Strategies for typecase optimization

Jim Newton

11th European Lisp Symposium

16-17 April 2017
1. Motivation and Background

2. Intro to Common Lisp Types and typecase

3. Optimization by s-expression transformation

4. Optimization using decision diagrams

5. Conclusion
Motivation and Background

1. Motivation and Background
2. Intro to Common Lisp Types and typecase
3. Optimization by s-expression transformation
4. Optimization using decision diagrams
5. Conclusion
Rational Type Expressions (RTE) recognize sequences based on element type.

Code gen for RTE: excessive use of `typecase` with complex, machine generated type specifiers.
(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
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

1 Motivation and Background

2 Intro to Common Lisp Types and typecase

3 Optimization by s-expression transformation

4 Optimization using decision diagrams

5 Conclusion
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

- $\text{integer} \subseteq \text{rational}$
- $\text{ratio} = \text{rational} \cap \text{integer}$
- $\text{ratio} = (\text{and} \ \text{rational} \ (\text{not} \ \text{integer}))$
Some types can be identified Boolean operations

- $\text{integer} \subseteq \text{rational}$
- $\text{ratio} = \text{rational} \cap \text{integer}$
- $\text{float} \subseteq \text{rational}$
- $\emptyset = \text{rational} \cap \text{float}$
- $\text{nil} = (\text{and rational (not integer)})$

![Venn diagram showing the relationships between integer, rational, ratio, and float types]
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**
  
  ```lisp
  (typecase expr
    (fixnum body-forms-1...)
    (number body-forms-2...)
    (string body-forms-3...))
  ```

- **typecase may use any valid type specifier.**
  
  ```lisp
  (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**
  
  ```lisp
  (typecase expr
    (fixnum body-forms-1...)
    (number body-forms-2...)
    (string body-forms-3...))
  
  typecase may use any valid type specifier.

  ```lisp
  (typecase expr
    ((and fixnum (not (eq l 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

1 Motivation and Background

2 Intro to Common Lisp Types and typecase

3 Optimization by s-expression transformation

4 Optimization using decision diagrams

5 Conclusion
Macro expansion of typecase

- We can use macroexpand-1 from SBCL.

\[
\text{(typecase } x \\
\quad (((\text{and} \ \text{fixnum} \ (\text{not} \ (\text{eql} \ 0))) \ (f1))) \\
\quad ((\text{eql} \ 0) \ (f2)) \\
\quad ((\text{symbol}) \ (f3)) \\
\quad (t \ (f4))))
\]
Macro expansion of \textit{typecase}

- We can use \texttt{macroexpand-1} from SBCL.

\begin{verbatim}
(typecase x
  ((and fixnum (not (eq l 0))) (f1))
  ((eq l 0) (f2))
  (symbol (f3))
  (t (f4)))
\end{verbatim}

- The expansion \textit{essentially} involves \texttt{cond} and \texttt{typep}.

\begin{verbatim}
(cond ((typep x '(and fixnum (not (eq l 0)))) (f1))
  ((typep x '(eq l 0)) (f2))
  ((typep x 'symbol) (f3))
  (t (f4)))
\end{verbatim}
Issues we wish to address

- Redundant type checks
- Unreachable code
- Exhaustiveness
Redundant type checks

- Redundant type checks
- Unreachable code
- Exhaustiveness

\[
\text{(typecase obj}
\begin{align*}
& ((\text{and fixnum (not bignum)}) \ (f1)) \\
& ((\text{and bignum (not unsigned-byte)}) \ (f2)) \\
& (\text{bignum} \ (f3))
\end{align*}
\]

- The type check for \text{bignum} might be executed multiple times. Perhaps not an enormous problem...
Redundant type checks

- Redundant type checks
- Unreachable code
- Exhaustiveness

\[(\text{typecase } \text{obj} \quad (((\text{and } \text{fixnum} \quad (\text{not } \text{bignum})) \quad (f1)) \quad (((\text{and } \text{bignum} \quad (\text{not } \text{unsigned-byte})) \quad (f2)) \quad (\text{bignum} \quad (f3))))\]

- The type check for \text{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

\[
\text{(typecase } \text{obj} \\
((\text{or number string symbol}) \quad (f1)) \\
((\text{and (satisfies slow-predicate) number}) \quad (f2)) \\
((\text{and (satisfies slow-predicate) (or symbol string)}) \quad (f3)))
\]

- The function calls, \((f2)\) and \((f3)\), are unreachable.
Unreachable code

- Redundant type checks
- Unreachable code
- Exhaustiveness

\[
\text{typecase obj}
\]
\[
((\text{or number string symbol}) (f1))
\]
\[
((\text{and (satisfies slow-predicate) number}) (f2))
\]
\[
((\text{and (satisfies slow-predicate) (or symbol string)}) (f3))
\]

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

- Redundant type checks
- Unreachable code
- Exhaustiveness

\[
\text{typcase obj}
\begin{align*}
((\text{or number string symbol}) & \quad (f1)) \\
((\text{and (satisfies slow-predicate) number}) & \quad (f2)) \\
((\text{and (satisfies slow-predicate) (or symbol string)}) & \quad (f3)))
\end{align*}
\]

- 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

\[
\text{(typecase obj}
\begin{align*}
& ((\text{not (or number symbol)}) \quad (f1)) \\
& (\text{number} \quad (f2)) \\
& (\text{symbol} \quad (f3))
\end{align*}
\]

The final \text{symbol} check is unnecessary, can be replaced with \text{T}.

\[
\text{(typecase obj}
\begin{align*}
& ((\text{not (or number symbol)}) \quad (f1)) \\
& (\text{number} \quad (f2)) \\
& (\text{t} \quad (f3))
\end{align*}
\]
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)))
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
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.
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
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

  \[
  \text{typecase obj}
  \begin{align*}
  &\text{(number (f1))} \\
  &\text{(fixnum (f2)); f2 unreachable} \\
  &\text{(t (f3)))}
  \end{align*}
  \]

  vs.

  \[
  \text{typecase obj}
  \begin{align*}
  &\text{(fixnum (f2)); f2 reachable} \\
  &\text{(number (f1))} \\
  &\text{(t (f3)))}
  \end{align*}
  \]
Semantics of typecase depends on order of clauses. E.g., \( \text{obj} = 2 \)

\[
\text{(typecase obj}
  \begin{cases}
    \text{number (f1)} \\
    \text{fixnum (f2)} \text{ ; f2 unreachable} \\
    \text{t (f3))}
  \end{cases}
\text{)}
\]

vs.

\[
\text{(typecase obj}
  \begin{cases}
    \text{fixnum (f2)} \text{ ; f2 reachable} \\
    \text{number (f1)} \\
    \text{t (f3))}
  \end{cases}
\text{)}
\]

Unreachable code, but forward substitution does not find it.
Order dependent clauses

- **Semantics of typecase depends on order of clauses.** E.g., \( \text{obj}=2 \)

\[
\text{(typecase \ obj} \\
\text{ (number \ (f1))} \\
\text{(fixnum \ (f2)) ; f2 \ unreachable} \\
\text{(t \ (f3)))}
\]

vs.

\[
\text{(typecase \ obj} \\
\text{ (fixnum \ (f2)) ; f2 \ reachable} \\
\text{(number \ (f1))} \\
\text{(t \ (f3)))}
\]

- **Unreachable code, but forward substitution does not find it.**
- \((f2)\) unreachable because \(\text{fixnum} \subset \text{number}\)
But, we can rewrite the type checks...

1: (typecase obj
2: (number (f1))
3: (fixnum (f2))
4: (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:   ((not number) (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.

```lisp
(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
Jim Newton (11th European Lisp Symposium) Strategies for typecase optimization 16-17 April 2017 23/41
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

1 Motivation and Background

2 Intro to Common Lisp Types and typecase

3 Optimization by s-expression transformation

4 Optimization using decision diagrams

5 Conclusion
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))
```
Optimization using decision diagrams

Decision Diagram representing irreducible typecase

This code flow diagram represents the calculation we want.
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 and 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
  \[ A \overline{C} \overline{D} + \overline{A}B\overline{C} \overline{D} + \overline{A}B\overline{D} \]
- Using Common Lisp type specifier notation
  
  \[
  \text{(or (and A (not C) (not D)))}
  \text{(and (not A) B (not C) (not D)))}
  \text{(and (not A) (not B) (not D)))}
  \]

\[ \begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\text{D} \\
\text{T} \\
\bot \\
\end{array} \]
Type specifier as ROBDD

CL-ROBDD

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

Question: Can we convert typecase into a type specifier?
Answer: Yes.
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?

Jim Newton (11th European Lisp Symposium) Strategies for typecase optimization 16-17 April 2017 29 / 41
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

\[
\text{(typecase obj} \\
(T.1 \text{ body-forms-1...}) \\
(T.2 \text{ body-forms-2...}) \\
\cdots \\
(T.n \text{ body-forms-n...}))
\]

- Encapsulate body-forms into named predicate functions.
  \[
P_1 \leftarrow (\text{encapsulate-as-predicate body-forms-1...}) \\
P_2 \leftarrow (\text{encapsulate-as-predicate body-forms-2...}) \\
\cdots \\
P_n \leftarrow (\text{encapsulate-as-predicate body-forms-n...})
\]
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

\[
\text{(bdd-typecase obj}
\begin{align*}
\text{((and unsigned-byte}
\begin{align*}
\text{(not bignum))}
\text{body-forms-1 \ldots)}
\text{((and bignum}
\begin{align*}
\text{(not unsigned-byte))}
\text{body-forms-2 \ldots))}
\end{align*}
\end{align*}
\[
\text{unsigned-byte}
\begin{align*}
\text{bignum}
\begin{align*}
\text{(satisfies P2)}
\end{align*}
\end{align*}
\begin{align*}
\text{bignum}
\begin{align*}
\text{(satisfies P1)}
\end{align*}
\end{align*}
\begin{align*}
\perp
\end{align*}
\end{align*}
\begin{align*}
\top
\end{align*}
\end{align*}
\]

Now we can represent a \textit{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 ⊥ avoids (satisfies P...), then the typecase is not exhaustive.
- Serializable to efficient Common Lisp code.
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...))
Optimization using decision diagrams

Bigger `bdd-typecase` example

- No duplicate type checks.
Bigger \textit{bdd-typecase} example

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

```lisp
(let ((obj obj))
  (labels ((L1 () (if (typep obj 'fixnum)
                   (L2)
                   (L4))))
            (L2 () (if (typep obj 'unsigned-byte)
                     (L3)
                     (P4))))
            (L3 () (if (typep obj '(eq l 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))))
)
```

Jim Newton (11th European Lisp Symposium)
Strategies for typecase optimization
16-17 April 2017 37 / 41
## 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.

$$|ROBDD_N| = (2^N - \theta - 1) + 2^{\lceil \log_2 (N - 2 - \log_2 N) \rceil - 2} \leq \theta \leq \lfloor \log_2 N \rfloor$$

But our ROBDD is never worst-case.
### ROBDD worst case size

| $N$ | $|\text{ROBDD}_N|$ |
|-----|----------------|
| 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.
ROBDD worst case size

<table>
<thead>
<tr>
<th>N</th>
<th>ROBDD_N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>47</td>
</tr>
<tr>
<td>8</td>
<td>79</td>
</tr>
<tr>
<td>9</td>
<td>143</td>
</tr>
<tr>
<td>10</td>
<td>271</td>
</tr>
<tr>
<td>11</td>
<td>511</td>
</tr>
<tr>
<td>12</td>
<td>767</td>
</tr>
<tr>
<td>13</td>
<td>1279</td>
</tr>
<tr>
<td>14</td>
<td>2303</td>
</tr>
<tr>
<td>15</td>
<td>4351</td>
</tr>
</tbody>
</table>

- 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.

\[
| \text{ROBDD}_N | = (2^{N-\theta} - 1) + 2^{2\theta}
\]

where

\[
\left\lceil \log_2(N - 2 - \log_2 N) \right\rceil - 2 \leq \theta \leq \left\lfloor \log_2 N \right\rfloor
\]
Optimization using decision diagrams

**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

$$[\log_2(N - 2 - \log_2 N)] - 2 \leq \theta \leq \lfloor \log_2 N \rfloor$$

- But our ROBDD is never worst-case.
Table of Contents

1. Motivation and Background
2. Intro to Common Lisp Types and typecase
3. Optimization by s-expression transformation
4. Optimization using decision diagrams
5. Conclusion
Summary

- **auto-permute-typecase**: find best simplification by exhaustive search
  - Combinatorial 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 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.

- Both approaches
  - Find unreachable code
  - Find non-exhaustive cases
Questions?
Examples of some Common Lisp types, and their intersections

unsinged-byte

bit

fixnum

rational

float

number
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.

```lisp
(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))))
```

Jim Newton (11th European Lisp Symposium) Strategies for typecase optimization 16-17 April 2017 44 / 41
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 (ignoreable \#: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

\[
\begin{align*}
&\text{(typeof} \ (\text{progl} \ (\text{aref} \ \text{seq} \ i) \ (\text{incf} \ i)) \\
&\quad (\text{fixnum} \\
&\quad \quad (\text{go} \ 7)) \\
&\quad ((\text{and} \ \text{real} \ (\text{not} \ \text{fixnum}) \ (\text{not} \ \text{ratio})) \\
&\quad \quad (\text{go} \ 11)) \\
&\quad ((\text{or} \ \text{ratio} \ (\text{and} \ \text{number} \ (\text{not} \ \text{real}))) \\
&\quad \quad (\text{go} \ 10)) \\
&\quad (t \ (\text{return-from} \ \text{check} \ \text{nil})))
\end{align*}
\]

Example of machine-generated code containing repeated type checks: \text{fixnum} and \text{ratio}. 
Reorderable clauses

\[(\text{typecase } \text{obj})\]
\[(\text{fixnum} \quad (f1))\]
\[((\text{and } \text{number} \ (\text{not} \ \text{fixnum})) \quad (f2))\]
\[((\text{and } \text{t} \ (\text{not} \ (\text{or} \ \text{fixnum} \ \text{number})))) \quad (f3))\]\n
Now the clauses can be reordered.

\[(\text{typecase } \text{obj})\]
\[((\text{and } \text{number} \ (\text{not} \ \text{fixnum})) \quad (f2))\]
\[(\text{fixnum} \quad (f1))\]
\[((\text{and } \text{t} \ (\text{not} \ (\text{or} \ \text{fixnum} \ \text{number})))) \quad (f3))\]
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
    
    \[
    \text{typecase obj}
    \begin{align*}
    \text{(fixnum } & (f1)) \\
    ((\text{and number (not ratio)}) & (f2))
    \end{align*}
    \]
  - Than to have complex expressions early
    
    \[
    \text{typecase obj}
    \begin{align*}
    ((\text{and number (not ratio)}) & (f2)) \\
    \text{(fixnum } & (f1))
    \end{align*}
    \]