Programmatic Manipulation of Common Lisp Type Specifiers

Jim E. Newton
jnewton@lrde.epita.fr

Didier Verna
didier@lrde.epita.fr

Maximilien Colange
maximilien.colange@lrde.epita.fr

EPITA/LRDE
14-16 rue Voltaire
F-94270 Le Kremlin-Bicêtre
France

ABSTRACT

In this article we contrast the use of the s-expression with the BDD (Binary Decision Diagram) as a data structure for programmatically manipulating Common Lisp type specifiers. The s-expression is the *de facto* standard surface syntax and also programmatic representation of the type specifier, but the BDD data structure offers advantages: most notably, type equivalence checks using s-expressions can be computationally intensive, whereas the type equivalence check using BDDs is a check for object identity. As an implementation and performance experiment, we define the notion of maximal disjoint type decomposition, and discuss implementations of algorithms to compute it: a brute force iteration, and as a tree reduction. The experimental implementations represent type specifiers by both aforementioned data structures, and we compare the performance observed in each approach.

CCS Concepts

•Theory of computation → Data structures design and analysis; *Type theory;* **•Computing methodologies → Representation of Boolean functions; •Mathematics of computing →** *Graph algorithms;*

1. INTRODUCTION

Common Lisp programs which manipulate type specifiers have traditionally used s-expressions as the programmatic representations of types, as described in the Common Lisp specification [Ans94, Section 4.2.3]. Such choice of internal data structure offers advantages such as homoiconicity, making the internal representation human readable in simple cases, and making programmatic manipulation intuitive, as well as enabling the direct use of built-in Common Lisp functions such as `typep` and `subtypep`. However, this approach does present some challenges. Such programs often make use of ad-hoc logic reducers—attempting to convert types to canonical form. These reducers can be complicated and difficult to debug. In addition run-time decisions about type equivalence and subtyping can suffer performance problems.

In this article we present an alternative internal representation for Common Lisp types: the Binary Decision Diagram (BDD) [Bry86, Ake78]. BDDs have interesting characteristics such as representational equality; *i.e.* it can be arranged that equivalent expressions or equivalent sub-expressions are represented by the same object (`eq`). While techniques to implement BDDs with these properties are well documented, an attempt apply the techniques directly to the Common Lisp type system encounters obstacles which we analyze and document in this article.

In order to compare performance characteristics of the two data structure approaches, we have constructed a problem called Maximal Disjoint Type Decomposition (MDTD): decomposing a given set of potentially overlapping types into a set of disjoint types. Although MDTD is interesting in its own right, we do not attempt, in this paper, to motivate in detail the applications or implications of the problem. We consider such development and motivation a matter of future research. Our use of the MDTD problem in this article is primarily a performance comparison vehicle.

We present two algorithms to compute the MDTD, and separately implement the algorithms with both data structures s-expressions and BDDs (4 implementations in total). Finally, we report performance characteristics of the four algorithms implemented in Common Lisp.

Key contributions of this article are:

- A description of how to extended known BDD related implementation techniques to represent Common Lisp types and facility type based calculations.
- Performance comparison of algorithms using traditional s-expression based type specifiers *vs.* using the BDD data structure.
- A graph based algorithm for reducing the computational complexity of MDTD.

2. DISJOINT TYPE DECOMPOSITION

In presenting the problem of decomposing a set of overlapping types into non-overlapping subtypes, we start with an example intended to convey an intuition of the problem. We continue by defining precisely what we intend to calcu-

ELS '10 April 3–4, 2016, Brussels, Belgium

ACM ISBN 978-1-4503-2138-9.

DOI: 10.5281/zenodo.3405363

late. Then in sections 2.1 and 2.2 we present two different algorithms for performing that calculation.

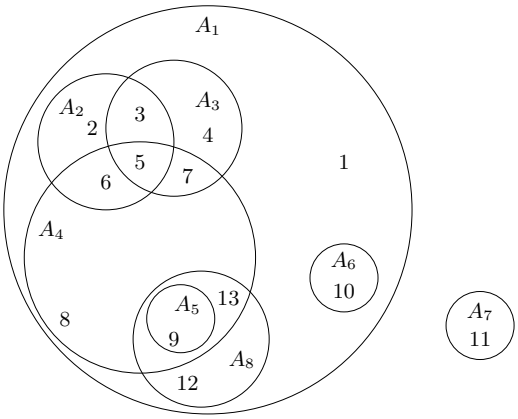


Figure 1: Example Venn Diagram

Disjoint Set	Derived Expression
X_1	$A_1 \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_4} \cap \overline{A_6} \cap \overline{A_8}$
X_2	$A_2 \cap \overline{A_3} \cap \overline{A_4}$
X_3	$A_2 \cap A_3 \cap \overline{A_4}$
X_4	$A_3 \cap \overline{A_2} \cap \overline{A_4}$
X_5	$A_2 \cap A_3 \cap A_4$
X_6	$A_2 \cap A_4 \cap \overline{A_3}$
X_7	$A_3 \cap A_4 \cap \overline{A_2}$
X_8	$A_4 \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_8}$
X_9	A_5
X_{10}	A_6
X_{11}	A_7
X_{12}	$A_8 \cap \overline{A_4}$
X_{13}	$A_4 \cap A_8 \cap \overline{A_5}$

Figure 2: Disjoint Decomposition of Sets from Figure 1

In the Venn diagram in Figure 1, $V = \{A_1, A_2, ..., A_8\}$. We wish to construct logical combinations of those sets to form as many mutually disjoint subsets as possible. The resulting *decomposition* should have the same union as the original set. The maximal disjoint decomposition $D = \{X_1, X_2, ..., X_{13}\}$ of V is shown in Figure 2.

NOTATION 1. *We use the symbol, $\perp$, to indicate the disjoint relation between sets. I.e., we take $A \perp B$ to mean $A \cap B = \varnothing$. We also say $A \not\perp B$ to mean $A \cap B \neq \varnothing$.*

NOTATION 2. *We use the notation, $A \subset B$, $(A \supset B)$ to indicate that A is either a strict subset (superset) of B or is equal to B.*

DEFINITION 1. *Let U be a set and V be a set of subsets of U. The Boolean closure of V, denoted $\widehat{V}$, is the (smallest) super-set of V such that $\alpha, \beta \in \widehat{V} \implies \{\alpha \cap \beta, \alpha \cap \overline{\beta}\} \subset \widehat{V}$.*

DEFINITION 2. *Let U be a set, and let V and D be finite sets of non-empty subsets of U. D is said to be a disjoint decomposition of V, if the elements of D are mutually disjoint, $D \subset \widehat{V}$, and $\bigcup_{X \in D} X = \bigcup_{A \in V} A$. If no larger set fulfills those properties, D is said to be the maximal disjoint decomposition of V.*

We claim without proof that there exists a unique maximal disjoint decomposition of a given V. A more complete discussion and formal proof are available [New17].

The MDTD problem: Given a set U and a set of subsets thereof, $V = \{A_1, A_2, ..., A_M\}$, suppose that for each pair (A_i, A_j), we know which of the relations hold: $A_i \subset A_j$, $A_i \supset A_j$, $A_i \perp A_j$. We would like to compute the maximal disjoint decomposition of V.

In Common Lisp, a type is a set of (potential) values [Ans94, Section Type], so it makes sense to consider the maximal disjoint decomposition of a set of types.

2.1 The RTE Algorithm

We first encountered the MDTD problem in our previous work on regular type expressions (RTE) [NDV16]. The following algorithm was the one presented in that paper, where we pointed that the algorithm suffers from significant performance issues. Performance issues aside, a notable feature of the RTE version of the MDTD algorithm is that it easily fits in 40 lines of Common Lisp code, so it is easy to implement and easy to understand.

1. Let U be the set of sets. Let V denote the set of disjoint sets, initially $D = \varnothing$.
2. Identify all the sets which are disjoint from each other and from all the other sets. ($\mathcal{O}(n^2)$ search) Remove these sets from U and collect them in D.
3. If possible, choose X and Y, for which $X \not\perp Y$.
4. Remove X and Y from U, and add any of $X \cap Y$, $X \backslash Y$, and $Y \backslash X$ which are non-empty. *I.e.*, $U \leftarrow (U \backslash \{X, Y\}) \cup (\{X \cap Y, X \backslash Y, Y \backslash X\} \backslash \{\varnothing\})$
5. Repeat steps 2 through 4 until $U = \varnothing$, at which point we have collected all the disjoint sets in D.

2.2 The graph based algorithm

One of the sources of inefficiency of the algorithm explained in Section 2.1 is at each iteration of the loop, an $\mathcal{O}(n^2)$ search is made to find sets which are disjoint from all remaining sets. This search can be partially obviated if we employ a little extra book-keeping. The fact to realize is that if $X \perp A$ and $X \perp B$, then we know *a priori* that $X \perp A \cap B$, $X \perp A \backslash B$, $X \perp B \backslash A$. This knowledge eliminates some of useless operations.

This algorithm is semantically similar to the algorithm shown in Section 2.1, but rather than relying on Common Lisp primitives to make decisions about connectivity of types, it initializes a graph representing the initial relationships, and thereafter manipulates the graph maintaining connectivity information. This algorithm is more complicated in terms of lines of code, 250 lines of Common Lisp code as opposed to 40 lines for the algorithm in Section 2.1.

Figure 3 shows a graph representing the topology (connectedness) of the diagram shown in Figure 1. Nodes ①, ②, ... ⑧ in Figure 3 correspond respective to A_1, A_2, ... A_8 in

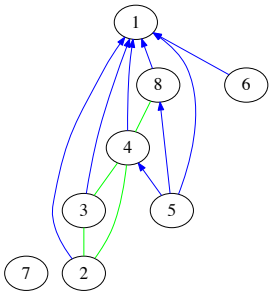


Figure 3: Topology graph

Figure 1. Blue arrows correspond to subset relations, pointing from subset to superset, and green lines correspond to other non-disjoint relations.

To construct this graph first eliminate duplicate sets. *I.e.*, if $X \subset Y$ and $X \supset Y$, then discard either X or Y. It is necessary to consider each pair (X, Y) of sets, $\mathcal{O}(n^2)$ loop.

- If $X \subset Y$, draw a blue arrow $X \to Y$
- Else if $X \supset Y$, draw a blue arrow $X \leftarrow Y$
- Else if $X \not\perp Y$, draw green line between X and Y.
- If it cannot be determined whether $X \subset Y$, assume the worst case, that they are non-disjoint, and draw green line between X and Y.

The algorithm proceeds by breaking the green and blue connections, in explicit ways until all the nodes become isolated. There are two cases to consider. Repeat alternatively applying both tests until all the nodes become isolated.

2.2.1 *Subset relation*

A blue arrow from X to Y may be eliminated if X has no blue arrow pointing to it, in which case Y must be relabeled as $Y \cap \overline{X}$ as indicated in Figure 4.

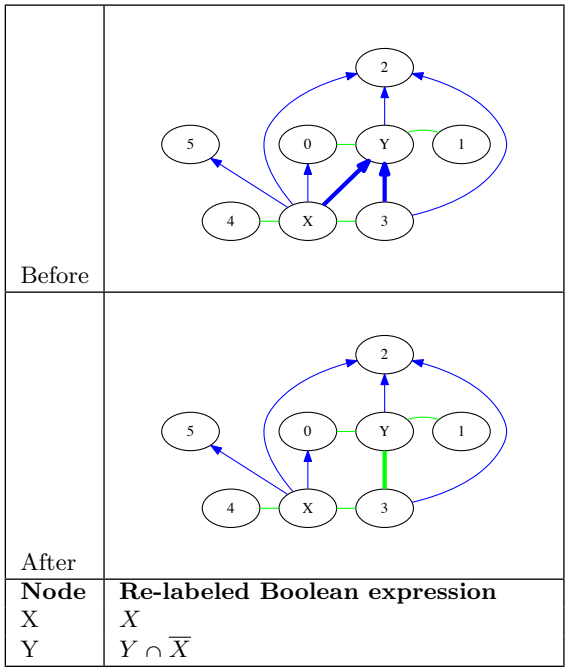
Figure 4 illustrates this mutation. Node Ⓨ may have other connections, including blue arrows pointing to it or from it, and green lines connected to it. However node Ⓧ has no blue arrows pointing to it; although it may have other blue arrows pointing away from it.

If X touches (via a green line) any sibling nodes, *i.e.* any other node that shares Y as super-class, then the blue arrow is converted to a green line. In the *before* image of Figure 4 there is a blue arrow from ③ to Ⓨ and in the *after* image this arrow has been converted to a green line.

2.2.2 *Touching connections*

A green line connecting X and Y may be eliminated if neither X nor Y has a blue arrow pointing to it. Consequently, X and Y must be relabeled and a new node must be added to the graph as indicated in Figure 5. The figure illustrates the step of breaking such a connection between nodes Ⓧ and Ⓨ by introducing the node Ⓩ.

Construct blue arrows from this node, Z, to all the nodes which either X or Y points to (union). Construct green lines from Z to all nodes which both X and Y connect to (intersection). If this process results in two nodes connected both by green and blue, omit the green line.



Node	Re-labeled Boolean expression
X	X
Y	$Y \cap \overline{X}$

Figure 4: Subset before and after mutation

3. TYPE SPECIFIER MANIPULATION

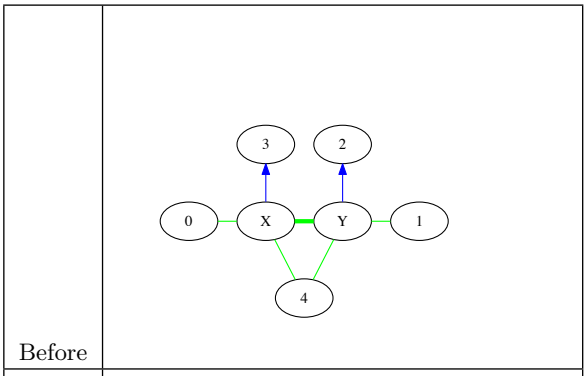
To correctly implement the MDTD by either strategy described above, we need operators to test for type-equality, type disjoint-ness, subtype-ness, and type-emptiness. Given a *subtype* predicate, the other predicates can be constructed. The emptiness check: $A = \varnothing \iff A \subset \varnothing$. The disjoint check: $A \perp B \iff A \cap B \subset \varnothing$. Type equivalence $A = B \iff A \subset B$ and $B \subset A$.

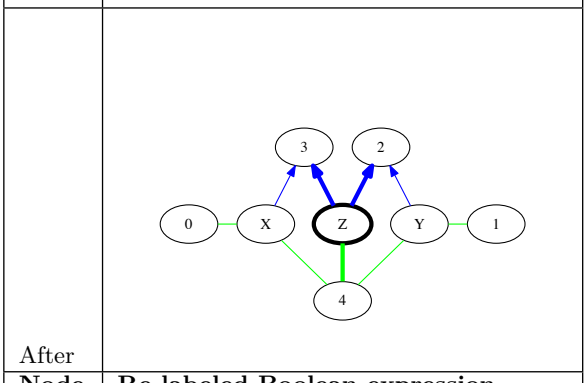
Common Lisp has a flexible type calculus making type specifiers human readable and also related computation possible. Even with certain limitations, s-expressions are an intuitive data structure for programmatic manipulation of type specifiers in analyzing and reasoning about types.

If `T1` and `T2` are Common Lisp type specifiers, the type specifier `(and T1 T2)` designates the intersection of the types. Likewise `(and T1 (not T2))` is the type difference. The empty type and the universal type are designated by `nil` and `t` respectively. The `subtypep` function serves as the subtype predicate. Consequently `(subtypep '(and T1 T2) nil)` computes whether `T1` and `T2` are disjoint.

There is an important caveat however. The `subtypep` function is not always able to determine whether the named types have a subtype relationship [Bak92]. In such a case, `subtypep` returns `nil` as its second value. This situation occurs most notably in the cases involving the `satisfies` type specifier. For example, to determine whether the `(satisfies F)` type is empty, it would be necessary to solve the halting problem, finding values for which the function `F` returns true.

As a simple example of how the Common Lisp programmer might manipulate s-expression based type specifiers, consider the following problem. In SBCL 1.3.0, the expression `(subtypep '(member :x :y) 'keyword)` returns `nil,nil`, rather than `t,t`. Although this is compliant behavior, the result is unsatisfying, because clearly both `:x` and `:y` are elements of the `keyword` type. By manipulating the type





Node	Re-labeled Boolean expression
X	$X \cap \overline{Y}$
Y	$\overline{X} \cap Y$
Z	$X \cap Y$

Figure 5: Touching connections before and after mutation

specifier s-expressions, the user can implement a smarter version of `subtypep` to better handle this particular case. Regrettably, the user cannot force the system to use this smarter version internally.

```
(defun smarter-subtypep (t1 t2)
  (multiple-value-bind (T1<=T2 OK) (subtypep t1 t2)
    (cond
      (OK
       (values T1<=T2 t))
      ;; (eql obj) or (member obj1 ...)
      ((typep t1 '(cons (member eql member)))
       (values (every #'(lambda (obj)
                          (typep obj t2))
                      (cdr t1))
               t))
      (t
       (values nil nil)))))
```

As mentioned above, programs manipulating s-expression based type specifiers can easily compose type intersections, unions, and relative complements as part of reasoning algorithms. Consequently, the resulting programmatically computed type specifiers may become deeply nested, resulting in type specifiers which may be confusing in terms of human readability and debuggability. The following programmatically generated type specifier is perfectly reasonable for programmatic use, but confusing if it appears in an error message, or if the developer encounters it while debugging.

```
(or
 (or (and (and number (not bignum))
          (not (or fixnum (or bit (eql -1)))))
     (and (and (and number (not bignum))
               (not (or fixnum (or bit (eql -1)))))
          (not (or fixnum (or bit (eql -1))))))
 (and (and (and number (not bignum))
           (not (or fixnum (or bit (eql -1)))))
      (not (or fixnum (or bit (eql -1))))))
```

This somewhat obfuscated type specifier is semantically equivalent to the more humanly readable form `(and number (not bignum) (not fixnum))`. Moreover, it is possible to write a Common Lisp function to *simplify* many complex type specifiers to simpler form.

There is a second reason apart from human readability which motivates reduction of type specifiers to canonical form. The problem arises when we wish to programmatically determine whether two s-expressions specify the same type, or in particular when a given type specifier specifies the `nil` type. Sometimes this question can be answered by calls to `subtypep` as in `(and (subtypep T1 T2) (subtypep T2 T1))`. However, as mentioned earlier, `subtypep` is allowed to return `nil,nil` in some situations, rendering this approach futile in many cases. If, on the other hand, two type specifiers can be reduced to the same canonical form, we can conclude that the specified types are equal.

We have implemented such a function, `reduce-lisp-type`. It does a good job of reducing the given type specifier toward a canonical form, by repeatedly recursively descending the expression, re-writing sub-expressions, incrementally moving the expression toward a fixed point. We choose to convert the expression to a disjunctive normal form, *e.g.*, `(or (and (not a) b) (and a b (not c)))`. The reduction procedure follows the models presented by Sussman and Abelson [AS96, p. 108] and Norvig [Nor92, ch. 8].

4. BINARY DECISION DIAGRAMS

A challenge using s-expressions for programmatic representation of type specifiers is the need to after-the-fact reduce complex type specifiers to a canonical form. This reduction can be computationally intense, and difficult to implement correctly. We present here a data structure called the Binary Decision Diagram (BDD) [Bry86, Ake78], which obviates much of the need to reduce to canonical form because it maintains a canonical form by design. Before looking at how the BDD can be used to represent Common Lisp type specifiers, we first look at how BDDs are used traditionally to represent Boolean equations. Thereafter, we explain how this traditional treatment can be enhanced to represent Common Lisp types.

4.1 Representing Boolean expressions

Andersen [And99] summarized many of the algorithms for efficiently manipulating BDDs. Not least important in Andersen's discussion is how to use a hash table and dedicated constructor function to eliminate redundancy within a single BDD and within an interrelated set of BDDs. The result of Andersen's approach is that if you attempt to construct two BDDs to represent two semantically equivalent but syntactically different Boolean expressions, then the two resulting BDDs are pointers to the same object.

Figure 6 shows an example BDD illustrating a function of three Boolean variables: A_1, A_2, and A_3. To reconstruct the DNF (disjunctive normal form), collect the paths from the root node, A_1, to a leaf node of 1, ignoring paths terminated by 0. When the right child is traversed, the Boolean complement ($\neg$) of the label on the node is collected (*e.g.* $\neg A_3$),

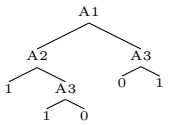


Figure 6: BDD for $(A_1 \wedge A_2) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge \neg A_3)$

and when the left child is traversed the non-inverted parent is collected. Interpret each path as a conjunctive clause, and form a disjunction of the conjunctive clauses. In the figure the three paths from A_1 to 1 identify the three conjunctive clauses $(A_1 \wedge A_2)$, $(A_1 \wedge \neg A_2 \wedge A_3)$, and $(\neg A_1 \wedge \neg A_3)$.

4.2 Representing types

Castagna [Cas16] explains the connection of BDDs to type theoretical calculations, and provides straightforward algorithms for implementing set operations (intersection, union, relative complement) of types using BDDs. The general recursive algorithms for computing the BDDs which represent the common Boolean algebra operators are straightforward.

Let B, B_1, and B_2 denote BDDs, $B_1 = (if\ a_1\ C_1\ D_1)$ and $B_2 = (if\ a_2\ C_2\ D_2)$.

C_1, C_2, D_1, and D_2 represent BDDs. The a_1 and a_2 are intended to represent type names, but for the definition to work it is only necessary that they represent labels which are order-able. We would eventually like the labels to accommodate Common Lisp type type names, but this is not immediately possible.

The formulas for $(B_1 \vee B_2)$, $(B_1 \wedge B_2)$, and $(B_1 \setminus B_2)$ are similar to each other. If $\circ \in \{\vee, \wedge, \setminus\}$, then

$$B_1 \circ B_2 = \begin{cases} (if\ a_1 \quad (C_1 \circ C_2) \quad (D_1 \circ D_2)) & \text{for } a_1 = a_2 \\ (if\ a_1 \quad (C_1 \circ B_2) \quad (D_1 \circ B_2)) & \text{for } a_1 < a_2 \\ (if\ a_2 \quad (B_1 \circ C_2) \quad (B_1 \circ D_2)) & \text{for } a_1 > a_2 \end{cases}$$

There are several special cases, the first three of which serve as termination conditions for the recursive algorithms.

- $(t \vee B)$ and $(B \vee t)$ reduce to t.
- $(nil \wedge B)$, $(B \wedge nil)$, and $(B \backslash t)$ reduce to nil.
- $(t \wedge B)$, $(B \wedge t)$, $(nil \vee B)$, and $(B \vee nil)$ reduce to B.
- $(t \setminus (if\ a\ B_1\ B_2))$ reduces to $(if\ a\ (t \backslash B_1)\ (t \backslash B_2))$.

4.3 Representing Common Lisp types

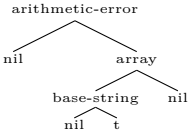
We have implemented the BDD data structure as a set of CLOS classes. In particular, there is one leaf-level CLOS class for an internal tree node, and one singleton class/instance for each of the two possible leaf nodes, *true* and *false*.

The label of the BDD contains a Common Lisp type name, and the logical combinators (`and`, `or`, and `not`) are represented implicitly in the structure of the BDD.

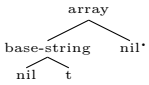
A disadvantage BDDs present when compared to s-expressions as presented in Section 3 is the loss of homoiconicity. Whereas, s-expression based type-specifiers may appear in-line in the Common Lisp code, BDDs may not.

A remarkable fact about this representation is that any two logically equivalent Boolean expressions have exactly the same BDD structural representation, provided the node labels are consistently, totally ordered. Andersen[And99] provides a proof for this claim. For example, the expression from Figure 6, $(A_1 \wedge A_2) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge \neg A_3)$ is equivalent to $\neg((\neg A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2 \vee \neg A_3) \wedge (A_1 \vee A_3))$. So they both have the same shape as shown in the Figure 6. However, if we naïvely substitute Common Lisp type names for Boolean variables in the BDD representation as suggested by Castagna, we find that this equivalence relation does not hold in many cases related to subtype relations in the Common Lisp type system.

An example is that the Common Lisp two types `(and (not arithmetic-error) array (not base-string))` *vs.* `(and array (not base-string))` are equivalent, but the naïvely constructed BDDs are different:



vs.



In order to assure the minimum number of BDD allocations possible, and thus ensure that BDDs which represent equivalent types are actually represented by the same BDD, the suggestion by Andersen [And99] is to intercept the BDD constructor function. This constructor should assure that it never returns two BDD which are semantically equivalent but not `eq`.

4.4 Canonicalization

Several checks are in place to reduce the total number of BDDs allocated, and to help assure that two equivalent Common Lisp types result in the same BDD. The following sections, 4.4.1 through 4.4.5 detail the operations which we found necessary to handle in the BDD construction function in order to assure that equivalent Common Lisp type specifiers result in identical BDDs. The first two come directly from Andersen's work. The remaining are our contribution, and are the cases we found necessary to implement in order to enhance BDDs to be compatible with the Common Lisp type system.

4.4.1 *Equal right and left children*

An optimization noted by Andersen is that if the left and right children are identical then simply return one of them, without allocating a new BDD [And99].

4.4.2 *Caching BDDs*

Another optimization noted by Andersen is that whenever a new BDD is allocated, an entry is made into a hash table so that the next time a request is made with the exactly same label, left child, and right child, the already allocated BDD is returned. We associate each new BDD with a unique integer, and create a hash key which is a list (a triple) of the type specifier (the label) followed by two integers corresponding to the left and right children. We use a Common Lisp `equal` hash table for this storage, although we'd like to investigate whether creating a more specific hash function specific to our key might be more efficient.

4.4.3 *Reduction in the presence of subtypes*

Since the nodes of the BDD represent Common Lisp types, other specific optimizations are made. The cases include situations where types are related to each other in certain ways: subtype, supertype, and disjoint types. In particular there are 12 optimization cases, detailed in Table 1. Each of these optimizations follows a similar pattern: when constructing a BDD with label X, search in either the left or right child

to find a BDD, 

. If X and Y have a particular relation, different for each of the 12 cases, then the 

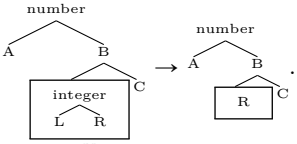
BDD reduces either to L or R. Two cases, 5 and 7, are further illustrated below.

Case	Child to search	Relation	Reduction
1	$X.left$	$X \perp Y$	$Y \rightarrow Y.right$
2	$X.left$	$X \perp \overline{Y}$	$Y \rightarrow Y.left$
3	$X.right$	$\overline{X} \perp Y$	$Y \rightarrow Y.right$
4	$X.right$	$\overline{X} \perp \overline{Y}$	$Y \rightarrow Y.left$
5	$X.right$	$X \supset Y$	$Y \rightarrow Y.right$
6	$X.right$	$X \supset \overline{Y}$	$Y \rightarrow Y.left$
7	$X.left$	$\overline{X} \supset Y$	$Y \rightarrow Y.right$
8	$X.left$	$\overline{X} \supset \overline{Y}$	$Y \rightarrow Y.left$
9	$X.left$	$X \subset Y$	$Y \rightarrow Y.left$
10	$X.left$	$X \subset \overline{Y}$	$Y \rightarrow Y.right$
11	$X.right$	$\overline{X} \subset Y$	$Y \rightarrow Y.left$
12	$X.right$	$\overline{X} \subset \overline{Y}$	$Y \rightarrow Y.right$

Table 1: BDD optimizations

Case 5: If $X \supset Y$ and 

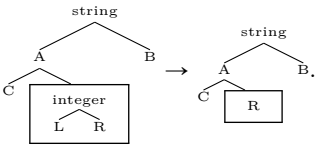
appears in $X.right$, then 

reduces to R. *E.g.*, $integer \subset number$; if $X = number$ and $Y = integer$; thus 

.

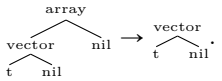
Case 7: If $\overline{X} \supset Y$ and 

appears in $X.left$, then 

reduces to R. *E.g.*, $integer \subset \overline{string}$; if $X = string$ and $Y = integer$; thus 

.

4.4.4 Reduction to child

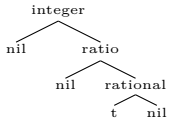
The list of reductions described in Section 4.4.3 fails to apply in cases where the root node itself needs to be eliminated. For example, since $vector \subset array$ we would like the following reductions: 

.

The solution which we have implemented is that before constructing a new BDD, we first ask whether the resulting BDD is type-equivalent to either the left or right children using the `subtypep` function. If so, we simply return the appropriate child without allocating the parent BDD. The expense of this type-equivalence is mitigated by the memoization. Thereafter, the result is in the hash table, and it will be discovered as discussed in Section 4.4.2.

4.4.5 More complex type relations

There are a few more cases which are not covered by the above optimizations. Consider the following BDD:



This represents the type `(and (not integer) (not ratio) rational)`, but in Common Lisp `rational` is identical to `(or integer ratio)`, which means `(and (not integer) (not ratio) rational)` is the empty type. For this reason, as a last resort before allocating a new BDD, we check, using the Common Lisp function `subtypep`, whether the type specifier specifies the `nil` or `t` type. Again this check is expensive, but the expense is mitigated in that the result is cached.

5. MDTD IN COMMON LISP

When attempting to implement the algorithms discussed in Sections 2.1 and 2.2 the developer finds it necessary to choose a data structure to represent type specifiers. Which ever data structure is chosen, the program must calculate type intersections, unions, and relative complements and type equivalence checks and checks for the empty type. As discussed in Section 3, s-expressions (*i.e.* lists and symbols) is a valid choice of data structure and the aforementioned operations may be implemented as list constructions and calls to the `subtypep` predicate.

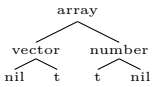


Figure 7: BDD representing `(or number (and array (not vector))`

As introduced in Section 4, another choice of data structure is the BDD. Using the BDD data structure along with the algorithms described in Section 4 we can efficiently represent and manipulate Common Lisp type specifiers. We may programmatically represent Common Lisp types largely independent of the actual type specifier representation. For example the following two type specifiers denote the same set of values: `(or number (and array (not vector)))` and `(not (and (not number) (or (not array) vector)))`, and are both represented by the BDD shown in Figure 5. Moreover, unions, intersections, and relative complements of Common Lisp type specifiers can be calculated using the reduction BDD manipulation rules also explained in Section 4.

We have made comparisons of the two algorithms described in Sections 2.1, 2.2. One implementation of each uses s-expressions, one implementation of each uses BDDs. Some results of the analysis can be seen in Section 6.

Using BDDs in these algorithms allows certain checks to be made more easily than with the s-expression approach. For example, two types are equal if they are the same object (pointer comparison, `eq`). A type is empty if it is identically the empty type (pointer comparison). Finally, given two types (represented by BDDs), the subtype check can be made using the following function:

```
(defun bdd-subtypep (bdd-sub bdd-super)
  (eq *bdd-false*
      (bdd-and-not bdd-sub bdd-super)))
```

This implementation of `bdd-subtype` should not be interpreted to mean that we have obviated the need for the Common Lisp `subtypep` function. In fact, `subtypep`, is still useful in constructing the BDD itself. However, once the

BDDs have been constructed, and cached, subtype checks may at that point avoid calls to `subtypep`, which in some cases might otherwise be more compute intensive.

6. PERFORMANCE OF MDTD

Sections 2.1 and 2.2 explained two different algorithms for calculating type decomposition. We look here at some performance characteristics of the two algorithms. The algorithms from Section 2.1 and Section 2.2 were tested using both the Common Lisp type specifier s-expression as data structure and also using the BDD data structure as described in Section 5. Figures 9 and 8 contrast the four effective algorithms in terms of execution time vs sample size.

We attempted to plot the results many different ways: time as a function of input size, number of disjoint sets in the input, number of new types generated in the output. Some of these plots are available in the technical report [New17]. The plot which we found heuristically to show the strongest visual correlation was calculation time vs the integer product of the number of given input types multiplied by the number of calculated output types. *E.g.*, if the algorithm takes a list of 5 type specifiers and computes 3 disjoint types in 0.1 seconds, the graph contains a point at (15,0.1). Although we don't claim to completely understand why this particular plotting strategy shows better correlation than the others we tried, it does seem that all the algorithms begin a $\mathcal{O}(n^2)$ loop by iterating over the given set of types which is incrementally converted to the output types, so the algorithms in some sense finish by iterating over the output types. More research is needed to better understand the correlation.

6.1 Performance Test Setup

The type specifiers used in Figure 9 are those designating all the subtypes of `fixnum` such as. `(member 2 6 7 9)` and `(member 1 2 8 10)`. The type specifiers used in Figure 8 are those designating a randomly selected set of subtypes of `cl:number` and `cl:condition` together with programmatically generated logical combinations thereof such as `(and number (not bit))` and `(or real type-error)`.

The performance tests comprise starting with a list of randomly selected type specifiers from a pool, calling each of the four functions to calculate the disjoint decomposition, and recording the time of each calculation. We have plotted in Figures 9 and 8 the results of the runs which took less than 30 seconds to complete. This omission does not in any way effect the presentation of which algorithms were the fastest on each test.

The tests were performed on a MacBook 2 GHz Intel Core i7 processor with 16GB 1600 MHz DDR3 memory, and using SBCL 1.3.0 ANSI Common Lisp.

6.2 Analysis of Performance Tests

There is no clear winner for small sample sizes. But it seems the tree based algorithms do very well on large sample sizes. This is not surprising, as the graph based algorithm was designed with the intent to reduce the number of passes, and take advantage of subtype and disjointness information.

Often the better performing of the graph based algorithms is the BDD based one as shown in Figure 8. However there is a notable exception shown in Figures 9 where graph algorithm using s-expressions performs best.

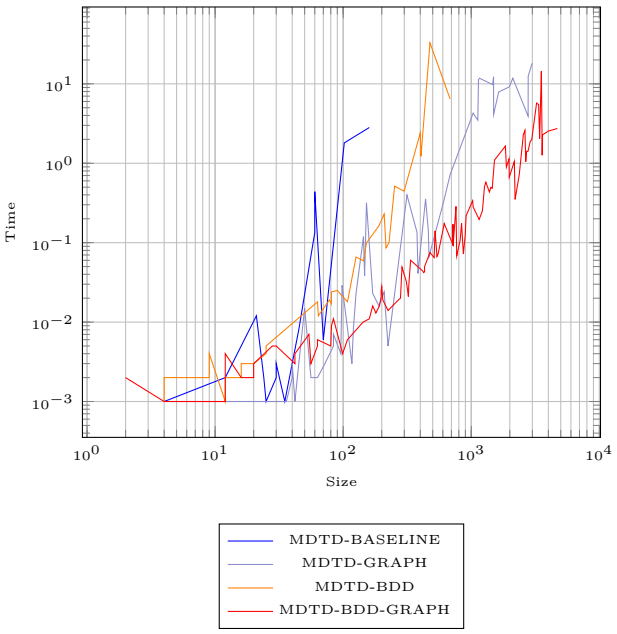


Figure 8: Combinations of number and condition

7. RELATED WORK

Computing a disjoint decomposition when permitted to look into the sets has been referred to as *union find* [PBM10, GF64]. MDTD differs in that we decompose the set without knowledge of the specific elements; *i.e.* we are not permitted to iterate over or visit the individual elements. The correspondence of types to sets and subtypes to subsets thereof is also treated extensively in the theory of semantic subtyping [CF05].

BDDs have been used in electronic circuit generation[CBM90], verification, symbolic model checking[BCM+92], and type system models such as in XDuce [HVP05]. None of these sources discusses how to extend the BDD representation to support subtypes.

Decision tree techniques are useful in the efficient compilation of pattern matching constructs in functional languages[Mar08]. An important concern in pattern matching compilation is finding the best ordering of the variables which is known to be NP-hard. However, when using BDDs to represent Common Lisp type specifiers, we obtain representation (pointer) equality, simply by using a consistent ordering; finding the *best* ordering is not necessary for our application.

8. CONCLUSION AND FUTURE WORK

The results of the performance testing in Section 6 lead us to believe that the BDD as data structure for representing Common Lisp type specifiers is promising, but there is still work to do, especially in identifying heuristics to predict its performance relative to more traditional approaches.

It is known that algorithms using BDD data structure tend to trade space for speed. Castagna [Cas16] suggests a lazy version of the BDD data structure which may reduce the memory footprint, which would have a positive effect on the BDD based algorithms. We have spent only a few weeks optimizing our BDD implementation based on the Ander-

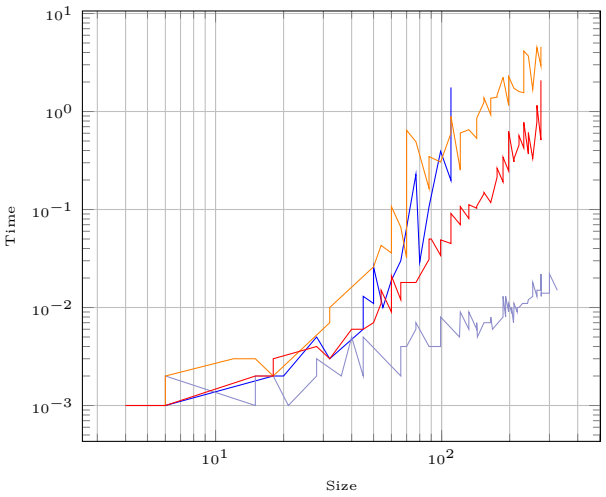


Figure 9: Subtypes of fixnum

sen's description [And99], whereas the CUDD [Som] developers have spent many years of research optimizing their algorithms. Certainly our BDD algorithm can be made more efficient using techniques of CUDD or others.

Although, we do not attempt, in this paper, to motivate in detail the applications or implications of MDTD, we suspect there may be a connection between the problem, and efficient compilation of `type-case` and its use in improving pattern matching capabilities of Common Lisp. We consider such development and motivation a matter of future research.

An immediate priority in our research is to formally prove the correctness of our algorithms, most notably the graph decomposition algorithm from Section 2.2. Experimentation leads us to believe that the graph algorithm always terminates with the correct answer, nevertheless we admit there may be exotic cases which cause deadlock or other errors.

It has also been observed that in the algorithm explained in section 2.2 that the convergence rate varies depending on the order the reduction operations are performed. We do not yet have enough data to characterize this dependence. Furthermore, the order to break connections in the algorithm in Section 2.2. It is clear that many different strategies are possible, (1) break busiest connections first, (2) break connections with the fewest dependencies, (3) random order, (4) closest to top of tree, etc. These are all areas of ongoing research.

We plan to investigate whether there are other applications MDTD outside the Common Lisp type system. We hope the user of Castagna's techniques [Cas16] on type systems with semantic subtyping may benefit from the optimizations we have discussed.

A potential application with Common Lisp is improving the `subtypep` implementation itself, which is known to be slow in some cases. Section 5 gave a BDD specific implementation of `bdd-subtypep`. We intend to investigate whether existing Common Lisp implementations could use our technique to represent type specifiers in their inferencing engines, and thereby make some subtype checks more efficient.

9. REFERENCES

[Ake78] S. B. Akers. Binary Decision Diagrams. IEEE Trans. Comput., 27(6):509–516, June 1978.

[And99] Henrik Reif Andersen. An introduction to binary decision diagrams. Technical report, Course Notes on the WWW, 1999.

[Ans94] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[AS96] Harold Abelson and Gerald J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[Bak92] Henry G. Baker. A Decision Procedure for Common Lisp's SUBTYPEP Predicate. Lisp and Symbolic Computation, 5(3):157–190, 1992.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 1020 States and Beyond. Inf. Comput., 98(2):142–170, June 1992.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, 35:677–691, August 1986.

[Cas16] Giuseppe Castagna. Covariance and Contravariance: a fresh look at an old issue. Technical report, CNRS, 2016.

[CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of Synchronous Sequential Machines Based on Symbolic Execution. In Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, pages 365–373, London, UK, UK, 1990. Springer-Verlag.

[CF05] Giuseppe Castagna and Alain Frisch. A Gentle Introduction to Semantic Subtyping. In Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '05, pages 198–199, New York, NY, USA, 2005. ACM.

[GF64] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. Communication of the ACM, 7(5):301–303, may 1964.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. ACM Trans. Program. Lang. Syst., 27(1):46–90, January 2005.

[Mar08] Luc Maranget. Compiling Pattern Matching to Good Decision Trees. In Proceedings of the 2008 ACM SIGPLAN Workshop on ML, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[NDV16] Jim Newton, Akim Demaille, and Didier Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In European Lisp Symposium, Kraków, Poland, May 2016.

[New17] Jim Newton. Analysis of Algorithms

Calculating the Maximal Disjoint Decomposition of a Set. Technical report, EPITA/LRDE, 2017.

[Nor92] Peter Norvig. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[PBM10] Md. Mostofa Ali Patwary, Jean R. S. Blair, and Fredrik Manne. Experiments on union-find algorithms for the disjoint-set data structure. In Paola Festa, editor, Proceedings of 9th International Symposium on Experimental Algorithms (SEA'10), volume 6049 of Lecture Notes in Computer Science, pages 411–423. Springer, 2010.

[Som] Fabio Somenzi. CUDD: BDD package, University of Colorado, Boulder.