Strategies for typecase optimization

Jim Newton

11th European Lisp Symposium

16-17 April 2017

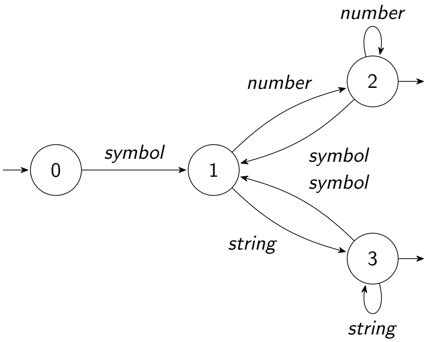
Table of Contents

Background

- Rational Type Expressions (RTE) recognize sequences based on element type.
- Code gen for RTE: excessive use of `typecase` with complex, machine generated type specifiers.

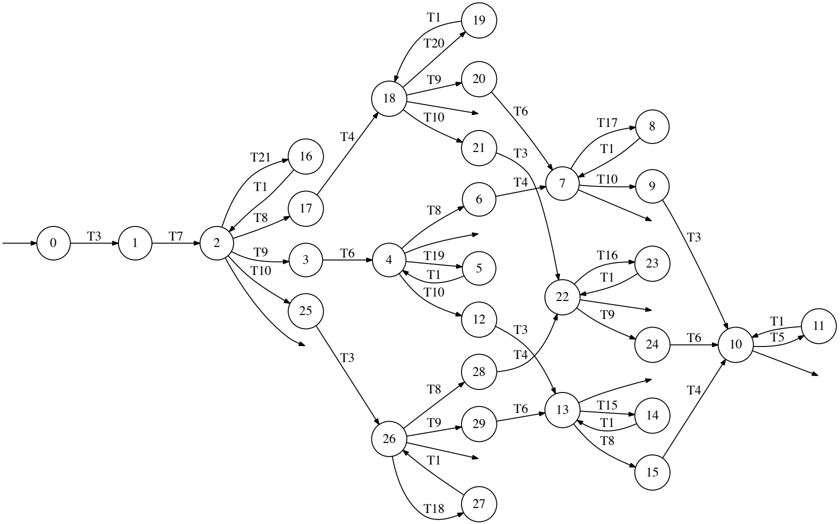
Code generated from RTE state machine

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




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

More complicated State Machine



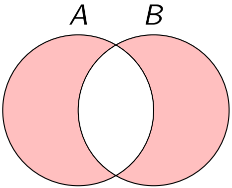
Background

- Problem: how to order the type specifiers and minimize redundancy.
- Two approaches
  1. S-expression manipulation and heuristics.
  2. Binary Decision Diagrams (BDD)
- Original hope was that the BDD approach would be superior.
- I now believe both approaches have merits.

Table of Contents

What is a Common Lisp type?

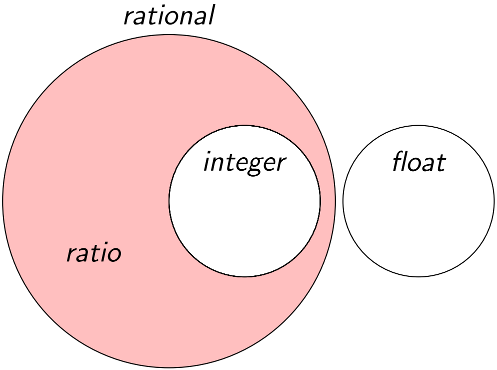
A type is a set of Lisp objects. Type operations are set operations.



- Subtypes are subsets.
- Intersecting types are intersecting sets.
- Disjoint types are disjoint sets.
- The empty type is the empty set.

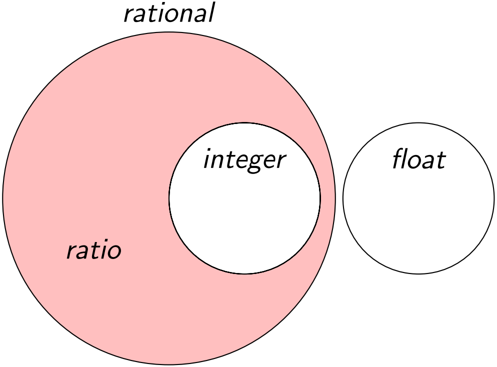
Some types can be identified Boolean operations

- $integer \subset rational$
- $ratio = rational \cap \overline{integer}$ `ratio = (and rational (not integer))`



Some types can be identified Boolean operations

- $integer \subset rational$
- $ratio = rational \cap \overline{integer}$ ratio `= (and rational (not integer))`
- $float \subset \overline{rational}$
- $\emptyset = rational \cap float$ `nil = (and rational float)`



What is typecase ?

- Simple example of typecase

```
(typecase expr
  (fixnum body-forms-1 ...)
  (number body-forms-2 ...)
  (string body-forms-3 ...))
```

What is typecase ?

- Simple example of typecase

```
(typecase expr
  (fixnum body-forms-1 ...)
  (number body-forms-2 ...)
  (string body-forms-3 ...))
```

- typecase may use any valid type specifier.

```
(typecase expr
  ((and fixnum (not (eql 0))) body-forms-1 ...)
  ((or fixnum string)         body-forms-2 ...)
  ((member -1 -2)             body-forms-3 ...)
  ((satisfies MY-FUN)         body-forms-4 ...)
  ...)
```

What is typecase ?

- Simple example of typecase

```
(typecase expr
  (fixnum body-forms-1 ...)
  (number body-forms-2 ...)
  (string body-forms-3 ...))
```

- typecase may use any valid type specifier.

```
(typecase expr
  ((and fixnum (not (eql 0))) body-forms-1 ...)
  ((or fixnum string)         body-forms-2 ...)
  ((member -1 -2)             body-forms-3 ...)
  ((satisfies MY-FUN)         body-forms-4 ...)
  ...)
```

- Rich built-in syntax for specifying lots of exotic types

Table of Contents

Macro expansion of `typecase`

- We can use `macroexpand-1` from SBCL.

```
(typecase x
  ((and fixnum (not (eql 0))) (f1))
  ((eql 0)                    (f2))
  (symbol                     (f3))
  (t                          (f4)))
```

Macro expansion of `typecase`

- We can use `macroexpand-1` from SBCL.

```
(typecase x
  ((and fixnum (not (eql 0))) (f1))
  ((eql 0)                    (f2))
  (symbol                     (f3))
  (t                          (f4)))
```

- The expansion *essentially* involves `cond` and `typep`.

```
(cond ((typep x '(and fixnum (not (eql 0))))
        (f1))
      ((typep x '(eql 0))
        (f2))
      ((typep x 'symbol)
        (f3))
      (t
        (f4)))
```

Issues we wish to address

- Redundant type checks
- Unreachable code
- Exhaustiveness

Redundant type checks

- Redundant type checks
- Unreachable code
- Exhaustiveness

```
(typecase obj
  ((and fixnum (not bignum))          (f1))
  ((and bignum (not unsigned-byte))   (f2))
  (bignum                             (f3)))
```

- The type check for `bignum` might be executed multiple times. Perhaps not an enormous problem...

Redundant type checks

- Redundant type checks
- Unreachable code
- Exhaustiveness

```
(typecase obj
  ((and fixnum (not bignum))          (f1))
  ((and bignum (not unsigned-byte)) (f2))
  (bignum                             (f3)))
```

- The type check for `bignum` might be executed multiple times. Perhaps not an enormous problem...
- But `satisfies` types and consequently user defined types may be arbitrarily complex.

Redundant type checks

- Redundant type checks
- Unreachable code
- Exhaustiveness

```
(typecase obj
  ((and fixnum (not bignum))           (f1))
  ((and bignum (not unsigned-byte))    (f2))
  (bignum                              (f3)))
```

- The type check for `bignum` might be executed multiple times. Perhaps not an enormous problem...
- But `satisfies` types and consequently user defined types may be arbitrarily complex.
- Especially in machine generated code.

Unreachable code

- Redundant type checks
- Unreachable code
- Exhaustiveness

```
(typecase obj
  ((or number string symbol)                                  (f1))
  ((and (satisfies slow-predicate) number)                    (f2))
  ((and (satisfies slow-predicate) (or symbol string)) (f3)))
```

- The function calls, `(f2)` and `(f3)`, are unreachable.

Unreachable code

- Redundant type checks
- Unreachable code
- Exhaustiveness

```
(typecase obj
  ((or number string symbol)                                    (f1))
  ((and (satisfies slow-predicate) number)                      (f2))
  ((and (satisfies slow-predicate) (or symbol string))          (f3)))
```

- The function calls, `(f2)` and `(f3)`, are unreachable.
- Perhaps programmer error

Unreachable code

- Redundant type checks
- Unreachable code
- Exhaustiveness

```
(typecase obj
  ((or number string symbol)                          (f1))
  ((and (satisfies slow-predicate) number)            (f2))
  ((and (satisfies slow-predicate) (or symbol string)) (f3)))
```

- The function calls, `(f2)` and `(f3)`, are unreachable.
- Perhaps programmer error
- However, your lisp compiler might not warn.

Exhaustiveness

- Redundant type checks
- Unreachable code
- Exhaustiveness

```
(typecase obj
  ((not (or number symbol)) (f1))
  (number                   (f2))
  (symbol                   (f3)))
```

The final `symbol` check is unnecessary, can be replaced with `T`.

```
(typecase obj
  ((not (or number symbol)) (f1))
  (number                   (f2))
  (t                        (f3)))
```

Issues

- Redundant type checks
- Unreachable code
- Exhaustiveness

How to address these issues?

Introducing: rewriting/forward-substitution/simplification according to heuristics.

Forward substitution

If line 3 is reached, then we know that `(or number string symbol)` failed.

```
1: (typecase obj
2:   ((or number string symbol) (f1))
3:   ((and (satisfies p1) number) (f2))
4:   ((and (satisfies p1) (or symbol string)) (f3)))
```

Forward substitution:

- *number* ← *nil*
- *string* ← *nil*
- *symbol* ← *nil*

```
1: (typecase obj
2:   ((or number string symbol) (f1))
3:   ((and (satisfies p1) nil) (f2))
4:   ((and (satisfies p1) (or nil nil)) (f3)))
```

… and Simplification

- Forward substitution results expression which can be simplified.

```
(typecase obj
  ((or number string symbol) (f1))
  ((and (satisfies p1) nil) (f2))
  ((and (satisfies p1) (or nil nil)) (f3)))
```

... and Simplification

- Forward substitution results expression which can be simplified.

```
1: (typecase obj
2:   ((or number string symbol) (f1))
3:   ((and (satisfies p1) nil) (f2))
4:   ((and (satisfies p1) (or nil nil)) (f3)))
```

- After *simplification* via `type-simplify`

```
1: (typecase obj
2:   ((or number string symbol) (f1))
3:   (nil (f2))    ; unreachable code detected
4:   (nil (f3)))   ; unreachable code detected
```

... and Simplification

- Forward substitution results expression which can be simplified.

```
1: (typecase obj
2:   ((or number string symbol) (f1))
3:   ((and (satisfies p1) nil) (f2))
4:   ((and (satisfies p1) (or nil nil)) (f3)))
```

- After *simplification* via `type-simplify`

```
1: (typecase obj
2:   ((or number string symbol) (f1))
3:   (nil (f2))    ; unreachable code detected
4:   (nil (f3)))   ; unreachable code detected
```

- Your compiler will warn about unreachable code.

Order dependent clauses

- Semantics of `typecase` depends on order of clauses. E.g., obj=2

```
(typecase obj
  (number (f1))
  (fixnum (f2)) ; f2 unreachable
  (t      (f3)))
```

vs.

```
(typecase obj
  (fixnum (f2)) ; f2 reachable
  (number (f1))
  (t      (f3)))
```

Order dependent clauses

- Semantics of `typecase` depends on order of clauses. E.g., obj=2

```
(typecase obj
  (number (f1))
  (fixnum (f2)) ; f2 unreachable
  (t      (f3)))
```

vs.

```
(typecase obj
  (fixnum (f2)) ; f2 reachable
  (number (f1))
  (t      (f3)))
```

- Unreachable code, but forward substitution does not find it.

Order dependent clauses

- Semantics of `typecase` depends on order of clauses. E.g., obj=2

```
(typecase obj
  (number (f1))
  (fixnum (f2)) ; f2 unreachable
  (t      (f3)))
```

vs.

```
(typecase obj
  (fixnum (f2)) ; f2 reachable
  (number (f1))
  (t      (f3)))
```

- Unreachable code, but forward substitution does not find it.
- `(f2)` unreachable because *fixnum* $\subset$ *number*

Rewriting

- But, we can rewrite the type checks...

```
(typecase obj
  (number (f1))
  (fixnum (f2))
  (t      (f3)))
```

Rewriting

- But, we can rewrite the type checks...

```
1: (typecase obj
2:    (number (f1))
3:    (fixnum (f2))
4:    (t      (f3)))
```

- ... to make previous failed clauses explicit.

```
1: (typecase obj
2:    (number                                  (f1))
3:    ((and fixnum (not number))               (f2))
4:    ((and t (not (or number fixnum)))        (f3)))
```

Rewriting

- But, we can rewrite the type checks...

```
1: (typecase obj
2:    (number (f1))
3:    (fixnum (f2))
4:    (t      (f3)))
```

- ... to make previous failed clauses explicit.

```
1: (typecase obj
2:    (number                                (f1))
3:    ((and fixnum (not number))             (f2))
4:    ((and t (not (or number fixnum)))      (f3)))
```

- Simplify to find unreachable code (intersection of disjoint sets).

```
1: (typecase obj
2:    (number       (f1))
3:    (nil          (f2)) ;; unreachable
4:    ((not number) (f3)))
```

Rewriting

- But, we can rewrite the type checks...

```
1: (typecase obj
2:   (number (f1))
3:   (fixnum (f2))
4:   (t      (f3)))
```

- ... to make previous failed clauses explicit.

```
1: (typecase obj
2:   (number                               (f1))
3:   ((and fixnum (not number))            (f2))
4:   ((and t (not (or number fixnum)))     (f3)))
```

- Simplify to find unreachable code (intersection of disjoint sets).

```
1: (typecase obj
2:   (number        (f1))
3:   (nil           (f2)) ;; unreachable
4:   ((not number)  (f3)))
```

- Moreover, the clauses can be reordered.

`auto-permute-typecase` macro

- Clauses can be reordered after rewriting, maintaining semantics.

`auto-permute-typecase` macro

- Clauses can be reordered after rewriting, maintaining semantics.
- Result of simplification depends on order of clauses.

`auto-permute-typecase` macro

- Clauses can be reordered after rewriting, maintaining semantics.
- Result of simplification depends on order of clauses.
- Using a `heuristic-cost` function we can compare semantically equivalent expansions.

auto-permute-typecase macro

- Clauses can be reordered after rewriting, maintaining semantics.
- Result of simplification depends on order of clauses.
- Using a heuristic-cost function we can compare semantically equivalent expansions.
- Implementation of auto-permute-typecase macro.

```
(defmacro auto-permute-typecase (obj &rest clauses)
  (let ((best-order (heuristic-cost clauses))
        (clauses (simplify (rewrite clauses))))
    (map-permutations (perm clauses)
      (let ((candidate (simplify (forward-substitute perm))))
        (when (< (heuristic-cost candidate)
                 (heuristic-cost best-order))
          (setf best-order candidate))))
    (list* 'typecase obj best-order)))
```

auto-permute-typecase macro

- Clauses can be reordered after rewriting, maintaining semantics.
- Result of simplification depends on order of clauses.
- Using a heuristic-cost function we can compare semantically equivalent expansions.
- Implementation of auto-permute-typecase macro.

```
(defmacro auto-permute-typecase (obj &rest clauses)
  (let ((best-order (heuristic-cost clauses))
        (clauses (simplify (rewrite clauses))))
    (map-permutations (perm clauses)
      (let ((candidate (simplify (forward-substitute perm))))
        (when (< (heuristic-cost candidate)
                 (heuristic-cost best-order))
          (setf best-order candidate))))
    (list* 'typecase obj best-order)))
```

- Finds permutation of clauses with minimum cost

Putting it together with `auto-permute-typecase`

Macro expansion example of `auto-permute-typecase`

```
(auto-permute-typecase obj
  ((or bignum unsigned-byte)      (f1))
  (string                         (f2))
  (fixnum                         (f3))
  ((or (not string) (not number)) (f4)))
```

Macro expansion example of `auto-permute-typecase`

```
(typecase obj
  (string                    (f2))
  ((or bignum unsigned-byte) (f1))
  (fixnum                    (f3))
  (t                         (f4)))
```

Table of Contents

Re-ordering sometimes fails to eliminate redundancy

- Sometimes no re-ordering of the `typecase` allows simplification.

```
(typecase obj
  ((and unsigned-byte (not bignum))
   body-forms-1 ...)
  ((and bignum (not unsigned-byte))
   body-forms-2 ...))
```

Re-ordering sometimes fails to eliminate redundancy

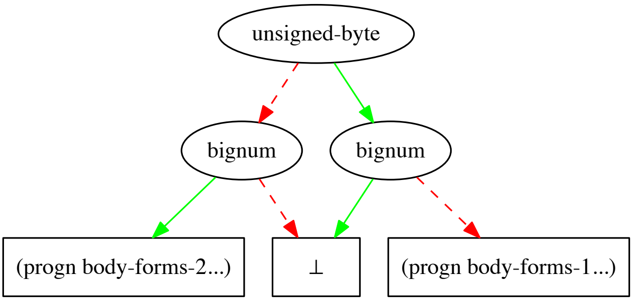
- Sometimes no re-ordering of the typecase allows simplification.

```
(typecase obj
  ((and unsigned-byte (not bignum))
   body-forms-1 ...)
  ((and bignum (not unsigned-byte))
   body-forms-2 ...))
```

- Consider expanding typecase to if/then/else

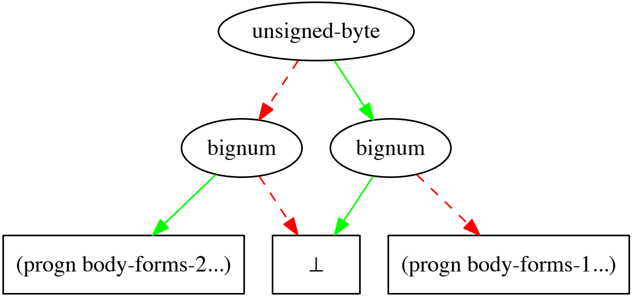
```
(if (typep obj 'unsigned-byte)
    (if (typep obj 'bignum)
        nil
        (progn body-forms-1 ...))
    (if (typep obj 'bignum)
        (progn body-forms-2 ...)
        nil))
```

Decision Diagram representing irreducible `typecase`



- This *code* flow diagram represents the calculation we want.

Decision Diagram representing irreducible `typecase`

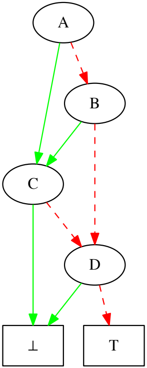


- This *code* flow diagram represents the calculation we want.
- It is *similar* to an ROBDD.

What is an ROBDD?

Reduced Ordered Binary Decision Diagram, a data structure for representing an manipulating Boolean expressions.

- Using Boolean algebra notation
  $A\overline{C}\ \overline{D} + \overline{A}B\overline{C}\ \overline{D} + \overline{A}\ \overline{B}\ \overline{D}$

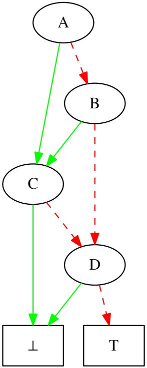


What is an ROBDD?

Reduced Ordered Binary Decision Diagram, a data structure for representing an manipulating Boolean expressions.

- Using Boolean algebra notation
  $A\overline{C}\ \overline{D} + \overline{A}B\overline{C}\ \overline{D} + \overline{A}\ \overline{B}\ \overline{D}$
- Using Common Lisp type specifier notation

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



Type specifier as ROBDD

CL-ROBDD

- Can create and manipulate ROBDDs which correspond to Common Lisp type specifiers.

Type specifier as ROBDD

CL-ROBDD

- Can create and manipulate ROBDDs which correspond to Common Lisp type specifiers.
- Adapted to accommodate subtype relations.

Type specifier as ROBDD

CL-ROBDD

- Can create and manipulate ROBDDs which correspond to Common Lisp type specifiers.
- Adapted to accommodate subtype relations.
- Can serialize such ROBDDs to efficient Common Lisp code.

Type specifier as ROBDD

CL-ROBDD

- Can create and manipulate ROBDDs which correspond to Common Lisp type specifiers.
- Adapted to accommodate subtype relations.
- Can serialize such ROBDDs to efficient Common Lisp code.
- Question: Can we convert `typecase` into a type specifier?

Type specifier as ROBDD

CL-ROBDD

- Can create and manipulate ROBDDs which correspond to Common Lisp type specifiers.
- Adapted to accommodate subtype relations.
- Can serialize such ROBDDs to efficient Common Lisp code.
- Question: Can we convert `typecase` into a type specifier?
- Answer: Yes.

Transform body-forms into predicates

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

```
(typecase obj
  (T.1 body-forms-1...)
  (T.2 body-forms-2...)
  ...
  (T.n body-forms-n...))
```

Transform body-forms into predicates

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

```
(typecase obj
  (T.1 body-forms-1...)
  (T.2 body-forms-2...)
  ...
  (T.n body-forms-n...))
```

- Encapsulate body-forms into named predicate functions.

$P_1 \leftarrow$ `(encapsulate-as-predicate body-forms-1...)`
$P_2 \leftarrow$ `(encapsulate-as-predicate body-forms-2...)`
...
$P_n \leftarrow$ `(encapsulate-as-predicate body-forms-n...)`

Transform `typecase` into type specifier

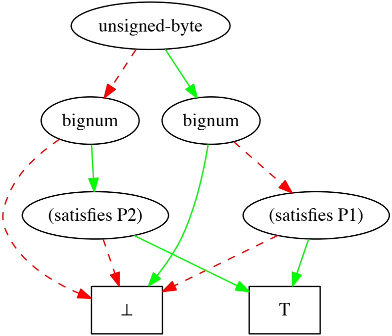
```
(typecase obj
  (T.1 body-forms-1...)
  (T.2 body-forms-2...)
  ...
  (T.n body-forms-n...))
```

Convert `typecase` to disjunctive normal form (DNF).

```
(or (and T.1
         (satisfies P1))
    (and T.2 (not T.1)
         (satisfies P2))
    ...
    (and T.n (not (or T.1 T.2 ... T.n-1))
         (satisfies Pn)))
```

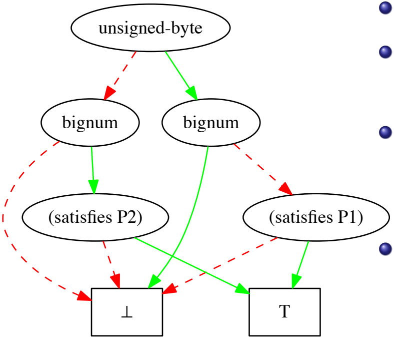
ROBDD with temporary valid `satisfies` types

```
(bdd-typecase obj
  ((and unsigned-byte
        (not bignum))
   body-forms-1 ...)
  ((and bignum
       (not unsigned-byte))
   body-forms-2 ...))
```



Now we can represent a *difficult* `typecase` as an ROBDD.

Advantages of ROBDD representation of `typecase`

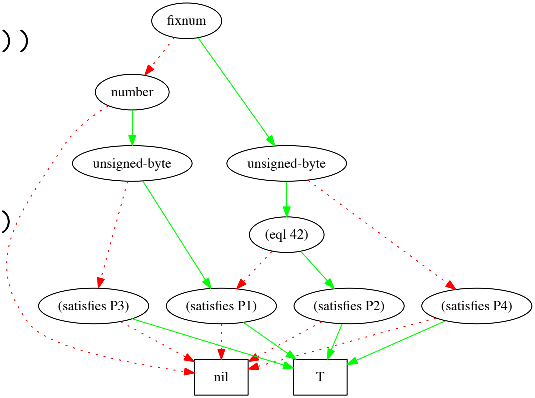


- No type check is done twice.
- Missing `(satisfies P...)` corresponds to unreachable code.
- If a path to $\perp$ avoids `(satisfies P...)`, then the `typecase` is not exhaustive.
- Serializable to efficient Common Lisp code.

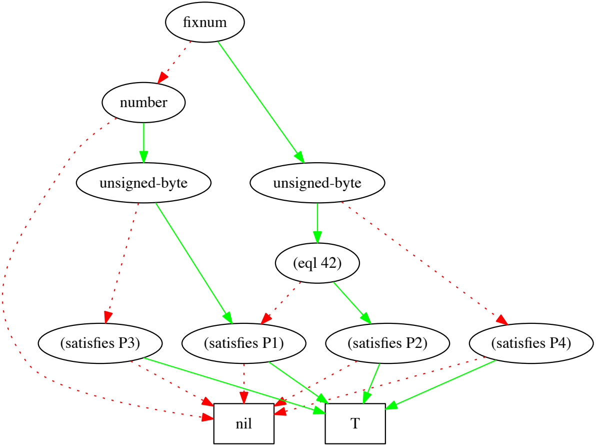
Bigger `bdd-typecase` example

Invocation of `bdd-typecase`

```
(bdd-typecase obj
  ((and unsigned-byte
        (not (eql 42)))
   body-forms-1...)
  ((eql 42)
   body-forms-2...)
  ((and number
        (not (eql 42))
        (not fixnum))
   body-forms-3...)
  (fixnum
   body-forms-4...))
```

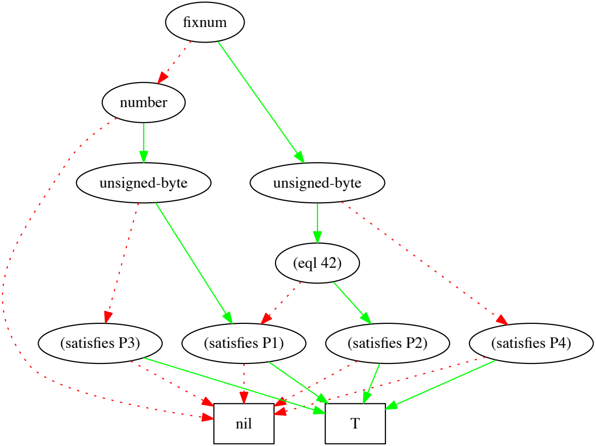


Bigger `bdd-typecase` example



- No duplicate type checks.

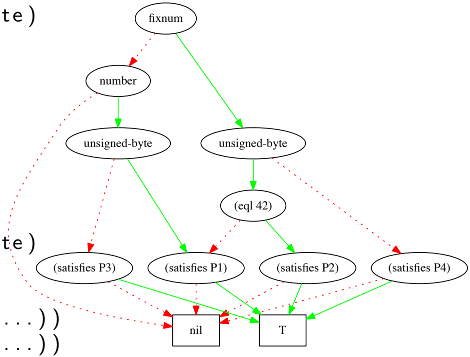
Bigger `bdd-typecase` example



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

Bigger `bdd-typecase` simplified example with `tagbody/go`.

```
(let ((obj obj))
 (tagbody
   L1 (if (typep obj 'fixnum)
          (go L2)
          (go L4))
   L2 (if (typep obj 'unsigned-byte)
          (go L3)
          (go P4))
   L3 (if (typep obj '(eql 42))
          (go P2)
          (go P1))
   L4 (if (typep obj 'number)
          (go L5)
          (return nil))
   L5 (if (typep obj 'unsigned-byte)
          (go P1)
          (go P3))
   P1 (return (progn body-forms-1 ...))
   P2 (return (progn body-forms-2 ...))
   P3 (return (progn body-forms-3 ...))
   P4 (return (progn body-forms-4 ...))))
```



Bigger `bdd-typecase` example with `labels`.

```
(let ((obj obj))
   (labels ((L1 () (if (typep obj 'fixnum)
                       (L2)
                       (L4)))
            (L2 () (if (typep obj 'unsigned-byte)
                       (L3)
                       (P4)))
            (L3 () (if (typep obj '(eql 42))
                       (P2)
                       (P1)))
            (L4 () (if (typep obj 'number)
                       (L5)
                       nil))
            (L5 () (if (typep obj 'unsigned-byte)
                       (P1)
                       (P3)))
            (P1 () body-forms-1 ...)
            (P2 () body-forms-2 ...)
            (P3 () body-forms-3 ...)
            (P4 () body-forms-4 ...))
    (L1)))
```

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	$\|ROBDD_N\|$
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.
- Worst case size is calculable.

$$|ROBDD_N| = (2^{N-\theta} - 1) + 2^{2^{\theta}}$$

where

$$\lceil \log_2(N - 2 - \log_2 N) \rceil - 2 \leq \theta \leq \lfloor \log_2 N \rfloor$$

ROBDD worst case size

N	$\|ROBDD_N\|$
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.
- Worst case size is calculable.

$$|ROBDD_N| = (2^{N-\theta} - 1) + 2^{2^{\theta}}$$

where

$$\lceil \log_2(N - 2 - \log_2 N) \rceil - 2 \le \theta \le \lfloor \log_2 N \rfloor$$

- But our ROBDD is never worst-case.

Table of Contents

Summary

- `auto-permute-typecase` : find *b*est simplification by exhaustive search
  - Combinatorical compile-time complexity
  - Sometimes fails to remove duplicate checks.
  - Difficult to implement a good/fast `type-simplify` function, (`subtypep` et.al.).
  - Heuristic function, topic for more research.

Summary

- `auto-permute-typecase` : find *best* simplification by exhaustive search
  - Combinatorical compile-time complexity
  - Sometimes fails to remove duplicate checks.
  - Difficult to implement a good/fast `type-simplify` function, (`subtypep` et.al.).
  - Heuristic function, topic for more research.
- `bdd-typecase` : expand `typecase` into inline state machine.
  - Eliminates duplicate checks
  - Exponential code size
  - Always removes duplicate type checks
  - Ongoing research to optimize CL-ROBDD.

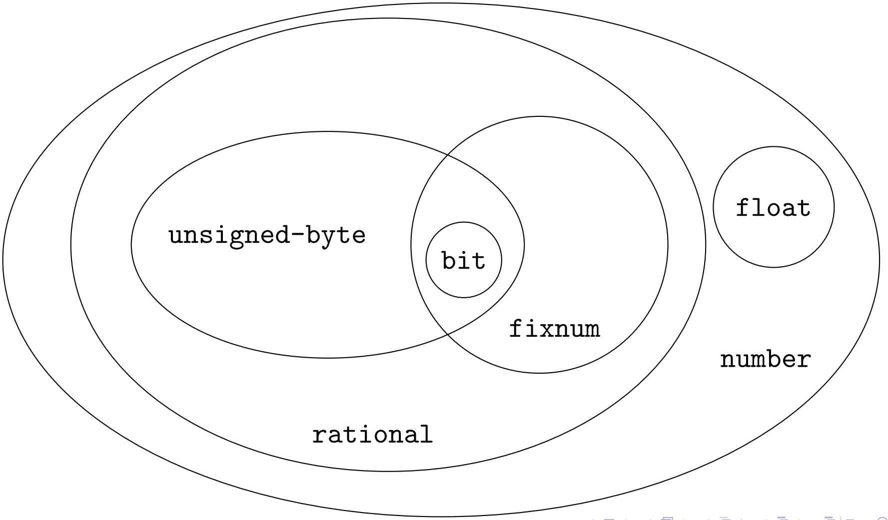
Summary

- `auto-permute-typecase` : find *b*est simplification by exhaustive search
  - Combinatorical compile-time complexity
  - Sometimes fails to remove duplicate checks.
  - Difficult to implement a good/fast `type-simplify` function, (`subtypep` et.al.).
  - Heuristic function, topic for more research.
- `bdd-typecase` : expand `typecase` into inline state machine.
  - Eliminates duplicate checks
  - Exponential code size
  - Always removes duplicate type checks
  - Ongoing research to optimize CL-ROBDD.
- Both approaches
  - Find unreachable code
  - Find non-exhaustive cases

Questions/Answers

Questions?

Examples of some Common Lisp types, and their intersections



Type specifiers are powerful and intuitive

Homoiconicity makes type specifiers intuitive and flexible.

- Simple
  - `integer`

Type specifiers are powerful and intuitive

Homoiconicity makes type specifiers intuitive and flexible.

- Simple
  - `integer`
- Compound type specifiers
  - `(satisfies oddp)`
  - `(float (0.0) 1.0)`
  - `(member 2 5 7 11)`

Type specifiers are powerful and intuitive

Homoiconicity makes type specifiers intuitive and flexible.

- Simple
  - `integer`
- Compound type specifiers
  - `(satisfies oddp)`
  - `(float (0.0) 1.0)`
  - `(member 2 5 7 11)`
- Logical combinations
  - `(and (or number string) (not (satisfies MY-FUN)))`

Type specifiers are powerful and intuitive

Homoiconicity makes type specifiers intuitive and flexible.

- Simple
  - `integer`
- Compound type specifiers
  - `(satisfies oddp)`
  - `(float (0.0) 1.0)`
  - `(member 2 5 7 11)`
- Logical combinations
  - `(and (or number string) (not (satisfies MY-FUN)))`
- Specifiers for the empty type
  - `nil`
  - `(and number string)`

Macro expansion of `typecase`

Example `macroexpand-1` from SBCL.

```
(typecase x
  ((and fixnum (not (eql 0))) (f1))
  ((eql 0) (f2))
  (symbol (f3))
  (t (f4)))

;; macro expansion
(let ((\#:g604 x))
  (declare (ignorable \#:g604))
  (cond ((typep \#:g604 '(and fixnum (not (eql 0)))) nil (f1))
        ((typep \#:g604 '(eql 0)) nil (f2))
        ((typep \#:g604 'symbol) nil (f3))
        (t nil (f4))))
```

Macro expansion of `typecase`

Example `macroexpand-1` from SBCL.

```
(typecase x
  ((and fixnum (not (eql 0))) (f1))
  ((eql 0) (f2))
  (symbol (f3))
  (t (f4)))

;; macro expansion
(let ((\#:g604 x))
  (declare (ignorable \#:g604))
  (cond ((typep \#:g604 '(and fixnum (not (eql 0)))) nil (f1))
        ((typep \#:g604 '(eql 0)) nil (f2))
        ((typep \#:g604 'symbol) nil (f3))
        (t nil (f4))))
```

We can clean up the expansion to make it easier to understand.

Macro expansion of `typecase`

```
(typecase x
  ((and fixnum (not (eql 0))) (f1))
  ((eql 0) (f2))
  (symbol (f3))
  (t (f4)))
```

Temporary variable because `x` might be an expression.

```
(let ((\#:g604 x))
  (declare (ignorable \#:g604))
  (cond ((typep \#:g604 '(and fixnum (not (eql 0)))) nil (f1))
        ((typep \#:g604 '(eql 0)) nil (f2))
        ((typep \#:g604 'symbol) nil (f3))
        (t nil (f4))))
```

Macro expansion of `typecase`

```
(typecase x
  ((and fixnum (not (eql 0))) (f1))
  ((eql 0) (f2))
  (symbol (f3))
  (t (f4)))
```

Protection against certain trivial/degenerate cases.

```
(let ((\#:g604 x))
  (declare (ignorable \#:g604))
  (cond ((typep \#:g604 '(and fixnum (not (eql 0)))) nil (f1))
        ((typep \#:g604 '(eql 0)) nil (f2))
        ((typep \#:g604 'symbol) nil (f3))
        (t nil (f4))))
```

Machine generated, redundant checks

- Redundant type checks
- Unreachable code
- Exhaustiveness

```
(typecase (prog1 (aref seq i) (incf i))
  (fixnum
    (go 7))
  ((and real (not fixnum) (not ratio))
    (go 11))
  ((or ratio (and number (not real)))
    (go 10))
  (t (return-from check nil)))
```

Example of machine-generated code containing repeated type checks: `fixnum` and `ratio`.

Reorderable clauses

```
(typecase obj
  (fixnum                              (f1))
  ((and number (not fixnum))           (f2))
  ((and t (not (or fixnum number)))    (f3)))
```

Now the clauses can be reordered.

```
(typecase obj
  ((and number (not fixnum))           (f2))
  (fixnum                              (f1))
  ((and t (not (or fixnum number)))    (f3)))
```

Heuristics for code cost

- When comparing two type specifiers:
  - built-in types are cheap
  - `satisfies` is expensive
  - `and`, `or`, `not` cost depend on the tree size

Heuristics for code cost

- When comparing two type specifiers:
  - built-in types are cheap
  - `satisfies` is expensive
  - `and`, `or`, `not` cost depend on the tree size
- When comparing two `typecase` expressions:
  - Better to have simple expressions early

    ```
    (typecase obj
      (fixnum                      (f1))
      ((and number (not ratio)) (f2)))
    ```

  - Than to have complex expressions early

    ```
    (typecase obj
      ((and number (not ratio)) (f2))
      (fixnum                      (f1)))
    ```