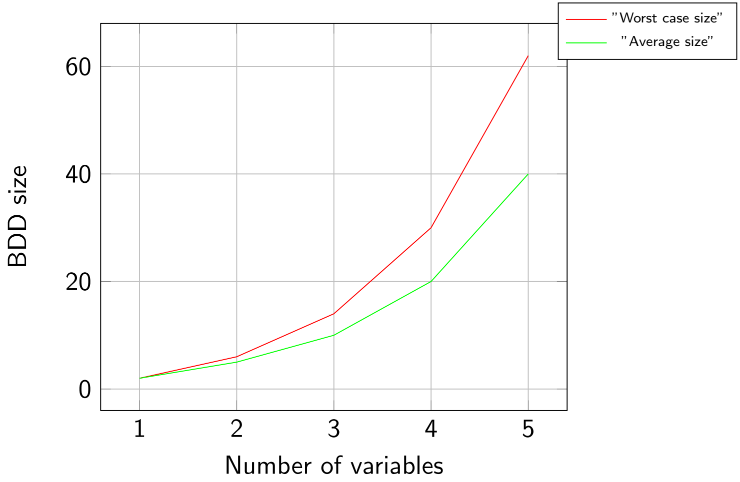


Distributions for 2 to 5 variables



Distribution of ROBDD size over all possible Boolean functions of N variables.

Expected and worst case ROBDD size



FIRST TRY: Expands to the following. $\mathcal{O}(2^n)$ code size. $\mathcal{O}(n)$ execution time.

If the type specifier is known at compile time.

```
(funcall (lambda (obj)
           (if (typep obj 'array)
               (if (typep obj 'sequence)
                   nil
                   t)
               (if (typep obj 'number)
                   t
                   (if (typep obj 'sequence)
                       t
                       nil))))
         X)
```

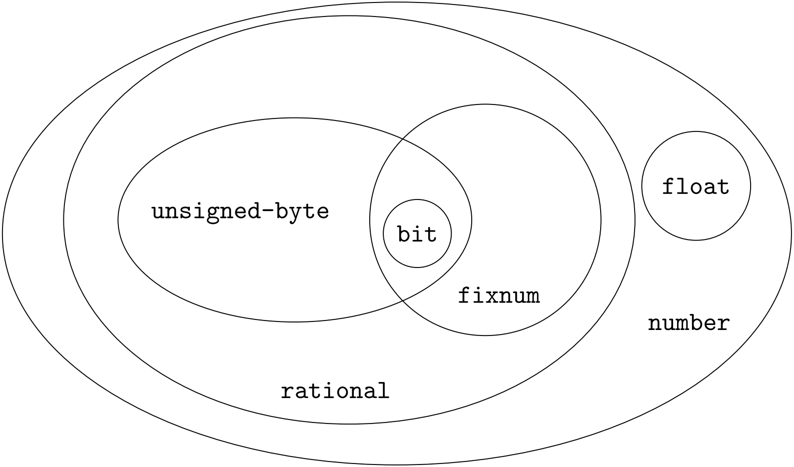
We can do better.

BETTER: $\mathcal{O}(2^{\frac{n}{2}})$ code size. $\mathcal{O}(n)$ execution time.[1]

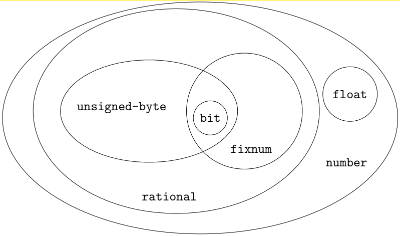
```
(funcall (lambda (obj)
           (labels ((#:f1 ()
                       (typep obj 'sequence))
                    (#:f2 ()
                       (or (typep obj 'number)
                           (#:f1)))
                    (#:f3 ()
                       (not (typep obj 'sequence)))
                    (#:f4 ()
                       (if (typep obj 'array)
                           (#:f3)
                           (#:f2))))
             (#:f4)))
         X)
```

[1] $\mathcal{O}(2^{\frac{n}{2}})$ is a non-rigorous estimate.

Experimental problem: thoroughly partition a set of types

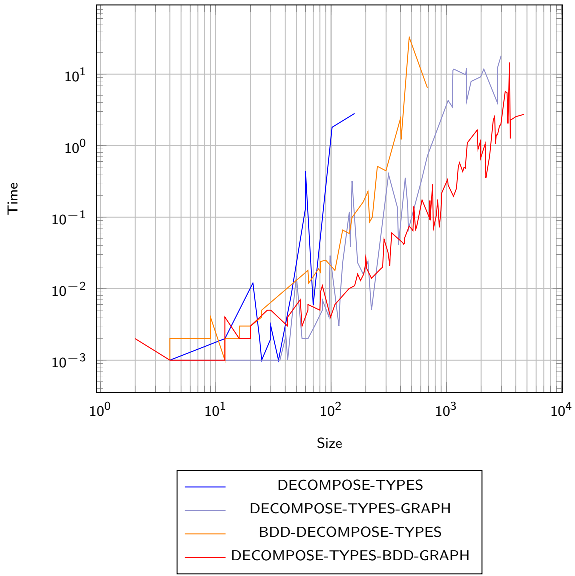


Maximal Disjoint Type Decomposition

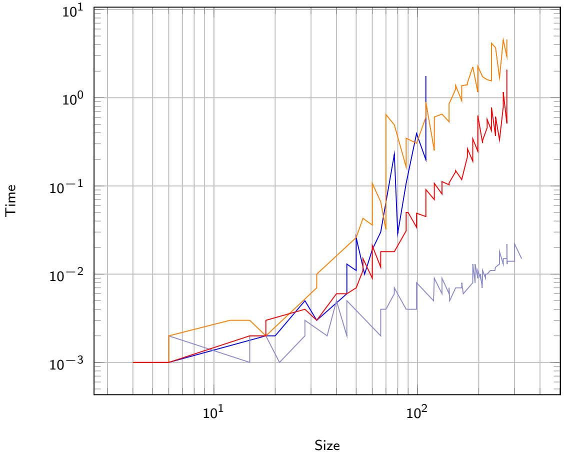



```
(bit float fixnum number rational unsigned-byte)
-->
   (bit
    float
    (and fixnum unsigned-byte (not bit))
    (and fixnum (not unsigned-byte))
    (and number (not float) (not rational))
    (and rational (not fixnum) (not unsigned-byte))
    (and unsigned-byte (not fixnum)))
```

Combinations of number and condition



Subtypes of fixnum: (member ...)

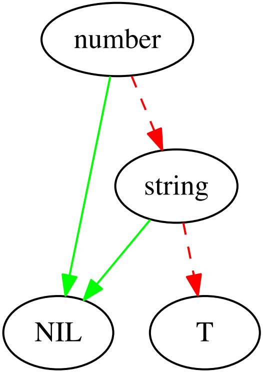


Type specifier summary

- Easy and intuitive (thanks to homoiconicity)
- Run-time calls to `subtypep` and `typep`
- Issues of performance and correctness of `subtypep` and `typep`

Subtypes

```
(and (not number)
     (not string))
```



Caveat of subtypep

Sometimes `subtypep` returns *don't know*. Sometimes for good reasons. Sometimes not.

```
CL-USER> (subtypep '(satisfies oddp) '(satisfies evenp))
> NIL, NIL

CL-USER> (subtypep 'arithmetic-error '(not cell-error))
> NIL, NIL
```