

Analysis of Algorithms Calculating the Maximal Disjoint Decomposition of a Set

Jim Newton

November 7, 2017

Abstract

In this article we demonstrate 4 algorithms for calculating the maximal disjoint decomposition of a given set of types. We discuss some advantages and disadvantages of each, and compare their performance. We extended currently known work to describe an efficient algorithm for manipulating binary decision diagrams representing types in a programming language which supports subtyping viewed as subsets.

1 Introduction

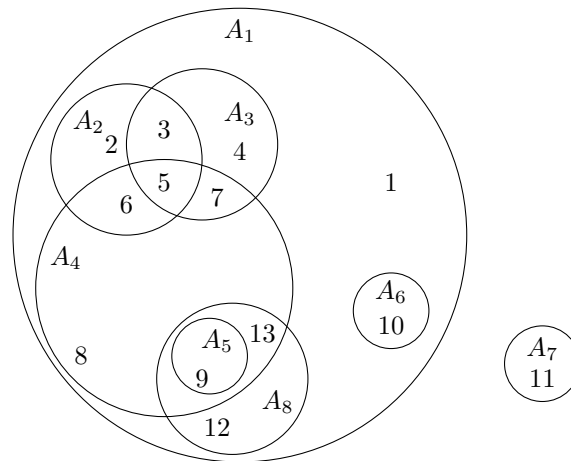


Figure 1: Example Venn Diagram

The problem we examine in this article is that of using Boolean operations to decompose a set of partially overlapping regions into a valid partition. In particular, given $V = \{A_1, A_2, \dots, A_M\}$, 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 . We define precisely what we mean by maximal disjoint decomposition in Definition 10 of Section 3.4.

An illustration should help give an intuition of the problem. The Venn diagram in Figure 1 is an example for $V = \{A_1, A_2, \dots, A_8\}$. The maximal disjoint decomposition $D = \{X_1, X_2, \dots, X_{13}\}$ of V in Figure 1. D is the largest possible set of pairwise disjoint subsets of $\bigcup V$, for which ever element thereof can be expressed as a Boolean combination of elements of V . *I.e.* the largest possible disjointed subset of \widehat{V} whose union is $\bigcup V$.

Solving this when you are permitted to look into the sets has been referred to as *union find* [PBM10, GF64]. However, we wish to solve the problem without knowledge of the specific elements; *i.e.* we are not permitted to iterate over or visit the individual elements. Rather, we have

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}	$A_4 \cap A_8 \cap \overline{A_5}$

Table 1: Disjoint Decomposition of Sets from Figure 1

knowledge of the subset, superset, disjoint-ness relations between any pair of sets. The correspondence of types to sets and subtypes to subsets thereof is also treated extensively in the theory of semantic subtyping [CF05].

1.1 Why study this problem?

Newton *et al.* [NDV16] presented this problem when attempting to determinize automata used to recognize rational type expressions.

Another potential application of this problem, which is still an open area of active research, is the problem of re-ordering clauses in a `typecase` in Common Lisp, or similar constructs in other programming languages. The property to note is that given an expression such as the following:

```
(typecase object
  (number ...)
  (symbol ...)
  (array ...)
  ...)
```

The clauses can be freely reordered provided the types are disjoint. Re-ordering the clauses is potentially advantageous for computational efficiency if the type which is most likely to occur at run-time appears first. Another reason to reorder is so that types can be simplified. Consider the following:

```
(typecase object
  ((and number (not integer)) E1)
  (integer E2)
  (array E3)
  ...)
```

The clauses of this `typecase` cannot be simplified, but we are allowed to swap the first and second clause the type yielding E1 can be simplified as follows.

```
(typecase object
  (integer E2)
  (number E1)
  (array E3)
  ...)
```

Thus it may be interesting to compute a disjoint type decomposition in order to express

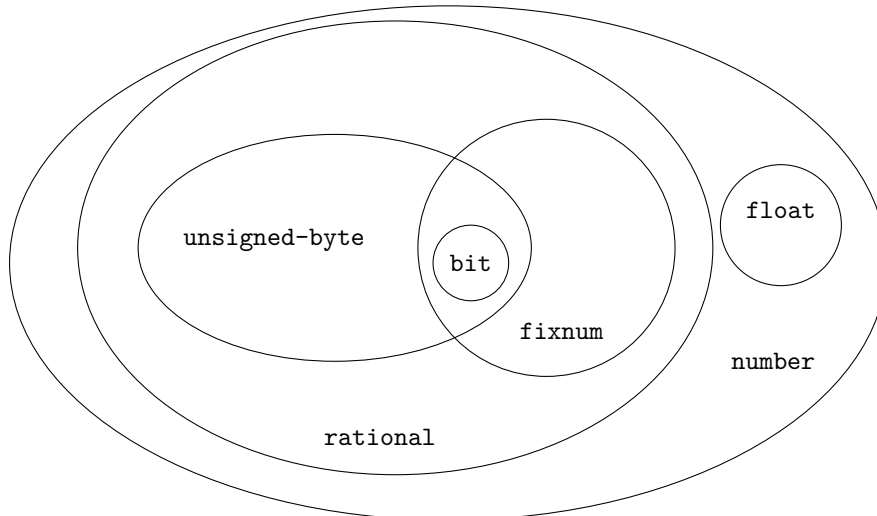


Figure 2: Example of some CL types

intermediate forms, which are thereafter simplified to more efficient carefully ordered clauses of intersecting types.

Finally, still another reason for studying this problem is because it allows us to examine lower level algorithms and data structures, the specifics of which may themselves have consequences which outweigh the type disjunction problem itself. There are several such cases in this technical report: including techniques and practices for using BDDs to represent lisp type specifiers (section 7) and techniques for optimizing programs characterized by heavy use of `subtypep` (section 9).

1.2 A Common Lisp based solution

As mentioned in Section 3.5, in the Common Lisp programming language, a *type* can also be thought of as a set of (potential) values [Ans94, Section Type]. Moreover, the language defines several *built-in* types and several subtype and disjoint relations between some of the types. Figure 2 illustrates a few of the subtypes of `number`. As the illustration shows, Common Lisp types are sets, and many intuitions about sets applies directly to Common Lisp types. Subtype relations correspond to subset relations. Intersecting types correspond to intersecting sets. Disjoint types are disjoint sets. As shown in the illustration `fixnum` is a subtype (subset) of `rational`; `unsigned-byte` intersects `fixnum`; `float` and `fixnum` are disjoint.

For such an algorithm to calculate the maximal disjoint decomposition as is explained in this paper to work, we must have operators to test for type-equality, type disjoint-ness, subtype-ness, and type-emptiness. It turns out that given a *subtype* predicate, a way to express a type intersection, and a way to express the empty type, the other predicates can be constructed. The emptiness check: $A = \emptyset \iff A \subset \emptyset$. The disjoint check: $A \perp B \iff A \cap B \subset \emptyset$. Type equivalence $A = B \iff A \subset B$ and $B \subset A$.

The Common Lisp language has a flexible type calculus which makes type related computation possible. If `T1` and `T2` are Common Lisp type specifiers, then the type specifier `(and T1 T2)` designates the intersection of the types. Likewise `(and T1 (not T2))` and `(and (not T1) T2)` are the two type differences. Furthermore, the Common Lisp function `subtypep` can be used as the subtype predicate, and `nil` designates the empty type. Consequently `subtypep` can be used to decide whether two designated types are in a subtype relation, or whether the two types are disjoint.

There is an important caveat. The `subtypep` function is not always able to determine whether the named types have a subtype relationship or not [Bak92, New16]. 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.

The remainder of this article proceeds as follows: Section 2 summarizes programmatic manipulation of Common Lisp type specifiers. Section 4 summarizes a simple easy to understand, easy to implement algorithm; Section 5 summarizes an algorithm based on a connectivity graph; Section 6 discusses an alternate solution by viewing this problem as a variant of the SAT problem; Section 7 discusses solving the problem with the aid of the binary decision diagram (BDD) data structure; and Section 9 discusses the performance of the various algorithms.

2 Type specifier manipulation

The Common Lisp language represents type specifiers not as opaque data structures with well defined APIs but rather as s-expressions. An s-expression such as `(and number (not (or bignum bit)))` specifies a valid type. You may think of types as sets. In fact the Common Lisp specification defines them as such. In particular if *number* indicates the set of all Common Lisp numbers, *bignum* as the set of all Common Lisp bignums, and *bit* as the set of all the two objects, 0 and 1, then the type specified by `(and number (not (or bignum bit)))` indicates the set $number \cap \overline{bignum \cup bit}$.

There are a few notable areas of caution.

- `t` and `nil` represent respectively the universal type and the empty type.
- The Common Lisp specification defines what a valid type specifier is, but it does not explain what an invalid type specifier is. This causes confusion in several cases. In particular it is unclear whether it is possible to determine whether a given s-expression is a type specifier. For example, if there is no such type named `xyzy`, then is `(or xyzy number)` even a type specifier. Some might say that it is a type specifier, just not a valid one. Others might say that it is not a type specifier at all. A related question is compliant behavior of evaluating `(subtypep t (gensym))`?
- Given a unary Boolean valued function `F`, the type specified by `(satisfies F)` is the set of Common Lisp objects for which `F` returns non-`nil`. However it is not specified as to what the meaning of `(not (satisfies F))` is. In particular the function `oddp` triggers an error if its argument is not an integer. Thus one should logically conclude that `3.4` is not an element of `(satisfies oddp)`, as `(oddp 3.4)` does not return non-`nil`. It is not clear whether `3.4` is an element of `(not (satisfies oddp))`.
- Given a function that never returns, *e.g.* `loops-forever`, it is not clear whether `3.4` is an element of `(satisfies loops-forever)` or of `(not (satisfies loops-forever))`.
- The `subtypep` function is allowed to return `nil, nil` in many situations which cause grief for the application programmer. For example, the Common Lisp specification allows `subtypep` to return `nil, nil` whenever at least one argument involves one of these type specifiers: `and`, `eql`, the list form of `function`, `member`, `not`, `or`, `satisfies`, or `values`.

2.1 S-expression manipulation

Even with certain limitations, s-expressions are an intuitive data structure for programmatic manipulation of type specifiers. Given two type specifiers, we may compute type specifiers representing their intersection, union, and relative complement simply by familiar s-expression manipulation.

```

(defun type-intersection (t1 t2)
  '(and ,t1 ,t2))

(defun type-union (t1 t2)
  '(or ,t1 ,t2))

(defun type-relative-complement (t1 t2)
  '(and ,t1 (not ,t2)))

```

Another example comes in analyzing and reasoning about type specifiers. The sbcl implementation of `subtypep` has several notable limitations: (`subtypep` `'(member :x :y)` `'keyword`) returns `nil, nil`, whereas it should return `t, t`. This is in-keeping with the specification because one of the arguments involves `member`. However, the user can implement a smarter version of `subtypep` to handle this case. Regrettably, the user cannot force the system to use this smarter version internally.

```

(deftype not-type-spec ()
  "Type spec of the form (not type)"
  '(cons (eql not)))

(defun smarter-subtypep (t1 t2)
  (multiple-value-bind (T1<=T2 OK) (subtypep t1 t2)
    (cond
      (OK
       (values T1<=T2 t))
      ((typep t1 '(cons (member eql member))) ; (eql obj) or (member obj1 ...)
       (values (every #'(lambda (obj)
                          (declare (notinline typep))
                          (typep obj t2)))
               (cdr t1))
              t))
      ;; T1 <: T2 ==> not(T2) <: not(T1)
      ((and (typep t1 'not-type-spec)
            (typep t2 'not-type-spec))
       (smarter-subtypep (cadr t2) (cadr t1)))
      ;; T1 <: T2 ==> not( T1 <= not(T2))
      ((and (typep t2 'not-type-spec)
            (smarter-subtypep t1 (cadr t2)))
       (values nil t))
      ;; T1 <: T2 ==> not( not(T1) <= T2)
      ((and (typep t1 'not-type-spec)
            (smarter-subtypep (cadr t1) t2))
       (values nil t))
      (t
       (values nil nil))))))

```

2.2 Boolean algebra

After applying the operations several times as shown in Section 2.1 it is not uncommon to result in type specification which are no longer easily human readable. Such as:

```
(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 type specifier actually may be reduced simply to `(and number (not bignum) (not fixnum))`. Not only is this an improvement in human readability, but it also not surprisingly allows calls to `typep` and `subtypep` execute more efficiently.

A Common Lisp function which does such type reduction may use a recursive descent reduction function to re-write sub expressions, incrementally moving the expression toward a canonical form. We've chosen to convert the expression to a *sum of minterms* form. This means it is an *OR* of *ANDs* of leaf level types specifiers and complements thereof, where none of the leaf level expression involve *OR* nor *AND*. Below are some examples of the cases of transformation which take place.

- Reductions of AND expressions.

```
- (and (and a b) x y) -> (and a b x y)
- (and x) -> x
- (and) -> t
- (and nil x y) -> nil
- (and number float x ) -> (and float x)
- (and float string) -> nil
- (and A (or x y) B) -> (or (and A B x) (and A B y))
```

- Reductions of OR expressions.

```
- (or) -> nil
- (or A) -> A
- (or A t B) -> t
- (or A nil B) -> (or A B)
- (or string (member 1 2 3) (eql 4) (member 2 5 6)) -> (or string (member 1
  2 3 4 5 6))
- (or fixnum string (member 1 2 "hello" a b)) -> (or fixnum string (member
  a b))
- (or number (not (member 1 2 a b))) -> (or number (not (member a b)))
- (or number (not (member a b))) -> (not (member a b))
- (or number (not number)) -> t
- (or A (and A B C D) E) -> (or A E)
- (or (and A B) (and A B C D) E F) -> (or (and A B) E F)
- (or A (and A (not B))) -> (or A B)
```

- Reductions of NOT expressions.

```
- (not nil) -> t
- (not t) -> nil
```

```

- (not atom) -> (not atom)
- (not (not A)) -> A
- (not (or A B C)) -> (and (not A) (not B) (not C))
- (not (and A B C)) -> (or (not A) (not B) (not C))

```

- Reductions of MEMBER and EQL expressions.

```

- (and (member a b 2 3) symbol) -> (member a b)
- (and (member a 2) symbol) -> (eql a)
- (and (member a b) fixnum) -> nil
- (and X Y (not (member 1 2 a)) (not (member 2 3 4 b))) -> (and X Y (not (member
  1 2 3 4 a b)))

```

- Consensus terms

```

- (or (and A B) (and A (not C)) (and B C)) -> (or (and A B) (and A (not B)))
- (or (and A B U) (and A (not C) U) (and B C U)) -> (or (and A B U) (and A
  (not C) U))

```

The function handling the reduction should call itself recursively on each argument of AND, OR, and NOT either before or after applying any case whose pattern matches. Moreover, this entire process must repeat multiple times; in fact it must repeat until a fixed point is found; *i.e.* it must repeat until the expression no longer changes with an additional application of the function.

Here is an example step by step reduction of the expression given above:

1. (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)))))
          (not (or fixnum (or bit (eql -1)))))
      )
      
```
2. (or


```

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


```

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

4. `(or
 (and number (not bignum) (not fixnum))
 (and (and (and number (not bignum))
 (not (or fixnum (or bit (eql -1))))))
 (not (or fixnum (or bit (eql -1)))))`
5. `(or
 (and number (not bignum) (not fixnum))
 (and (and number (not bignum) (not fixnum))
 (not (or fixnum (or bit (eql -1)))))`
6. `(or
 (and number (not bignum) (not fixnum))
 (and (and number (not bignum) (not fixnum))
 (not fixnum)))`
7. `(and number (not bignum) (not fixnum))`

3 Rigorous Development

In this section we define exactly what we mean by the term *maximal disjoint decomposition*. The main result of this section is Theorem 5 which claims the existence and uniqueness of the maximal disjoint decomposition. The reader who does not care about the rigorous treatment may skip most of this section, as long as he grasps the definition of *maximal disjoint decomposition* (Definition 10) and the claims of its existence and uniqueness (Theorem 5).

The presentation order used in this section is bottom-up; *i.e.*, we attempt to define and prove everything needed before it is actually used. This order, may cause some difficulty to the reader as it may not be clear at each point why something is being introduced. We attempt to alleviate some of this problem by providing motivational discussions as a prelude to each section and by giving lots of examples, so that even if the reader does not foresee how something will be used later, exactly why it is being presented, at least the reader can get a rigorous definition but also an intuitive feeling of the concept.

Some of the results presented in this section are also useful in Sections 4.2 and 5.5 where we argue for the correctness of the algorithms presented in Sections 4.1 and 5.1 respectively.

3.1 Partitions and Covers

In this report we often refer to the relation of *disjoint-ness* or *intersection* between two sets. For this reason we introduce a notation which some reader may find non-standard.

Notation 1. We use the symbol, \perp , to indicate the disjoint relation between sets. In particular we take $A \perp B$ to mean $A \cap B = \emptyset$. We also say $A \not\perp B$ to mean $A \cap B \neq \emptyset$.

Example 1. $\{1, 3, 5\} \perp \{2, 4\}$, and $\{1, 3, 5\} \not\perp \{2, 3\}$.

Definition 1. Let $D = \{X_1, X_2, \dots, X_M\}$, with $X_i \subset U$ for $1 \leq i \leq M$. If $X_i \neq X_j \implies X_i \perp X_j$, then D is said to be disjoined in U . *I.e.*, a disjoined set is a set of mutually disjoint subsets of a given U .

Example 2. The set $\{\{1, 3, 5\}, \{0, 2\}, \{4, 6\}, \emptyset\}$ is disjoined because its elements are sets, none of which have a common element. By contrast, the set $\{\{1, 2, 3\}, \{2, 4\}\}$ is not disjoined as 2 is common to $\{1, 2, 3\}$ and $\{2, 4\}$.

Notation 2. We denote the cardinality of set A , *i.e.* the number of elements, by $|A|$. We say that an infinite set has infinite cardinality; otherwise it has finite cardinality.

Example 3. $|\emptyset| = 0$.

Example 4. If $A = \{X, Y\}$ then $1 \leq |A| \leq 2$ because A is not empty, and it might be that $X = Y$.

Definition 2. If V is a set of subsets of U , then we define the unary \bigcup operator as follows:

$$\bigcup V = \begin{cases} \emptyset & \text{if } V = \emptyset \\ X & \text{if } |V| = 1 \text{ and } V = \{X\} \\ \bigcup_{X \in V} X & \text{if } |V| > 1 \end{cases}$$

Definition 3. If V is a set of subsets of U , then we define the unary \bigcap operator as follows:

$$\bigcap V = \begin{cases} U & \text{if } V = \emptyset \\ X & \text{if } |V| = 1 \text{ and } V = \{X\} \\ \bigcap_{X \in V} X & \text{if } |V| > 1 \end{cases}$$

Example 5. Let $V = \{\{1\}, \{1, 2\}, \{1, 2, 4\}, \{1, 3, 4, 5\}\}$, then

$$\bigcap V = \{1\} \cap \{1, 2\} \cap \{1, 2, 4\} \cap \{1, 3, 4, 5\} = \{1\}$$

and

$$\bigcup V = \{1\} \cup \{1, 2\} \cup \{1, 2, 4\} \cup \{1, 3, 4, 5\} = \{1, 2, 3, 4, 5\}$$

The values of $\bigcup V$ and $\bigcap V$ in the cases where $|V| = 0$ and $|V| = 1$ are defined as such so that the notation is consistent. In particular, the following identities hold:

$$\begin{aligned} \bigcup(V \cup V') &= (\bigcup V) \cup (\bigcup V') \\ \bigcap(V \cup V') &= (\bigcap V) \cup (\bigcap V') \end{aligned}$$

A more extensive treatment of these identities can be found in Appendix A.

Note that the definitions of $\bigcup V$ and $\bigcap V$ in no way make a claim or supposition about the cardinality of V . We may use the same notation whether V is infinite or finite.

Definition 4. A partition of a set V is a disjoint set P with the property that $\bigcup P = V$. A disjoint set is said to partition its union.

Example 6. $D = \{\{0\}, \{2, 4, 6, 8\}, \{1, 3, 5, 7, 9\}\}$ is a partition of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, because the three sets (the elements of D) are mutually disjoint and

$$\bigcup D = \{0\} \cup \{2, 4, 6, 8\} \cup \{1, 3, 5, 7, 9\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Example 7. The set $V = \{\{1\}, \{1, 2\}, \{2, 4\}, \{3, 4, 5\}\}$ from Example 5 is not a partition of $\{1, 2, 3, 4, 5\}$ because the elements of V are not mutually disjoint. However, V does cover $\{1, 2, 3, 4, 5\}$. The definition of *cover* is given below (Definition 5).

Notation 3. By 2^U we denote the power set of U , *i.e.* the set of subsets of U . Consequently we may take $V \subset 2^U$ to mean that V is a set of subsets of U .

Example 8. If $U = \{1, 2, 3\}$, then $2^U = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Notice that $|U| = 3$ and $|2^U| = 2^3 = 8$. It holds in general that $|2^U| = 2^{|U|}$.

Definition 5. If $V \subset 2^U$ and $C \subset 2^U$, C is said to be a cover of V , or equivalently we say that C *covers* V , provided $\bigcup V \subset \bigcup C$. Furthermore, if $\bigcup V = \bigcup C$ we say that C is an exact cover of V or that C *exactly covers* V .

Example 9. If $U = \{1, 2, 3, 4\}$ then $V = \{\{1, 2, 3\}, \{2, 4, 6\}, \emptyset\}$ covers U . However, V is not an exact cover of U because $6 \in \bigcup V$ but $6 \notin U$, *i.e.* $\bigcup V \not\subset U$.

3.2 The Boolean Closure

The purpose of the next several definitions is to define the set \widehat{V} which is intended to be the set of all Boolean combinations of sets in a given set V . \widehat{V} is defined in Definition 8, and is proved to exist and be unique in Corollary 1.

Lemma 1. *If V is a set of subsets of U , and $\exists X_0 \in V$ such that $X \in V \implies X_0 \subset X$, then $X_0 = \bigcap V$.*

Proof. $X \in V \implies X_0 \subset X$, therefore $X_0 \subset \bigcap V$.

Case 1: $(\bigcap V) \setminus X_0 = \emptyset$, therefore $\bigcap V \subset X_0$

Case 2: $(\bigcap V) \setminus X_0 \neq \emptyset$, so let $\alpha \in (\bigcap V) \setminus X_0$. This means $\alpha \notin X_0$, rather $\alpha \in \bigcap V$. $\alpha \in \bigcap V$ means that $\forall X \in V, \alpha \in X$, which is a contradiction because $X_0 \in V$.

□

Intuitively, Lemma 1 says that given a set of subsets, if one of those subsets happens to be a subset of all the given subsets, then it is in fact the intersection of all the subsets.

Example 10. Let $V = \{\emptyset, \{1\}, \{2\}\}$. Notice that there is an X_0 , namely $X_0 = \emptyset$, which has the property that it is a subset of every element of V , i.e., $X \in V \implies X_0 \subset X$. Therefore, $\bigcap V = X_0 = \emptyset$.

Example 11. Let $V = \{\{1\}, \{1, 2\}, \{1, 2, 3\}, \{1, 2, 3, 4\}, \dots, \{1, 2, 3, 4, \dots, N\}\}$ for some N . Notice that there is an X_0 , namely $X_0 = \{1\}$, which has the property that it is a subset of every element of V , i.e., $\{1\}$ is a subset of every element of V , or $X \in V \implies X_0 \subset X$. Therefore, $\bigcap V = \{1\}$.

Definition 6. Let $V \subset U$, and let F be a set of binary functions mapping $U \times U \mapsto U$. A superset $V' \supset V$ is said to be a closed superset of V under F if $\alpha, \beta \in V'$ and $f \in F \implies f(\alpha, \beta) \in V'$.

Example 12. If $V = \{1\}$, and $F = \{+\}$, then the set of integers \mathbb{Z} is closed superset of V under F . Why, because $V \subset \mathbb{Z}$, and $\alpha, \beta \in \mathbb{Z} \implies \alpha + \beta \in \mathbb{Z}$.

Definition 7. Let $V \subset U$, and let F be a set of binary functions mapping $U \times U \mapsto U$. The closure of V under F , denoted $clos_F(V)$, is the intersection of all closed supersets of V under F .

Example 13. If $V = \{1\}$, and $F = \{+\}$, then $clos_F(V)$ is the set of positive integers, \mathbb{N} .

Proof. To show this we argue that \mathbb{N} is a closed superset of V and that if V' is a closed superset of V then $\mathbb{N} \subset V'$.

$1 \in \mathbb{N}$ so $V \subset \mathbb{N}$. If $\alpha, \beta \in \mathbb{N}$, then $\alpha + \beta \in \mathbb{N}$, so \mathbb{N} is closed.

Let V' be a closed superset of V . $\mathbb{N} \subset V'$ can be shown by induction. $1 \in V'$ because $1 \in V \subset V'$. Now assume $k \in V'$. Is $k + 1 \in V'$? Yes, because $\{k\} \cup \{1\} = \{k, 1\} \subset V' \implies k + 1 \in V'$. Therefore $\mathbb{N} \subset V'$.

So by Lemma 1, $\mathbb{N} = clos_F(V)$. □

Theorem 1. *If $V \subset U$, and if F is a set of binary functions defined on U , then there exists a unique $clos_F(V)$.*

Proof. Existence: Define a sequence $\{\Phi_n\}_{n=0}^\infty$ of sets as follows:

- $\Phi_0 = V$
- If $i > 0$, then $\Phi_i = \Phi_{i-1} \cup \bigcup_{f \in F} \{f(x, y) \mid x \in \Phi_{i-1}, y \in \Phi_{i-1}\}$

Define the set $\Phi = \bigcup_{i=0}^{\infty} \Phi_i$. We know that $V = \Phi_0 \subset \bigcup_{i=0}^{\infty} \Phi_i$. Next, let $\alpha \in \Phi$, $\beta \in \Phi$, $f \in F$; take $n \geq 0$ such that $\alpha \in \Phi_n$ and $\beta \in \Phi_n$. By definition $f(\alpha, \beta) \in \Phi_{n+1} \subset \Phi$. Thus Φ satisfies the definition of $\text{clos}_F(V)$.

Uniqueness: Suppose Ψ is the set of all sets satisfying the definition of $\text{clos}_F(V)$. Set $\Phi = \bigcap \Psi$. We will show that Φ satisfies the definition of $\text{clos}_F(V)$. V is a subset of every element of Ψ so $V \subset \bigcap \Psi = \Phi$. Now, take $\alpha \in \Phi$, $\beta \in \Phi$, and $f \in F$. Since $\alpha \in \Phi$, that means α is in every element of Ψ , similarly for β , so $f(\alpha, \beta)$ is an element of every element of Ψ , which means $f(\alpha, \beta) \in \bigcap \Psi = \Phi$. Therefore, Φ is uniquely defined and satisfies the definition of $\text{clos}_F(V)$. \square

α	β	$\alpha \cup \beta$	$\alpha \cap \beta$
\emptyset	\emptyset	\emptyset	\emptyset
\emptyset	$\{1, 2\}$	$\{1, 2\}$	\emptyset
\emptyset	$\{2\}$	$\{2\}$	\emptyset
\emptyset	$\{2, 3\}$	$\{2, 3\}$	\emptyset
\emptyset	$\{1, 2, 3\}$	$\{1, 2, 3\}$	\emptyset
$\{1, 2\}$	\emptyset	$\{1, 2\}$	\emptyset
$\{1, 2\}$	$\{1, 2\}$	$\{1, 2\}$	$\{1, 2\}$
$\{1, 2\}$	$\{2\}$	$\{1, 2\}$	$\{2\}$
$\{1, 2\}$	$\{2, 3\}$	$\{1, 2, 3\}$	$\{2\}$
$\{1, 2\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{1, 2\}$
$\{2\}$	\emptyset	$\{2\}$	\emptyset
$\{2\}$	$\{1, 2\}$	$\{1, 2\}$	$\{2\}$
$\{2\}$	$\{2\}$	$\{2\}$	$\{2\}$
$\{2\}$	$\{2, 3\}$	$\{2, 3\}$	$\{2\}$
$\{2\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2\}$
$\{2, 3\}$	\emptyset	$\{2, 3\}$	\emptyset
$\{2, 3\}$	$\{1, 2\}$	$\{1, 2, 3\}$	$\{2\}$
$\{2, 3\}$	$\{2\}$	$\{2, 3\}$	$\{2\}$
$\{2, 3\}$	$\{2, 3\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{1, 2, 3\}$	\emptyset	$\{1, 2, 3\}$	\emptyset
$\{1, 2, 3\}$	$\{1, 2\}$	$\{1, 2, 3\}$	$\{1, 2\}$
$\{1, 2, 3\}$	$\{2\}$	$\{1, 2, 3\}$	$\{2\}$
$\{1, 2, 3\}$	$\{2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$

Table 2: Closure under a set of operations

Example 14. Let $V = \{\{1, 2\}, \{2, 3\}\}$, and F be the set containing the set-union and set-intersection operations, denoted $F = \{\cup, \cap\}$. Then $\text{clos}_F(V) = \{\emptyset, \{1, 2\}, \{2\}, \{2, 3\}, \{1, 2, 3\}\}$, because if we take $\alpha, \beta \in \{\emptyset, \{1, 2\}, \{2\}, \{2, 3\}, \{1, 2, 3\}\}$ then both $\alpha \cup \beta$ and $\alpha \cap \beta$ are also therein. This can be verified exhaustively as in Table 2.

Any smaller set would not fulfill the definition of $\text{clos}_F(V)$. In particular if either or $\{1, 2\}$ or $\{2, 3\}$ were omitted, then it would no longer be a superset of V , and if any of $\{\emptyset, \{2\}, \{1, 2, 3\}\}$ were omitted, then it would no longer be a closed superset of V . Finally, if any element were added, such as $V' = V \cup \{3\}$, it would no longer fulfill the intersection requirement; *i.e.* $V' \not\subset V$ so V' is not the intersection of all closed supersets of V under F .

Definition 8. If $V \subset U$, and F is the set of three primitive set operations union, intersection, and relative complement, ($F = \{\cup, \cap, \setminus\}$) then we denote $\text{clos}_F(V)$ simply by \widehat{V} and call it the Boolean closure of V .

Corollary 1. *If $V \subset U$, the closure \widehat{V} exists and is unique.*

Proof. Simple application of Theorem 1. □

Example 15. Let $V = \{\{1, 2\}, \{2, 3\}\}$ as in Example 14.

$\widehat{V} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. I.e., $\widehat{V} = 2^{\{1, 2, 3\}}$.

3.3 Disjoint Decompositions

In this section we define the notion of *disjoint decomposition* along with some examples. We also present and prove two results, Lemma 2 which is used to prove Theorem 4, and Lemma 4 which is used to prove Lemma 2. Lemma 2 in particular sheds insight on how one might create an algorithm to improve or refine a given disjoint decomposition.

Notation 4. We use the symbol, \sqcup , to indicate union of mutually disjoint sets. Moreover, if we write $\sqcup X_n$ then we claim or emphasize that $i \neq j \implies X_i \perp X_j$.

Theorem 2. *If $V \subset 2^U$ with $|V| < \infty$, a disjoint decomposition of V has finite cardinality.*

Proof. A disjoint decomposition of V is a set of Boolean combinations of elements of V . There are finitely many such Boolean combinations of elements of V , and any subset of a finite set is finite.

An argument that there are finitely many Boolean combinations of a finite set is presented in Appendix B. □

Theorem 3. *If $V \subset 2^U$ with $|V| < \infty$, then there are only finitely many disjoint decompositions.*

Proof. Each disjoint decomposition is a set of Boolean combinations of elements of V . The set of all Boolean combinations of elements of V is finite. And the set of subsets of a finite set is finite.

As stated in Theorem 2, an argument that there are finitely many Boolean combinations of a finite set is presented in Appendix B. □

Definition 9. Given a set of non-empty, possibly overlapping sets $V \subset 2^U$, set D is said to be a disjoint decomposition of V , if D is disjointed, $D \subset \widehat{V}$, and D exactly covers V .

Another way of thinking about Definition 9 is that if D is a disjoint decomposition of V , then D is a partition of $\bigcup V$, and that every element of D is a Boolean combination of elements of V . We will see in Theorem 4 that each of these Boolean combinations is an intersection with some element of V .

Example 16. Let $V = \{\{1, 2\}, \{2, 3\}\}$ as in Example 14. $D = \{\{1\}, \{2, 3\}\} \subset \widehat{V}$ is a disjoint decomposition because $\{1\} \perp \{2, 3\}$, and $\bigcup D = \{1\} \cup \{2, 3\} = \{1, 2\} \cup \{2, 3\} = \bigcup V$.

Lemma 2. *Suppose D is disjoint decomposition of V , and $X \in D$. If there exist distinct $\alpha, \beta \in \widehat{V}$, both different from \emptyset such that $\{\alpha, \beta\}$ is a disjoint decomposition of $\{X\}$, then $\{\alpha, \beta\} \cup D \setminus \{X\}$ is a disjoint decomposition of V with cardinality $|D| + 1$.*

Proof. First we show that both α and β are disjoint from all elements of $D \setminus X$. Proof by contradiction. Without loss of generality take $\emptyset \neq \gamma \in D \setminus X$ such that $\alpha \not\perp \gamma$. So $\emptyset \neq \alpha \cap \gamma \subset D \setminus X$. So $\alpha \cap \gamma \not\subset X$. But since $\alpha \subset X$ we also have $\alpha \cap \gamma \subset X$. Contradiction!

Since $\alpha \cup \beta = X$, $\bigcup D = \alpha \cup \beta \cup \bigcup D \setminus \{X\}$. Thus $\{\alpha, \beta\} \cup D \setminus \{X\}$ is a cover of V .

Now since α and β are disjoint from all elements of $D \setminus \{X\}$ and disjoint from each other, we know $\alpha \not\subset D \setminus X$ and $\beta \not\subset D \setminus X$, $|\{\alpha, \beta\} \cup D \setminus \{X\}| = |\{\alpha, \beta\}| + |D \setminus \{X\}| = 2 + |D| - 1 = |D| + 1$ □

A brief intuitive explanation of Lemma 2 may be useful. The lemma basically says that if we start with a disjoint decomposition, D but one of the elements, X , of that disjoint decomposition

can itself be decomposed into $\{\alpha, \beta\} \subset \widehat{V}$, then we can construct a *better* disjoint decomposition having exactly one additional element, simply by removing X and adding back α and β . *I.e.* starting with D , if X is a decomposable element of D , decomposable into $\{\alpha, \beta\}$, then a *better* disjoint decomposition is $\{\alpha, \beta\} \cup D \setminus \{X\}$.

Example 17. Let $V = \{\{1, 2\}, \{2, 3\}\}$ as in Example 14.

Recall that $\widehat{V} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

$D = \{\{1\}, \{2, 3\}\}$ is a disjoint decomposition of V . Let $X = \{2, 3\}$, $\alpha = \{2\}$, $\beta = \{3\}$. Notice that $\alpha = \{2\} \in \widehat{V}$ and $\beta = \{3\} \in \widehat{V}$, that $\{2\} \perp \{3\}$, and that $X = \{2, 3\} = \{2\} \cup \{3\} = \alpha \cup \beta$.

According to Lemma 2, $\{\alpha, \beta\} \cup D \setminus \{X\}$ is a disjoint decomposition of V with cardinality $|D|+1$. Is this true? Yes!

$$\begin{aligned} D' &= \{\alpha, \beta\} \cup D \setminus \{X\} \\ &= \{\{2\}, \{3\}\} \cup \{\{1\}, \{2, 3\}\} \setminus \{\{2, 3\}\} \\ &= \{\{1\}, \{2\}, \{3\}\}. \end{aligned}$$

Moreover, $|D| = 2$, whereas $|D'| = 3$.

Lemma 3. *If $A \subset U$ and $B \subset U$, with $|A| = |B| < \infty$ but $A \neq B$, then $A \setminus B \neq \emptyset$ and $B \setminus A \neq \emptyset$.*

Proof. Proof by contradiction: If $A \setminus B = B \setminus A = \emptyset$ then $A = B$. Contradiction! So without loss of generality assume $A \setminus B = \emptyset$, and $B \setminus A \neq \emptyset$. We have $|A \setminus B| = 0$, and $|B \setminus A| > 0$.

$$A = A \setminus B \cup A \cap B$$

$$A \setminus B \perp A \cap B, \text{ so } |A| = |A \setminus B \cup A \cap B| = |A \setminus B| + |A \cap B| = 0 + |A \cap B| = |A \cap B|.$$

$$B = B \setminus A \cup A \cap B.$$

$$B \setminus A \perp A \cap B, \text{ so } |B| = |B \setminus A \cup A \cap B| = |B \setminus A| + |A \cap B| > |A \cap B|$$

$$|A| = |B| > |A \cap B| = |A|. \text{ Contradiction!} \quad \square$$

Example 18. Let $A = \{1, 2, 3\}$, $B = \{1, 2, 4\}$. This A and B fulfill Lemma 3. In particular $|A| = |B| = 3 < \infty$, and $A \neq B$. The lemma claims that $A \setminus B \neq \emptyset$ and $B \setminus A \neq \emptyset$, and this is in fact the case. $A \setminus B = \{1, 2, 3\} \setminus \{1, 2, 4\} = \{3\} \neq \emptyset$. $B \setminus A = \{1, 2, 4\} \setminus \{1, 2, 3\} = \{4\} \neq \emptyset$.

Lemma 4. *If D_1 and D_2 are disjoint decompositions of V , such that $D_1 \neq D_2$ and $|D_1| = |D_2|$, then there exists a disjoint decomposition of V with higher cardinality.*

Proof. Denote $N = |D_1| = |D_2|$. Denote $D_1 = \{X_1, X_2, \dots, X_N\}$ and $D_2 = \{Y_1, Y_2, \dots, Y_N\}$. By Lemma 3 we can take $Y_j \in D_2$ such that $Y_j \notin D_1$, *i.e.* $Y_j \in D_2 \setminus D_1$. Since $Y_j \subset \bigsqcup D_2 = \bigsqcup D_1$, take $X_i \in D_1$ such that $Y_j \not\perp X_i$. $Y_j \not\perp X_i \implies Y_j \cap X_i \neq \emptyset$.

Case 1: $X_i \not\subset Y_j \implies X_i \setminus Y_j \neq \emptyset$. $X_i = X_i \setminus Y_j \cup X_i \cap Y_j$, with $X_i \setminus Y_j \perp X_i \cap Y_j$, so by Lemma 2, $\{X_i \setminus Y_j, X_i \cap Y_j\} \cup D_1 \setminus \{X_i\}$ is disjoint and has cardinality $N + 1$.

Case 2: $X_i = Y_j$. Impossible since $X_i \in D_1$ while $Y_j \in D_2 \setminus D_1$.

Case 3: $X_i \subsetneq Y_j \implies Y_j \setminus X_i \neq \emptyset$. $Y_j = Y_j \setminus X_i \cup X_i \cap Y_j$, with $Y_j \setminus X_i \perp X_i \cap Y_j$, so by Lemma 2, $\{Y_j \setminus X_i, X_i \cap Y_j\} \cup D_2 \setminus \{Y_j\}$ is disjoint and has cardinality $N + 1$. □

Example 19. Let $V = \{\{1, 2\}, \{2, 3\}\}$ as in Example 14. Let $D_1 = \{\{1\}, \{2, 3\}\}$ and $D_2 = \{\{1, 2\}, \{3\}\}$. Notice that both D_1 and D_2 are both disjoint decompositions of V , and they fulfill the assumptions of Lemma 4. In particular, $D_1 \neq D_2$ and $|D_1| = |D_2| = 2$.

The lemma claims that there is therefore a disjoint decomposition with cardinality 3. Moreover, the proof of Lemma 4 suggests a way to find such a disjoint decomposition. To do this we must

take $X \in D_1$ and $Y \in D_2$ such that $X \neq Y$, and $X \not\perp Y$.

$$\begin{aligned} \text{Let's take } X &= \{2, 3\} \in D_1 \\ \text{and } Y &= \{1, 2\} \in D_2. \end{aligned}$$

This choice suffices because $X \neq Y$ and $X \not\perp Y$.

We can now construct two sets D'_1 and D'_2 , and one of these or the other (or perhaps both) will be a disjoint decomposition of cardinality 3.

$$\begin{aligned} D'_1 &= \{X \setminus Y, X \cap Y\} \cup D_1 \setminus \{X\} \\ &= \{\{2, 3\} \setminus \{1, 2\}, \{2, 3\} \cap \{1, 2\}\} \cup \{\{1\}, \{2, 3\}\} \setminus \{\{2, 3\}\} \\ &= \{\{3\}, \{2\}\} \cup \{\{1\}\} \\ &= \{\{1\}, \{2\}, \{3\}\} \end{aligned}$$

$$\begin{aligned} D'_2 &= \{Y \setminus X, X \cap Y\} \cup D_2 \setminus \{Y\} \\ &= \{\{1, 2\} \setminus \{2, 3\}, \{2, 3\} \cap \{1, 2\}\} \cup \{\{1, 2\}, \{3\}\} \setminus \{\{1, 2\}\} \\ &= \{\{1\}, \{2\}\} \cup \{\{3\}\} \\ &= \{\{1\}, \{2\}, \{3\}\} \end{aligned}$$

We see, in this case, that both D'_1 and D'_2 are disjoint decompositions of cardinality 3.

It happens in this example that $D'_1 = D'_2$. This equality is, however, not an immediate consequence of Lemma 4.

3.4 Maximal Disjoint Decomposition

Definition 10. If D is a disjoint decomposition such that for any other disjoint decomposition D' it holds that $|D| > |D'|$, then D is said to be a maximal disjoint decomposition.

The intuition here is that the *maximal disjoint decomposition* of a given set V of subsets of U is the most thorough partitioning of $\bigcup V$ which is possible when we are only allowed to use Boolean combinations of the sets given in V . I.e., while it might be possible to conceive of a better partitioning (more complete) and in fact, it may not even be possible in general to find the most thorough partitioning in general, when restricted to using only Boolean combinations of elements of V , we can think of the most thorough such partitioning.

There is a potential confusion if we informally use a term such as *largest decomposition*. By *most thorough*, we mean the set, $D = \{X_1, X_2, \dots, X_n\}$ of subsets for which D has the maximum possible number of elements (maximizing n), not such that the X 's themselves has as many elements as possible. I.e., we want a large collection of small sets, rather than a small collection of large sets.

Example 20. Let $V = \{\{1, 2\}, \{2, 3, 4\}\}$; $\widehat{V} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}, \{3, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}\}$.
 $D = \{\emptyset, \{1\}, \{2\}, \{3, 4\}\}$ is the maximal disjoint decomposition of V .

Why? Clearly D is a disjoint decomposition of V . The only way to decompose further would be to consider the set $D' = \{\emptyset, \{1\}, \{2\}, \{3\}, \{4\}\}$. However, D' is not the maximal disjoint decomposition of V because $D' \not\subseteq \widehat{V}$. Notice that $\{3\} \notin \widehat{V}$ (and neither is $\{4\}$). There is no way to produce the set $\{3\}$ (nor $\{4\}$) starting with the elements of V and combining them finitely many times using intersection, union, and relative complement.

Theorem 4. If D is a maximal disjoint decomposition of V , then $\forall X \in D \exists A \in V$ such that $X \subset A$.

Proof. Proof by contradiction: Let D be a maximal disjoint decomposition of V . Since $\bigcup D = \bigcup V$, let $Y \in V$ such that $X \not\subseteq Y$. By contrary assumption $X \not\subseteq Y \implies X \setminus Y \neq \emptyset$. So $X = X \setminus Y \cup X \cap Y$, $X \setminus Y \perp X \cap Y$, so by Lemma 2, $\{X \setminus Y, X \cap Y\} \cup \bigcup D \setminus X$ is disjoint decomposition of V with greater cardinality than D , which is impossible since D was assumed to be a maximal disjoint decomposition. Contradiction! \square

Theorem 4 basically says that every element of a maximal disjoint decomposition of V is a subset of some element of V . This means that if we want to construct a maximal disjoint decomposition algorithmically, an interesting strategy might be to start with the elements of V and look at cleverly constructed intersections thereof. This is in fact the strategy we will apply in the algorithms explained in Section 4 and 5.

Example 21. As in Example 20, let $V = \{\{1, 2\}, \{2, 3, 4\}\}$. $D = \{\emptyset, \{1\}, \{2\}, \{3, 4\}\}$ is the maximal disjoint decomposition of V . For each $X \in D$ we can find an $A \in V$ such that $X \subset A$. Note, that A is not necessarily unique. In particular.

$$\begin{aligned} D \ni \emptyset &\subset \{1, 2\} \in V \\ D \ni \{1\} &\subset \{1, 2\} \in V \\ D \ni \{2\} &\subset \{2, 3, 4\} \in V \\ D \ni \{3, 4\} &\subset \{2, 3, 4\} \in V \end{aligned}$$

Theorem 5. *There exists a unique maximal disjoint decomposition.*

Proof. Let D^* be the set of all disjoint decompositions of V . D^* is a finite set by Theorem 3. Let C be the set of cardinalities of elements of D^* , each of which is finite by Theorem 2. C is a finite set of integers, let $M = \max(C)$. Let D_{\max} be the set of elements $X \in D^*$ such that $|X| = M$. We now show that $|D_{\max}| = 1$.

Case $|D_{\max}| = 0$: Impossible because there exists at least one decomposition, namely the trivial one: $\{\bigcup V\}$.

Case $|D_{\max}| \geq 2$: Impossible because if $\alpha, \beta \in D_{\max}, \alpha \neq \beta$, then by Lemma 4 there exists another disjoint decomposition, γ such that $|\gamma| > |\alpha|$, which would mean that α is not maximal.

Case $|D_{\max}| = 1$: Since $|D_{\max}|$ cannot be negative, 0, nor greater than 1, it must be equal to 1.

Thus there is exactly one disjoint decomposition whose cardinality is larger than any other disjoint decomposition. \square

3.5 Some definitions from the Common Lisp specification

The Common Lisp specification defines several terms which may in some cases be helpful and in some cases confusing with regard to the definitions given in Section 1. The terms we have already introduced, are defined in terms of general sets. In Common Lisp, types are defined in a way which is isomorphic to sets. Some of the Common Lisp terms describing types are analogous to definitions we have made in terms of sets. These definitions come verbatim from the Common Lisp specification. [Ans94, Glossary] More explanation is given in Section 1.2.

exhaustive partition: n. (of a type) a set of pairwise disjoint types that form an exhaustive union.

exhaustive union: n. (of a type) a set of subtypes of the type, whose union contains all elements of that type.

pairwise: adv. (of an adjective on a set) applying individually to all possible pairings of elements of the set. "The types A , B , and C are pairwise disjoint if A and B are disjoint, B and C are disjoint, and A and C are disjoint."

type:	n. <ol style="list-style-type: none"> 1. a set of objects, usually with common structure, behavior, or purpose. (Note that the expression “X is of type S_a” naturally implies that “X is of type S_b” if S_a is a subtype of S_b.) 2. (immediately following the name of a type) a subtype of that type. “The type vector is an array type.”
type equivalent:	adj. (of two types X and Y) having the same elements; that is, X is a subtype of Y and Y is a subtype of X .
type expand:	n. [sic] ¹ to fully expand a type specifier, removing any references to derived types. (Common Lisp provides no program interface to cause this to occur, but the semantics of Common Lisp are such that every implementation must be able to do this internally, and some situations involving type specifiers are most easily described in terms of a fully expanded type specifier.)
type specifier:	n. an expression that denotes a type. “The symbol <code>random-state</code> , the list <code>(integer 3 5)</code> , the list <code>(and list (not null))</code> , and the class named <code>standard-class</code> are type specifiers.”

In terms of these definitions from Common Lisp, we may think of a disjoint decomposition of $V \subset 2^U$ as an exhaustive partition of $\bigcap V$. The distinction between the two concepts is subtle. To find an exhaustive partition, we start with a single set and partition it into disjoint subsets whose union is the original set. To find a disjoint decomposition, we start with a set of possibly overlapping subsets of a given set, whose union is not necessarily the entire site, and we proceed by finding another set of subsets which is pairwise disjoint and which has the same union as the given set of subsets.

4 Simple set disjoint decomposition

This section presents both a conceptually simple algorithm for calculating the maximal decomposition (Section 4.1), and thereafter argues for the correctness of the algorithm (Section 4.2). The algorithm suffers some performance problems which are addressed in Section 5.

4.1 Algorithm for set disjoint decomposition

The algorithm used in the Common Lisp package *regular-type-expressions*² [NDV16] for calculating the maximal disjoint decomposition is shown in Algorithm 1.³ This algorithm is straightforward and brute force. A notable feature of this algorithm is that it easily fits in 40 lines of Common Lisp code, so it is easy to implement and easy to understand, albeit not the most efficient possible

¹Type expand is a verb, but the specification contains what seems to be a typo claiming it to be a noun.

²<https://gitlab.lrde.epita.fr/jnewton/regular-type-expression>, The Common Lisp package source code is available from the EPITA/LRDE website.

³Many thanks to Dr. Alexandre Duret-Lutz for more than a few white board discussions to help me express these ideas.

in terms of run-time performance.

Algorithm 1: Finds the maximal disjoint decomposition

Input: A finite non-empty set U of sets
Output: A finite set V of disjoint sets

```

1.1  $V \leftarrow \emptyset$ 
1.2 begin
1.3   while true do
1.4      $V' \leftarrow \{u \in U \mid u' \in U \setminus \{u\} \implies u \perp u'\}$ 
1.5      $V \leftarrow V \cup V'$ 
1.6      $U \leftarrow U \setminus V'$ 
1.7     if  $U = \emptyset$  then
1.8       return  $V$ 
1.9     else
1.10      Find  $X \in U$  and  $Y \in U$  such that  $X \perp Y$ 
1.11      if  $X \subset Y$  then
1.12         $U \leftarrow U \setminus \{Y\} \cup \{Y \setminus X\}$ 
1.13      else if  $Y \subset X$  then
1.14         $U \leftarrow U \setminus \{X\} \cup \{X \setminus Y\}$ 
1.15      else
1.16         $U \leftarrow U \setminus \{X, Y\} \cup \{X \cap Y, X \setminus Y, Y \setminus X\}$ 
1.17   return  $V$ 

```

Notes about Algorithm 1:

Line 1.4: we find the set, V' of all elements of U which are disjoint from all other elements of U . Notice that in line 1.4, if U is a singleton set, then V' is that singleton element, thus $U \leftarrow \emptyset$ on line 1.6.

Line 1.4: This is of course an $\mathcal{O}(n^2)$ search, and what is worse, it is repeated each time through the loop headed on line 1.3. Part of the motivation of the algorithm in Section 5 is to eliminate this $\mathcal{O}(n^3)$ search.

Line 1.8: If $U = \emptyset$ then we have collected all the disjoint sets into V .

Line 1.10: this search is $\mathcal{O}(n^2)$.

Line 1.10: It is guaranteed that X and Y exist, because $|U| > 1$, and if all the elements of U were mutually disjoint, they would have all been collected in line 1.4.

Line 1.11: The case analysis here does the following: $U \leftarrow \{X \cap Y, X \setminus Y, Y \setminus X\} \cup U \setminus \{\emptyset\}$. However, some elements of $\{X \cap Y, X \setminus Y, Y \setminus X\}$ may be \emptyset or in fact X or Y depending on the subset relation between X and Y , thus the three cases specialize the possible subset relations.

Line 1.12: If $X \subset Y$, then $X \cap Y = X$ and $X \setminus Y = \emptyset$. Thus update U by removing Y , and adding $Y \setminus X$.

Line 1.14: If $Y \subset X$, then $X \cap Y = Y$ and $Y \setminus X = \emptyset$. Thus update U by removing X , and adding $X \setminus Y$.

Line 1.16: Otherwise, update U by removing X and Y , and adding $X \cap Y$, $X \setminus Y$, and $Y \setminus X$.

4.2 Correctness of the simple algorithm

5 Set disjoint decomposition as graph problem

One of the sources of inefficiency of the algorithm explained in Section 4 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 $X \perp A \cap B$, $X \perp A \setminus B$, $X \perp B \setminus A$. This knowledge eliminates some of useless operations.

TODO: need a proof that the algorithm terminates, and that it gets the correct answer.

5.1 The Algorithm

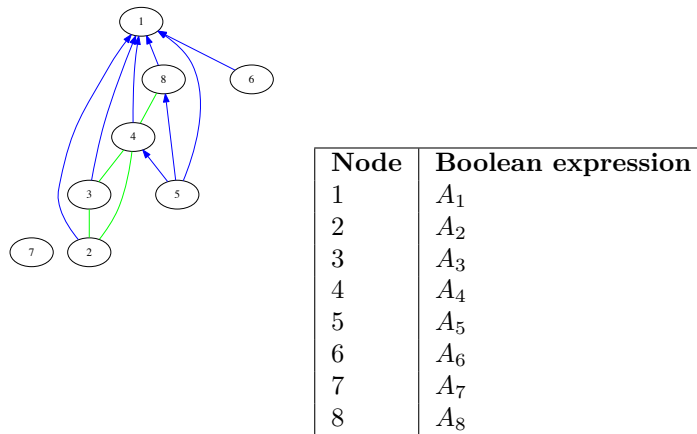


Figure 3: State 0: Topology graph

This algorithm is semantically very similar to the algorithm shown in Section 4 but rather than relying on Common Lisp primitives to make decisions about connectivity of sets/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 4.

This more complicated algorithm is presented here for two reasons. (1) It has much faster execution times, especially for larger sets types. (2) We hope that presenting the algorithm in a way which obviates the need to use Common Lisp primitives makes it evident how the algorithm might be implemented in a programming language other than Common Lisp.

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 X_1, X_2, \dots, X_8 in Figure 1.

5.2 Graph-based Disjoint Decomposition Algorithm

The algorithm commences by constructing a graph from a given set of subsets (Section 5.2.1), and proceeds by breaking the green and blue connections, one by one, in controlled ways until all the nodes become isolated. Sometimes it is necessary to temporarily create new connections which certain connections are broken as is seen below. There are a small number of cases to consider as are explained in detail in Sections 5.2.2, 5.2.3, 5.2.4, and 5.2.5. Repeat alternatively applying both tests until all the nodes become isolated.

There are several possible flows that variations of the algorithm. We start by two alternative

basic flows in Algorithms 2 and 3. We discuss variations of these flows in Section 9.2.

Algorithm 2: DECOMPOSEBYGRAPH-1

Input: U : A finite non-empty set of sets, *i.e.* U has no repeated elements and no empty elements.

Output: The maximal decomposition of U

```

2.1 begin
2.2    $G \leftarrow \text{ConstructGraph}(U)$ 
2.3   while  $G.\text{blue}$  or  $G.\text{green}$  do
2.4     for  $(X \rightarrow Y) \in G.\text{blue}$  do
2.5        $\text{BreakRelaxedSubset}(G, X, Y)$ ;
2.6     for  $\{X, Y\} \in G.\text{green}$  do
2.7        $\text{BreakTouching}(G, X, Y)$ ;
2.8   return  $G.\text{disjoint}$ 

```

Algorithm 3: DECOMPOSEBYGRAPH-2

Input: U : A finite non-empty set of sets, *i.e.* U has no repeated elements and no empty elements.

Output: The maximal decomposition of U

```

3.1 begin
3.2    $G \leftarrow \text{ConstructGraph}(U)$ 
3.3   while  $G.\text{blue}$  or  $G.\text{green}$  do
3.4     for  $(X \rightarrow Y) \in G.\text{blue}$  do
3.5        $\text{BreakStrictSubset}(G, X, Y)$ ;
3.6     for  $\{X, Y\} \in G.\text{green}$  do
3.7        $\text{BreakTouching}(G, X, Y)$ ;
3.8     for  $(X \rightarrow Y) \in G.\text{blue}$  do
3.9        $\text{BreakLoop}(G, X, Y)$ ;
3.10  return  $G.\text{disjoint}$ 

```

5.2.1 Graph construction

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. Algorithm 4

describes the graph construction, and uses functions defined in Algorithms 6, 7, 8, and 9.

Algorithm 4: CONSTRUCTGRAPH

Input: A finite non-empty set U of sets, *i.e.* U has no repeated elements.

Output: A graph, G , in particular a set of blue ordered edges and green ordered edges.

The nodes of nodes of G are some (or all) of the elements of U

```

4.1 begin
4.2    $G.blue \leftarrow \emptyset$ 
4.3    $G.green \leftarrow \emptyset$ 
4.4    $G.nodes \leftarrow$  set of of labeled nodes, seeded from  $U$ 
4.5    $G.disjoint \leftarrow \emptyset$ 
4.6   for  $\{X, Y\} \subset U$  do
4.7     if  $X \subset Y$  then
4.8       |  $AddBlueArrow(G, X, Y)$ 
4.9     else if  $X \supset Y$  then
4.10      |  $AddBlueArrow(G, Y, X)$ 
4.11     else if  $X \not\subset Y$  then
4.12      |  $AddGreenLine(G, X, Y)$ 
4.13     else if  $X \perp Y$  then
4.14      | Nothing
4.15     else
4.16      |  $AddGreenLine(G, X, Y)$ 
4.17   for  $\alpha \in G.nodes$  do
4.18     |  $MaybeDisjointNode(G, \alpha)$ 
4.19   return  $G$ 

```

Notes about Algorithm 4:

Line 4.4: Each labeled node is a record with fields $\{label, subsets, supersets\}$. The label represents the Boolean expression for the subset in question. originally this label is identically the element coming from U , but at the end of the algorithm this label has become a Boolean expression expression some combination of elements of U .

Line 4.6: Notice that $X \neq Y$ are different because $\{X, Y\}$ is a two element set.

Line 4.6: Notice that once $\{X, Y\}$ has been visited $\{Y, X\}$ cannot be visited later, because it is the same set.

Line 4.16: This final **Else** clause covers the case in which it is not possible to determine whether whether $X \subset Y$ or whether $X \perp Y$. In this case the worst case, that they are non-disjoint, and draw green line between X and Y .

Because sometimes it is the case that the relation of two sets cannot be determined, we must interpreted as follows:

- a blue arrow between two nodes means a subset relation
- no blue arrow nor green line between two nodes means the disjoint relation.
- a green line between two nodes means the sets *may* touch.

Algorithm 5: MAYBEDISJOINTNODE

Input: G : a graph

Input: X : a node of G

Side Effects: Perhaps modifies $G.nodes$ and $G.disjoint$.

```
5.1 begin
5.2   if  $X.label = \emptyset$  then
5.3     | Delete  $X$  from  $G.nodes$ 
5.4   else if  $\emptyset = X.touches = X.supersets = X.subsets$  then
5.5     | Delete  $X$  from  $G.nodes$ 
5.6     | Push  $X$  onto  $G.disjoint$ 
```

Notes about Algorithm 5:

Line 5.2: Section 5.2.6 explains in more details, but here we simply avoid collecting the empty set.

Line 5.6: This push should have set-semantics. *I.e.*, if there is already a set in $G.disjoint$ which is equivalent to to this one, then do not push a second one.

Algorithm 6: ADDGREENLINE

Input: G : a graph

Input: X : a node of G

Input: Y : a node of G

Side Effects: Modifies G adding a green line between X and Y

6.1 **begin**

6.2 Push $\{X, Y\}$ onto $G.green$

6.3 Push X onto $Y.touches$

6.4 Push Y onto $X.touches$

Algorithm 7: DELETEGREENLINE

Input: G , a graph

Input: X : a node of G

Input: Y : a node of G

Side Effects: Modifies G deleting the green line between X and Y . Perhaps extends $G.disjoint$ and shortens $G.nodes$.

7.1 **begin**

7.2 Remove $\{X, Y\}$ from $G.green$

7.3 Remove X from $Y.touches$

7.4 Remove Y from $X.touches$

7.5 MaybeDisjointNode(G, X)

7.6 MaybeDisjointNode(G, Y)

Algorithm 8: ADDBLUEARROW

Input: G : a graph

Input: X : a node of G

Input: Y : a node of G

Side Effects: Modifies G adding a blue arrow from X to Y

8.1 **begin**

8.2 Push (X, Y) onto $G.blue$

8.3 Push X onto $Y.subsets$

8.4 Push Y onto $X.supersets$

Algorithm 9: DELETEBLUEARROW

Input: G : a graph

Input: X : a node of G

Input: Y : a node of G

Side Effects: Modifies G removing the blue arrow from X to Y . Perhaps extends $G.disjoint$ and shortens $G.nodes$

9.1 **begin**

9.2 Remove (X, Y) from $G.blue$

9.3 Remove X from $Y.subsets$

9.4 Remove Y from $X.supersets$

9.5 MaybeDisjointNode(G, X)

9.6 MaybeDisjointNode(G, Y)

This construction assures that no two nodes have both a green line and also a blue arrow between them.

There is an important design choice to be made: How to represent transitive subset relationships. There are two variations. We call these variations *implicit-inclusion vs. explicit-inclusion*.

The graph shown in Figure 3 uses explicit-inclusion. In the graph $A_5 \subset A_8 \subset A_1$. The question is whether it is necessary to include the blue arrow from ⑤ to ①. Explicit inclusion means that all three arrows are maintained in the graph: ⑤ to ⑧, ⑧ to ① and explicitly ⑤ to ①. Implicit-inclusion means that the arrow from ⑤ to ① is omitted.

The algorithm which we explain in this section can be made to accommodate either choice as long as the choice is enforced consistently. Omitting the arrow obviously lessens the number of arrows to have to maintain, but makes some of the algorithm more tedious.

As far as the initialization of the graph is concerned, in the variation of implicit-inclusion it is necessary to avoid creating arrows (or remove them after the fact) which are implicit.

5.2.2 Strict sub-set

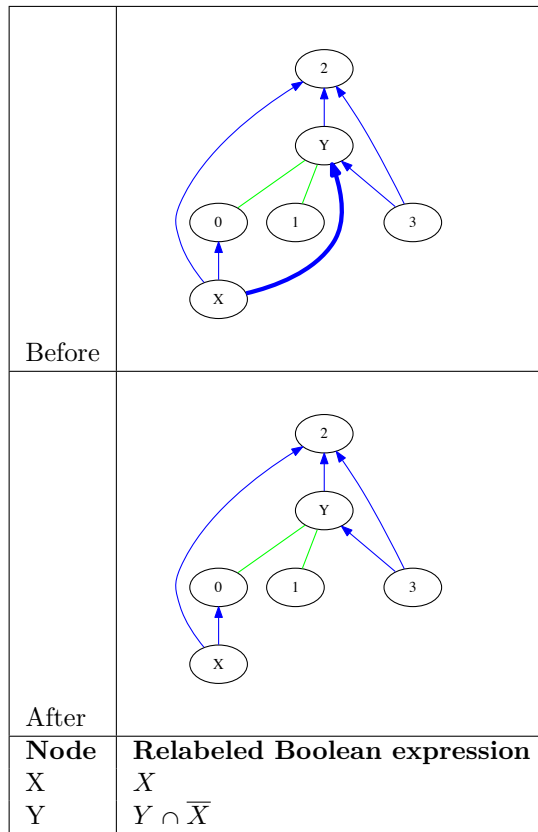


Figure 4: Strict sub-set before and after mutation

Algorithm 10: BREAKSTRICTSUBSET: Breaks a strict subset edge if possible

Input: G is a graph.
Input: X : a node in G
Input: Y : a node in G
Output: G
Side Effects: Possibly deletes a vertex and changes a label.

```

10.1 begin
10.2   if  $Y \notin X.\text{supersets}$  then
10.3     | Nothing
10.4   else if  $X.\text{subsets} \neq \emptyset$  then
10.5     | Nothing
10.6   else if  $X.\text{touches} \neq \emptyset$  then
10.7     | Nothing
10.8   else
10.9     |  $Y.\text{label} \leftarrow Y \cap \bar{X}$ 
10.10    |  $DeleteBlueArrow(G, X, Y)$ 
10.11  return  $G$ 
  
```

As shown in Algorithm 10, blue arrows indicate sub-set/super-set relations, they point from a sub-set to a super-set. A blue arrow from X to Y may be eliminated if conditions are met:

- X has no blue arrows pointing to it, and
- X has no green lines touching it.

These conditions mean that X represents a set which has no subsets elsewhere in the graph, and also that X represents a set which touches no other set in the graph.

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

On eliminating the blue arrow, replace the label Y by $Y \cap \bar{X}$.

In Figure 3, nodes $\textcircled{5}$ and $\textcircled{6}$ meet this criteria. A node, such as $\textcircled{5}$ may have multiple arrows leaving it. Once the final such arrow is broken, the node becomes isolated.

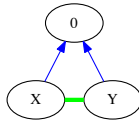


Figure 5: Subtle case

The restriction that the node X have no green line touching it is subtle. Consider the graph in Figure 5. If either the blue arrow from \textcircled{X} to $\textcircled{0}$ or the blue arrow from \textcircled{Y} to $\textcircled{0}$ is broken by the rule *Strict sub-set*, then the other of the two arrows becomes incorrect. Therefore, we have the restriction for the *Strict sub-set* rule that \textcircled{X} and \textcircled{Y} have no green lines connecting to them. The *Relaxed sub-set* condition is intended to cover this case.

In the variation of implicit-inclusion is it necessary to add all the superclasses of Y to become also superclasses of X . This means in the graph, for each blue arrow from \textcircled{Y} to $\textcircled{0}$, we must add blue arrows leading from \textcircled{X} to $\textcircled{0}$. In in the example 4, if we were using implicit-inclusion, the arrow from \textcircled{X} to $\textcircled{2}$ would be missing as it would be implied by the other arrows. Therefore, the blue arrow from \textcircled{X} to $\textcircled{2}$ would need to be added in the *After* graph of Figure 4.

5.2.3 Relaxed sub-set

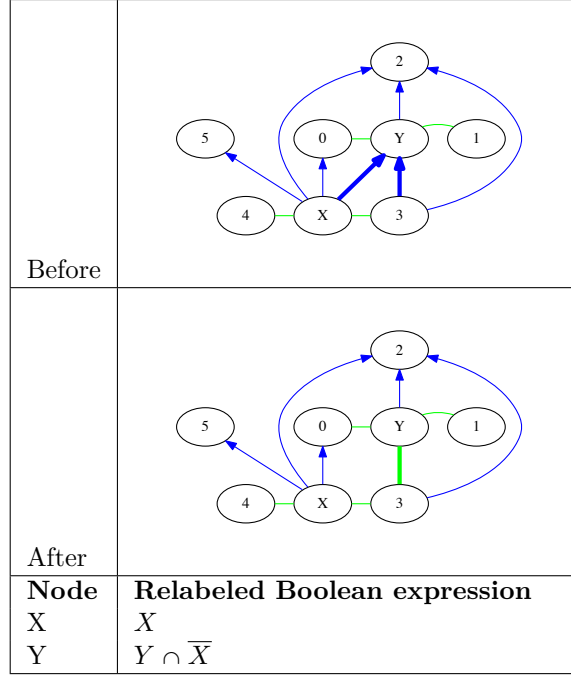


Figure 6: Relaxed sub-set before and after mutation

Algorithm 11 is similar to the *Strict sub-set* algorithm except that the sub-set node X is allowed to touch other nodes. But special attention is given if X touches a sibling node; *i.e.* if X has a green line connecting to a intermediate node which also has a blue arrow pointing to Y . This case is illustrated in the *Before* graph in Figure 6. This graph can be mutated to the *After* graph shown in Figure 6.

Algorithm 11: BREAKRELAXEDSUBSET

Input: G : a graph

Input: X : a node in G

Input: Y : a node in G

Output: G

Side Effects: Perhaps changes a label, and some blue vertices removed or converted to green.

```

11.1 begin
11.2   if  $Y \notin X.\text{supersets}$  then
11.3     | Nothing
11.4   else if  $X.\text{subsets} \neq \emptyset$  then
11.5     | Nothing
11.6   else
11.7     |  $Y.\text{label} \leftarrow Y \cap \bar{X}$ 
11.8     for  $\alpha \in (X.\text{touches} \cap Y.\text{subsets})$  do
11.9       |  $\text{AddGreenLine}(G, \alpha, Y)$ 
11.10      |  $\text{DeleteBlueArrow}(G, \alpha, Y)$ 
11.11      $\text{DeleteBlueArrow}(G, X, Y)$ 
11.12   return  $G$ 

```

Notes about Algorithm 11:

Line 11.8: α iterates over the intersection of $X.touches$ and $Y.subsets$.

Line 11.10: Be careful to *add* and *delete* in that order. Reversing the order may cause the function *DeleteBlueArrow* to mark the node as disjoint via a call to *MaybeDisjointNode*.

The node label transformations for this case are exactly the same for the *Strict sub-set* condition. The only different consequence is that each node connected to Y with a blue arrow (such as ③ in the *Before* graph of Figure 6) which is also connected to X by a green line must be transformed from a blue arrow to a green line as shown in the *After* graph.

In the variation of implicit-inclusion is it necessary to add all the superclasses of Y to become also superclasses of X and to all of the children of Y which touch X . This means in the graph, for each blue arrow from \textcircled{Y} to \textcircled{n} , we must add blue arrows leading from \textcircled{X} to \textcircled{n} and also from $\textcircled{3}$ to \textcircled{n} . In the example 6, if we were using implicit-inclusion, the arrows from \textcircled{X} to $\textcircled{2}$ and from $\textcircled{3}$ to $\textcircled{2}$ would be missing as they would be implied by the other arrows. Therefore, the blue arrows from \textcircled{X} to $\textcircled{2}$ and from $\textcircled{3}$ to $\textcircled{2}$ would need to be added in the *After* graph.

5.2.4 Touching connections

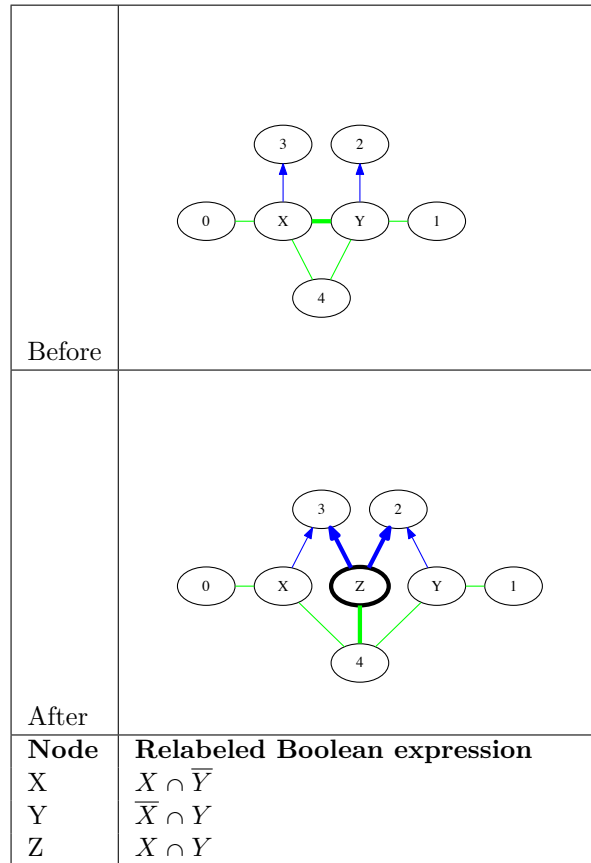


Figure 7: Touching connections before and after mutation

Green lines indicate partially overlapping sets. A green line connecting X and Y may be broken if the following condition is met:

- Neither X nor Y has a blue arrow pointing to it; *i.e.* neither represents a super-set of something else in the graph.

Eliminating the green line *separates* X and Y . To do this X and Y must be relabeled and a new node must be added to the graph. Algorithm 12 explains this procedure.

Algorithm 12: BREAKTOUCHING

Input: G : a graph
Input: X : a node in G
Input: Y : a node in G
Output: G
Side Effects: Perhaps removes some vertices from G , and adds new nodes and vertices.

```

12.1 begin
12.2   if  $Y \notin X.touches$  then
12.3     | Nothing
12.4   else if  $X.subsets \neq \emptyset$  then
12.5     | Nothing
12.6   else if  $Y.subsets \neq \emptyset$  then
12.7     | Nothing
12.8   else
12.9      $(X.label, Y.label) \leftarrow (X \cap \bar{Y}, Y \cap \bar{X})$ 
12.10     $Z \leftarrow G.AddNode()$ 
12.11     $Z.label \leftarrow X \cap Y$ 
12.12    for  $\alpha \in (X.supersets \cup Y.supersets)$  do
12.13      |  $AddBlueArrow(G, Z, \alpha)$ 
12.14    for  $\alpha \in (X.touches \cap Y.touches)$  do
12.15      |  $AddGreenLine(G, Z, \alpha)$ 
12.16     $DeleteGreenLine(G, X, Y)$ 
12.17  return  $G$ 

```

Notes about Algorithm 12:

Line 12.9: This is a parallel assignment. *I.e.*, calculate $X \cap \bar{Y}$ and $Y \cap \bar{X}$ before assigning them respectively to $X.label$ and $Y.label$.

Line 12.11: Introduce new node labeled $X \cap Y$.

Line 12.12: Blue union, draw blue arrows from this node, $X \cap Y$, to all the nodes which either X or Y points to. *I.e.*, the super-sets of $X \cap Y$ are the union of the super-sets of X and of Y .

Line 12.14: Green Intersection, draw green lines from $X \cap Y$ to all nodes which both X and Y connect to. *I.e.* the connections to $X \cap Y$ are the intersection of the connections of X and of Y .

Line 12.15: Exception, if there is (would be) a green and blue vertex between two particular nodes, omit the green one.

Line 12.16: Be careful to *add* and *delete* in that order. Calling $DeleteGreenLine$ before $AddGreenLine$ cause the function $DeleteGreenLine$ to mark the node as disjoint via a call to $MaybeDisjointNode$.

Figure 7 illustrates the step of breaking such a connection between nodes \textcircled{X} and \textcircled{Y} by introducing the node \textcircled{Z} .

5.2.5 Loops

This step can be omitted if one of the previous conditions is met: either *strict subset* or *touching connection*. The decision of whether to omit this step is not a correctness question, but rather a performance question which is addressed in Section 9.2.

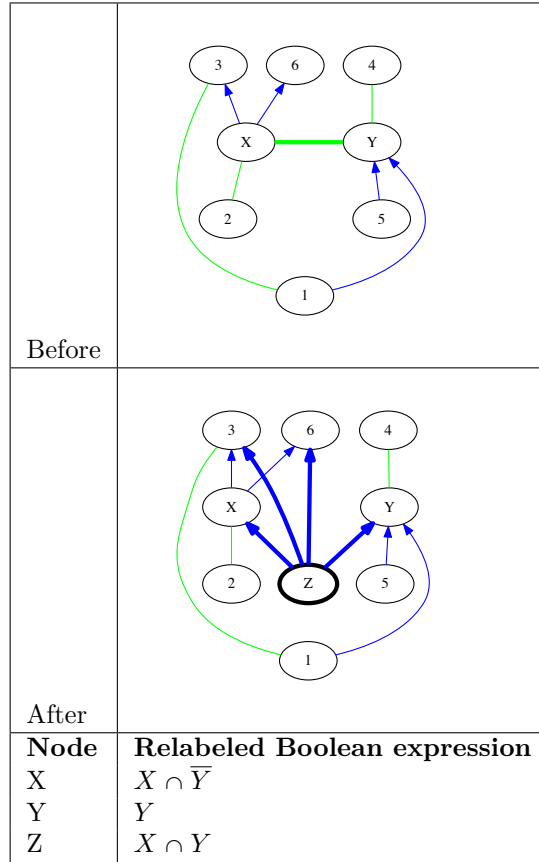


Figure 8: Graph meeting the *loop* condition

As is detailed in Algorithm 13 a green line connecting \textcircled{X} and \textcircled{Y} may be removed if the following conditions are met:

- \textcircled{X} has no blue arrow pointing to it.
- \textcircled{Y} has at least one blue arrow pointing to it.

The rare necessary for this operation arises in graphs such as the green line connecting \textcircled{X} and \textcircled{Y} in Figure 8. To eliminate this green line proceed by splitting node \textcircled{X} into two nodes: the part that is included in set Y and the part that is disjoint from set Y . The result of the mutation is shown in the *After* graph.

- Remove the green line between \textcircled{X} and \textcircled{Y}
- Create a new node \textcircled{Z} copying all the green and blue connections from \textcircled{X} .
- Create a blue arrow from \textcircled{Z} to \textcircled{Y} , because $Z = X \cap Y \subset Y$.
- Create a blue arrow from \textcircled{Z} to \textcircled{X} , because $Z = X \cap Y \subset X$.
- $Z \leftarrow X \cap Y$. *I.e.* label the new node.
- $X \leftarrow \bar{X} \cap Y$. *I.e.* relabel \textcircled{X} .
- For each node $\textcircled{\Theta}$ which is a parent of either \textcircled{X} or \textcircled{Y} (union), draw an blue arrow from \textcircled{Z} to $\textcircled{\Theta}$.

This graph operation effectively replaces the two nodes X and Y with the three nodes $X \cap Y$, $X \cap \bar{Y}$, and Y . This is a reasonable operation because $X \cup Y = X \cap \bar{Y} \cup \bar{X} \cap Y \cup Y$.

Algorithm 13: BREAKLOOP

Input: G : a graph
Input: X : a node in G
Input: Y : a node in G
Output: G
Side Effects: Perhaps removes some with some of its green vertices, and adds new nodes and blue vertices.

```

13.1 begin
13.2   if  $Y \notin X.touches$  then
13.3     | Nothing
13.4   else if  $X.subsets \neq \emptyset$  then
13.5     | Nothing
13.6   else if  $Y.subsets = \emptyset$  then
13.7     | Nothing
13.8   else
13.9     |  $Z \leftarrow G.AddNode()$ 
13.10    |  $Z.label \leftarrow X \cap Y$ 
13.11    |  $X.label \leftarrow X \cap \bar{Y}$ 
13.12    | for  $\alpha \in X.touches$  do
13.13      |  $AddGreenLine(G, Z, \alpha)$ 
13.14    | for  $\alpha \in (X.supersets \cup Y.supersets)$  do
13.15      |  $AddBlueArrow(G, Z, \alpha)$ 
13.16    |  $AddBlueArrow(G, Z, Y)$ 
13.17    |  $AddBlueArrow(G, Z, X)$ 
13.18    |  $DeleteBlueArrow(G, X, Y)$ 
13.19  return  $G$ 

```

5.2.6 Discovered Empty Set

During the execution of the algorithm it may happen that derived sets are discovered to be empty. This occurrence is a consequence of the semantics of the green lines and blue arrows. Recall that a green line connecting two nodes indicates that the corresponding sets are not *known* to be disjoint. A consequence is this semantic is that two nodes representing disjoint sets are connected by a green line, when the intersection of the two sets is eventually calculated, it the intersection will be found to be empty.

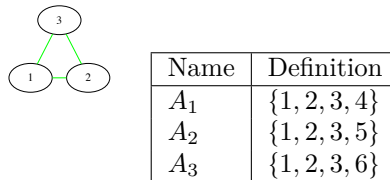


Figure 9: Setup for Discovered Empty Set

Example 22. Consider the definitions and graph shown in Figure 9. If we break the connection between node ① and ②, the configuration in Figure 10 results. The resulting graph shows a green line connecting ① and ③ and also a green line connecting ② and ③.

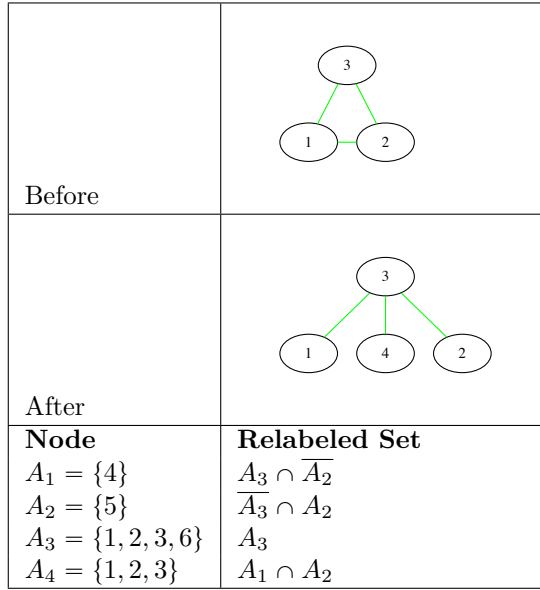


Figure 10: Connection between disjoint sets

This does not cause a problem in the semantics of the representation, but rather a performance issue. Figure 11 shows the result of breaking the green line connecting ② and ③. Node ② becomes isolated which correctly represents the set $\{5\}$, and ⑤ is derived which represents \emptyset .

The semantics are preserved because $\bigcup_{n=1}^5 A_n = \{1, 2, 3, 4, 5\}$, and the isolated nodes ② and ⑤ represent disjoint sets. $\{5\} \perp \emptyset$

Another case where a set may be discovered to be empty is when the relative complement is performed between a superset and subset. If the two sets are actually equal (recall that $A = B \implies A \subset B$) then the relative complement is empty, ($A = B \implies A \cap \overline{B} = \emptyset$).

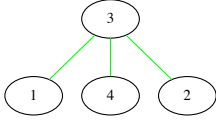
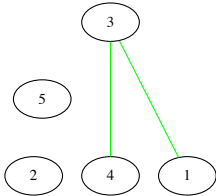
Before	
After	
Node $A_1 = \{4\}$ $A_2 = \{5\}$ $A_3 = \{1, 2, 3, 6\}$ $A_4 = \{1, 2, 3\}$ $A_5 = \emptyset$	Relabeled Set A_1 $A_2 \cap \overline{A_3}$ $\overline{A_2} \cap A_3$ A_4 $A_2 \cap A_3$

Figure 11: Discovered Empty Set

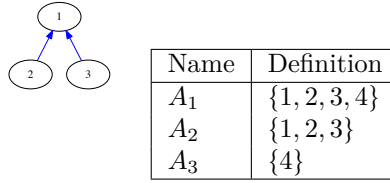


Figure 12: Setup for Discovered Equal Sets

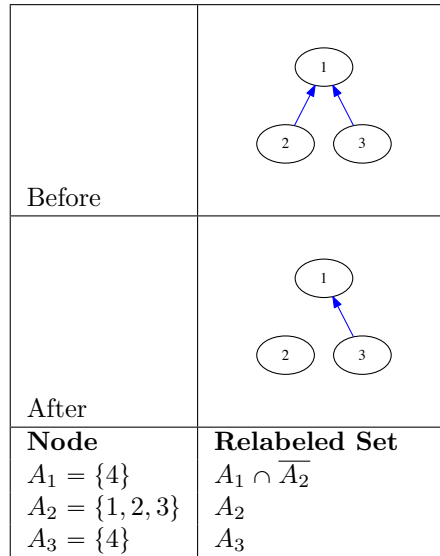


Figure 13: Connection between equal sets

Example 23. Consider the definitions and graph shown in Figure 12. In Figure 13 we break the blue arrow from ② to ①, then a blue arrow remains from ③ to ①, which represents a set which is a subset of itself. As in Example 22, this is not an error in the semantics of the representation, but rather a performance problem. In Figure 14 we break the connection from ③ to ① resulting in three isolated nodes representing the sets \emptyset , $\{1, 2, 3\}$, and $\{4\}$ which are three disjoint sets whose union is correct is $\{1, 2, 3, 4\}$ as expected.

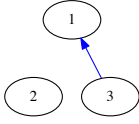
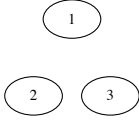
Before	
After	
Node $A_1 = \emptyset$ $A_2 = \{1, 2, 3\}$ $A_3 = \{4\}$	Relabeled Set $A_1 \cap \overline{A_3}$ A_2 A_3

Figure 14: Connection between equal sets

Since intersection and relative complement operations may result in the empty set, as illustrated in Examples 22 and 23, the implementation of the set disjoint decomposition algorithm must take this into account. There are several possible approaches.

- Late check:** The simplest approach is to check for vacuity once a node has been disconnected from the graph. Once a node has no touching nodes, no subset nodes, and not superset nodes, it should be *removed* from the graph data structure, checked for vacuity, and if non-empty, and the set it represents added to a list of disjoint sets.
- Correct label:** Test for \emptyset each time a label of a node changes, *i.e.*, it thereafter represents a smaller set than before.
- Correct connection:** Each time the label of a node changes, *i.e.* it thereafter represents a smaller set than before, all of the links to neighbors (green lines and blue arrows) become suspect. *I.e.*, when a label changes, re-validate all the touching, subset, and superset relationships and update the blue arrows and green lines accordingly.

The choice of data structure used for the algorithm may influence how expensive the vacuity check is, and thus may influence which approach is taken.

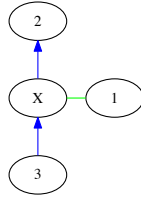


Figure 15: Discovered emptyset

When a node is discovered to represent the empty set, its connections must be visited. Consider the graph in Figure 15.

- Touching node:** A green line connecting the node to another node can be deleted, because the empty set is disjoint from every set including from the empty set. In Figure 15 if X is discovered to represent \emptyset , then the green line between \textcircled{x} and $\textcircled{1}$ can simply be deleted.
- Superset:** A blue arrow from the node to another node may be deleted. In Figure 15 if X is discovered to represent \emptyset , then the blue arrow from \textcircled{x} to $\textcircled{2}$ can be deleted. This is because if we attempted to relabel A_2 as $A_2 \cap \overline{X}$ we'd result again with A_2 because $A_2 \setminus \emptyset = A_2$.
- Subset:** A blue arrow from another node to the node in question can be removed, and the other node can be inferred to represent the empty set. Thus the empty set reduction can be applied recursively to that node. In Figure 15 if X is discovered to represent \emptyset , then we know that $A_3 \subset X$ and thus $A_3 = \emptyset$. This means we can delete the blue arrow from $\textcircled{3}$ to \textcircled{x} , and then recursively apply the reduction to $\textcircled{3}$.

5.3 Recursion and Order of Iteration

It is natural to wonder whether the order in which the nodes are visited may have an effect on the execution time of the algorithm. The manner in which the traversal order effects the calculation performance has been investigated and is explained in Section 9.2.

Our implementation represents nodes of the graph as objects, where each node object has three slots containing lists of subset, superset, and touching nodes. Thus graph reduction, in our implementation, involves iterating over the nodes, and repeating the iteration until they all loose all their connections.

An alternate implementation might as well represent connections as objects, thus allowing the reduction algorithm to directly iterate over the connections until they are eliminated. We have not investigated this approach.

Given that our algorithm is required to visit each node, there are several obvious strategies to choose the order of iteration. This boils down to sorting the list of nodes into some order before iterating over it. We have investigated five such strategies.

- SHUFFLE:** Sort the list into random order.
- INCREASING-CONNECTIONS:** Sort the nodes into order of increasing number of connections; *i.e.*, number of touching nodes plus the number of subset nodes plus the number of superset nodes.
- DECREASING-CONNECTIONS:** Sort the nodes into order of decreasing number of connections.
- BOTTOM-TO-TOP:** Sort into increasing order according to number of superset nodes. This sort assumes supersets are explicitly represented in the connections, because a subset node will also contain connections to the supersets of its direct supersets.
- TOP-TO-BOTTOM:** Sort into decreasing order according to number of superset nodes.

All of the graph operations described in Section 5.1 depend on a node not having subset nodes. An additional detail of the strategy which can be (and has been) employed in addition to the node visitation order is whether to recursively apply the reduction attempts to subset nodes before superset nodes. *I.e.*, while iterating over the nodes, would we recursively visit subset nodes? The hope is that a node is more likely to be isolate-able if we attempt to break subclass relations directly when encountered, rather than treating the nodes when they appear in the visitation order.

5.4 Running the algorithm on an example

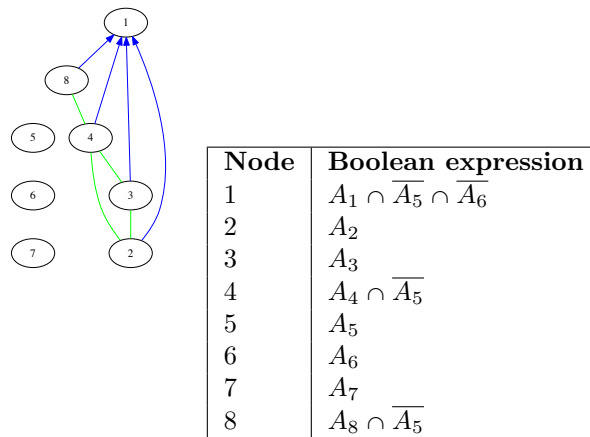


Figure 16: State 1: Topology graph, after isolating 5 and 6.

Nodes ⑤ and ⑥ in Figure 3 meet the *strict sub-set* conditions, thus the arrow connecting them to their super-sets, $5 \rightarrow 1$, $5 \rightarrow 4$, $5 \rightarrow 8$, and $6 \rightarrow 1$ can be eliminated and the super-set

nodes relabeled. *I.e.* $\textcircled{5}$ relabeled $A_8 \mapsto A_8 \cap \overline{A_5}$, $\textcircled{4}$ relabeled $A_4 \mapsto A_4 \cap \overline{A_5}$, and $\textcircled{1}$ relabeled $A_1 \mapsto A_1 \cap \overline{A_5} \cap \overline{A_6}$. The result of these operations is that nodes $\textcircled{5}$ and $\textcircled{6}$ have now severed all connections, and are thus isolated. The updated graph is shown in Figure 16.

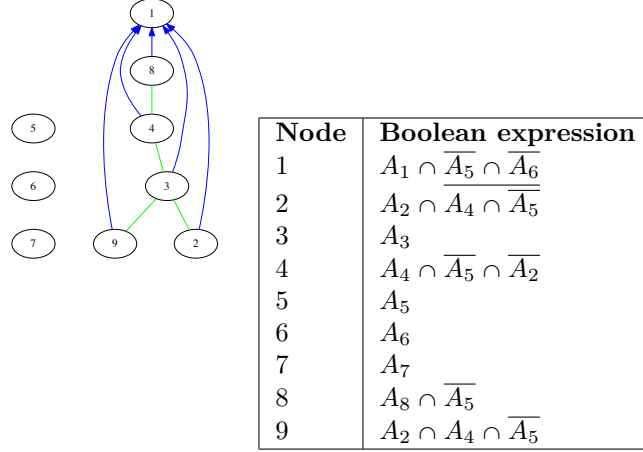


Figure 17: State 2: Topology graph after disconnecting 2 from 4

The green line between nodes as $\textcircled{2}$ and $\textcircled{4}$ in Figure 16 meets the *touching connections* conditions. The nodes can thus be *separated* by breaking the connection, deleting the green line. To do this we must introduce a new node which represents the intersection of the sets $\textcircled{2}$ and $\textcircled{4}$. The new node is labeled as the Boolean intersection: $A_2 \cap A_4 \cap \overline{A_5}$, and is labeled $\textcircled{9}$ in Figure 17.

We must also introduce new blue lines from $\textcircled{9}$ to *any* node that either $\textcircled{2}$ points to or $\textcircled{4}$ points to, which is $\textcircled{1}$ in this case.

In addition we must draw green lines to nodes which both $\textcircled{2}$ and $\textcircled{4}$ have green lines touching. In this cases that is only the node $\textcircled{3}$. So a green line is drawn between $\textcircled{9}$ and $\textcircled{3}$.

The green line between $\textcircled{2}$ and $\textcircled{4}$ is deleted. The two nodes are relabeled: $\textcircled{2}$: $A_2 \mapsto A_2 \cap A_4 \cap \overline{A_5}$ and $\textcircled{4}$: $A_4 \cap \overline{A_5} \mapsto A_4 \cap \overline{A_5} \cap \overline{A_2}$.

These graph operations should continue until all the nodes have become isolated. Observing Figure 17 we see that several green lines meet the *touching connections*: $\textcircled{2} - \textcircled{3}$, $\textcircled{3} - \textcircled{4}$, $\textcircled{3} - \textcircled{9}$, and $\textcircled{4} - \textcircled{8}$. It is not clear which of these connections should be broken next. *I.e.* what is the best strategy to employ when choosing the order to break connections. This is a matter for further research; we don't suggest any best strategy at this time. Nevertheless, we continue the segmentation algorithm a couple more steps.

In Figure 17, consider eliminating the green connection $\textcircled{4} - \textcircled{8}$. We introduce a new node $\textcircled{10}$ representing the intersection, thus associated with the Boolean expression $A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$. The union of the super-sets of $\textcircled{4}$ and $\textcircled{8}$, *i.e.* the union of the destinations of the arrows leaving $\textcircled{4}$ and $\textcircled{8}$ is just the node $\textcircled{1}$, thus we must introduce a blue arrow $\textcircled{10} \rightarrow \textcircled{1}$. There are no nodes which both $\textcircled{4}$ and $\textcircled{8}$ touch with a green line, so no green lines need to be added connecting to $\textcircled{10}$. We now relabel $\textcircled{4}$ and $\textcircled{8}$ with the respective relative complements. $8 \mapsto 8 \cap \overline{4}$ and $4 \mapsto 4 \cap \overline{8}$. The Boolean expressions are shown in Figure 18.

Observing Figure 18 we see it is possible to disconnect $\textcircled{8}$ from $\textcircled{1}$ and thereafter disconnect $\textcircled{10}$ from $\textcircled{1}$. Actually you may choose to do this in either order. We will operate on $\textcircled{8}$ and then on $\textcircled{10}$, to result in the graph in Figure 19.

From Figure 19 it should be becoming clear that the complexity of the Boolean expressions in each node is becoming more complex. If we continue this procedure, eliminating all the blue arrows and green connecting lines, we will end up with 13 isolated nodes (each time a green line is eliminated one additional node is added). Thus the Boolean expressions can become exceedingly complex. A question which arises is whether it is better to *simplify* the Boolean expressions at

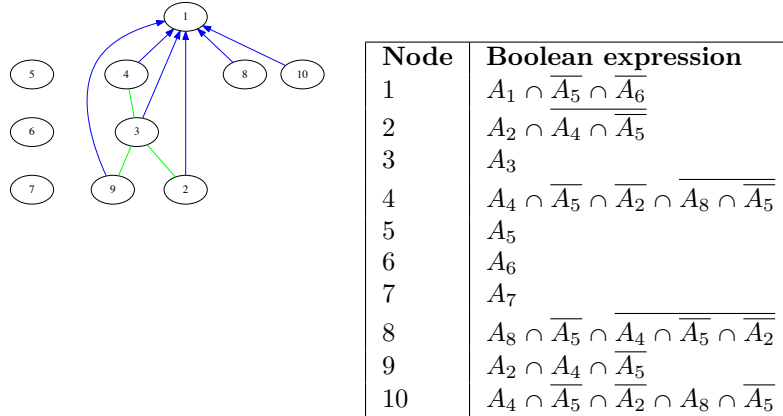


Figure 18: State 3: Topology graph after disconnecting 4 from 8

each step, or whether it is better to wait until the end. The algorithm shown in Section 7 promises to obviate that dilemma.

There are some subtle corner cases which may not be obvious. It is possible in these situations to end up with some disjoint subsets which are empty. It is also possible also that the same subset is derived by two different operations in the graph, but whose equations are very different.

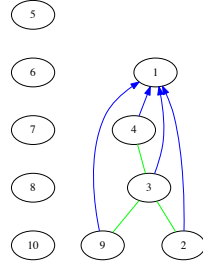
This phenomenon is a result of a worst case assumption, *green intersection* in the algorithm. Consider a case where nodes \textcircled{A} , \textcircled{B} , and \textcircled{C} mutually connected with green lines signifying that the corresponding sets touch (are not disjoint). If the connection $\textcircled{A} - \textcircled{B}$ is broken, a new green line must be drawn between the new node \textcircled{D} and \textcircled{C} . Why? Because it is possible that the set represented by $A \cap B \not\perp C$. However, this it is not guaranteed. It may very well be the case that both $A \not\perp C$ and $B \not\perp C$ while $A \cap B \perp C$. Consider the simple example $A = \{1, 2\}$, $B = \{2, 3\}$, $C = \{1, 3\}$. $A \not\perp C$, $B \not\perp C$, but $A \cap B = \{2\} \perp \{1, 3\} = C$.

This leads to the possibility that there be green lines in the topology graph which represent phantom connections. Later on in the algorithm when the green line between \textcircled{D} and \textcircled{C} is broken redundant sets may occur. Nodes \textcircled{C} and \textcircled{D} will be broken into three, $C \cap D$, $C \cap \overline{D}$, and $D \cap \overline{C}$. But $C \cap D = \emptyset$, $C = C \cap \overline{D}$ and $D = D \cap \overline{C}$. If a similar phenomenon occurs between C and some other set, say E , then we may end up with multiple equivalent sets with different names, and represented by different nodes of the topology graph: $C = C \cap \overline{D} = C \cap \overline{E}$.

To identify each of these cases, each of the resulting sets must be checked for vacuity, and uniqueness. No matter the programming language of the algorithm implementation, it is necessary to be implement these two checks.

In Common Lisp there are two possible ways to check for vacuity, *i.e.* to detect whether a type is empty. (1) Symbolically reduce the type specifier, *e.g.* `(and fixnum (not fixnum))` to a canonical form with `nil` in case the specifier specifies the `nil` type. (2) Use the `subtypep` function to test whether the type is a subtype of `nil`. To test whether two specifiers specify the same type there are two possible approaches in Common Lisp. (1) Symbolically reduce each expression such as `(or integer number string)` and `(or string fixnum number)` to canonical form, and compare the results with the `equal` function. (2) Use the `subsetp` function twice to test whether each is a subtype of the other.

See section 9 for a description of the performance of this algorithm.



Node	Boolean expression
1	$A_1 \cap \overline{A_6} \cap A_8 \cap \overline{A_5} \cap \overline{A_4} \cap \overline{A_5} \cap \overline{A_2} \cap \overline{A_4} \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$
2	$A_2 \cap A_4 \cap \overline{A_5}$
3	A_3
4	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$
5	A_5
6	A_6
7	A_7
8	$A_8 \cap \overline{A_5} \cap \overline{A_4} \cap \overline{A_5} \cap \overline{A_2}$
9	$A_2 \cap A_4 \cap \overline{A_5}$
10	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$

Figure 19: State 4: Topology graph after isolating 8 and 10

5.5 Correctness of the graph based algorithm

6 Type disjoint decomposition as SAT problem

This problem of how to decompose sets, like those shown in Figure 1 into disjoint subsets as shown in Table 1 can be viewed as a variant of the well known *Satisfiability Problem*, commonly called SAT. [JEH01] The problem is this: given a Boolean expression in n variables, find a solution. This is to say: find an assignment (either *true* or *false*) for each variable which makes the expression evaluate to *true*. This problem is known to be NP-Complete.

The approach is to consider the correspondence between the solutions of the Boolean equation: $A_1 + A_2 + \dots + A_M$, versus the set of subsets of $A_1 \cup A_2 \cup \dots \cup A_M$. Just as we can enumerate the $2^M - 1 = 255$ solutions of $A_1 + A_2 + \dots + A_M$, we can analogously enumerate the subsets of $A_1 \cup A_2 \cup \dots \cup A_M$.

discard	0000 0000	$\overline{A_1} \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_4} \cap \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$
1	1000 0000	$A_1 \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_4} \cap \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$
2	0100 0000	$\overline{A_1} \cap A_2 \cap \overline{A_3} \cap \overline{A_4} \cap \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$
3	1100 0000	$A_1 \cap A_2 \cap \overline{A_3} \cap \overline{A_4} \cap \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$

254	1111 1110	$A_1 \cap A_2 \cap A_3 \cap A_4 \cap A_5 \cap A_6 \cap A_7 \cap \overline{A_8}$
255	1111 1111	$A_1 \cap A_2 \cap A_3 \cap A_4 \cap A_5 \cap A_6 \cap A_7 \cap A_8$

Figure 20: Correspondence of Boolean true/false equation with Boolean set equation

If we assume $M = 8$ as in Figure 1, the approach here is to consider every possible solution of the Boolean equation: $A_1 + A_2 + \dots + A_8$. There are $2^8 - 1 = 255$ such solutions, because every

8-tuple of 0's and 1's is a solution except 0000 0000. If we consider the enumerated set of solutions: 1000 0000, 0100 0000, 1100 0000, ... 1111 1110, 1111 1111. We can analogously enumerate the potential subsets of the union of the sets shown in Figure 1: $A_1 \cup A_2 \cup A_3 \cup A_4 \cup A_5 \cup A_6 \cup A_7 \cup A_8$. Each is a potential solution represents an intersection of sets in $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$. Such a correspondence is shown in Figure 20.

It remains only to eliminate the intersections which can be proven to be empty. For example, we see in 1 that A_1 and A_7 are disjoint, which implies $\emptyset = A_1 \cap A_7$, which further implies that every line of Table 20 which contains A_1 and A_7 is \emptyset . That is 64 lines eliminated. For example line 255: $\emptyset = A_1 \cap A_2 \cap A_3 \cap A_4 \cap A_5 \cap A_6 \cap A_7 \cap A_8$, and 63 other lines.

In this as all SAT problems, certain of these 2^8 possibilities can be eliminated because of known constraints. The constraints are derived from the known subset and disjoint-ness relations of the given sets. Looking at the Figure 1 we see that $A_5 \subset A_8$, which means that $A_5 \cap \overline{A_8} = \emptyset$. So we know that all solutions where $A_5 = 1$ and $A_8 = 0$ can be eliminated. This elimination by constraint $A_5 \cap \overline{A_8} = \emptyset$ and the previous one $A_1 \cap A_7 = \emptyset$ are represented each in the boolean equation as a multiplication (Boolean multiply) by $\overline{A_1 A_7}$ and $\overline{A_5 A_8}$: $(A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8) \cdot \overline{A_1 A_7} \cdot \overline{A_5 A_8}$.

There are as many as $\frac{8 \cdot 7}{2} = 28$ possible constraints imposed by pair relations. For each $\{X, Y\} \subset \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$:

Subset If $X \subset Y$, multiply the by constraint $\overline{X Y} = (\overline{X} + Y)$

Super-set If $Y \subset X$, multiply by the constraint $\overline{X Y} = (X + \overline{Y})$

Disjoint If $X \cap Y = \emptyset$, multiply by the constraint $\overline{X Y} = (\overline{X} + \overline{Y})$.

Otherwise no constraint.

A SAT solver will normally find one solution. That's just how they traditionally work. But the SAT flow can easily be extended so that once a solution is found, a new constraint can be generated by logically negating that solution, allowing the SAT solver to find a second solution. For example, when it is found that 1111 0000 (corresponding to $A_1 \cap A_2 \cap A_3 \cap A_4 \cap \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$) is a solution, the equation can thereafter be multiplied by the new constraint $(A_1 A_2 A_3 A_4 \overline{A_5} \overline{A_6} \overline{A_7} \overline{A_8})$, allowing the SAT solver to find yet another solution if such exists.

The process continues until there are no more solutions.

As a more concrete example of how the SAT approach works when applied to Common Lisp types, consider the case of the three types `array`, `sequence`, and `vector`. Actually, `vector` is the intersection of `array` and `sequence`.

First the SAT solver constructs (explicitly or implicitly) the set of candidates corresponding to the list types.

```
(and array      sequence      vector)
(and array      sequence      (not vector))
(and array      (not sequence) vector)
(and array      (not sequence) (not vector))
(and (not array) sequence      vector)
(and (not array) sequence      (not vector))
(and (not array) (not sequence) vector)
(and (not array) (not sequence) (not vector))
```

The void one `(and (not array) (not sequence) (not vector))` can be immediately disregarded.

Since `vector` is a subtype of `array`, all types which include both `(not array)` and also `vector` can be disregarded: `(and (not array) sequence vector)` and `(and (not array) (not sequence) vector)`. Furthermore, since `vector` is a subtype of `sequence`, all types which include both `(not sequence)` and also `vector` can be disregarded. `(and array (not sequence) vector)` and `(and (not array) (not sequence) vector)` (which has already been eliminated by the previous step). The remaining ones are:

```

(and array      sequence      vector)      = vector
(and array      sequence      (not vector)) = nil
(and array      (not sequence) (not vector)) = (and array (not vector))
(and (not array) sequence      (not vector)) = (and sequence (not vector))

```

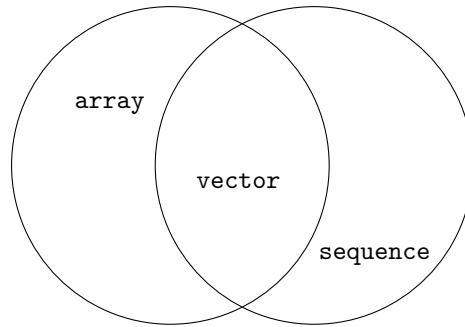


Figure 21: Relation of vector, sequence, and array

The algorithm returns a false positive. Unfortunately, this set still contains the `nil`, empty, type `(and array sequence (not vector))`. Figure 21 shows the relation of the Common Lisp types `array`, `vector`, and `sequence`. We can see that `vector` is the intersection of `array` and `sequence`. The algorithm discussed above failed to introduce a constraint corresponding to this identity which implies that $array \cap sequence \cap \overline{vector} = \emptyset$.

It seems the SAT algorithm greatly reduces the search space, but is not able to give the minimal answer. The resulting types must still be tested for vacuity. This is easy to do, just use the `subtypep` function to test whether the type is a subtype of `nil`. E.g., `(subtypep '(and array sequence (not vector)) nil)` returns `t`. There are cases where the `subtypep` will not be able to determine the vacuity of a set. Consider the example: `(and fixnum (not (satisfies oddp)) (not (satisfies evenp)))`.

7 Binary Decision Diagrams

A Boolean equation can be represented by a data structure called a Binary Decision Diagram (BDD) [Bry86, Bry92, Ake78][Knu09, Section 7.1.4]. Henrik Reif Andersen summarized many of the algorithms for efficiently manipulating BDDs [And99]. Not least important in Andersen's discussion is how to use a hash table and dedicated constructor function to eliminate any redundancy with a tree of within a forest of trees.

Figure 22 shows an example of a BDD which represents a particular function of three Boolean variables: A_1 , A_2 , and A_3 .

7.1 BDD Grammar

A simple grammar defining the BDD is:

$$B := 0 \mid 1 \mid a?B : B$$

and is subject to the following interpretation:

$$\begin{aligned} \llbracket 0 \rrbracket &= \text{Empty} \\ \llbracket 1 \rrbracket &= \text{Universe} \\ \llbracket a?B_1 : B_2 \rrbracket &= (a \wedge \llbracket B_1 \rrbracket) \vee (\neg a \wedge \llbracket B_2 \rrbracket) \end{aligned}$$

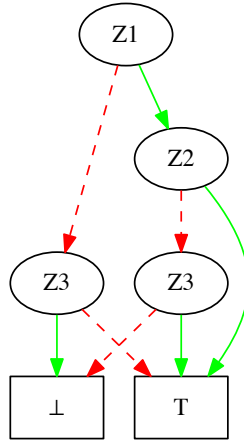


Figure 22: BDD for $(A_1 \wedge A_2) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge \neg A_3)$

Castagna [Cas16] introduces 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 algorithms for computing the BDDs which represent the common Boolean algebra operators are straightforward. Let B , B_1 , and B_2 denote BDDs, $B_1 = a_1?C_1 : D_1$ and $B_2 = a_2?C_2 : D_2$.

$$\begin{aligned}
1 \vee B &= 1 \\
B \vee 1 &= 1 \\
1 \wedge B &= B \\
B \wedge 1 &= B \\
0 \vee B &= B \\
B \vee 0 &= B \\
0 \wedge B &= 0 \\
B \wedge 0 &= 0 \\
0 \setminus B &= 0 \\
B \setminus 0 &= B \\
B \setminus 1 &= 0 \\
1 \setminus (a?B_1 : B_2) &= a?(1 \setminus B_1) : (1 \setminus B_2) \\
B_1 \vee B_2 &= \begin{cases} a_1?C_1 \vee C_2 : D_1 \vee D_2 & \text{for } a_1 = a_2 \\ a_1?C_1 \vee B_2 : D_1 \vee B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \vee C_2 : B_1 \vee D_2 & \text{for } a_1 > a_2 \end{cases} \\
B_1 \wedge B_2 &= \begin{cases} a_1?C_1 \wedge C_2 : D_1 \wedge D_2 & \text{for } a_1 = a_2 \\ a_1?C_1 \wedge B_2 : D_1 \wedge B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \wedge C_2 : B_1 \wedge D_2 & \text{for } a_1 > a_2 \end{cases} \\
B_1 \setminus B_2 &= \begin{cases} a_1?C_1 \setminus C_2 : D_1 \setminus D_2 & \text{for } a_1 = a_2 \\ a_1?C_1 \setminus B_2 : D_1 \setminus B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \setminus C_2 : B_1 \setminus D_2 & \text{for } a_1 > a_2 \end{cases}
\end{aligned}$$

Notice that 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

$$B_1 \circ B_2 = \begin{cases} a_1?C_1 \circ C_2 : D_1 \circ D_2 & \text{for } a_1 = a_2 \\ a_1?C_1 \circ B_2 : D_1 \circ B_2 & \text{for } a_1 < a_2 \\ a_2?B_1 \circ C_2 : B_1 \circ D_2 & \text{for } a_1 > a_2 \end{cases}$$

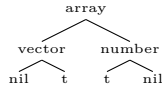


Figure 23: BDD representing (or number (and array (not vector)))

7.2 Boolean variable ordering

A remarkable fact about this representation subject to these rules is that any two equivalent Boolean expressions have exactly the same BDD representation, provided the terms (variable names) are totally ordered.[Knu09, Section 7.4.1 Page 73] For example the expression from Figure 22, $(A_1 \wedge A_2) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge \neg A_3)$ 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.

In order to assure an unambiguous representation of a type in terms of a BDD, it is necessary that the type specifiers be totally ordered. It really doesn't matter the ordering chosen, as long as it is a complete ordering. For any three type specifiers: A , B , and C , the following hold.

- If $A < B$ and $B < C$, then $A < C$.
- If $A < B$ then $B > A$.
- $A = B$ if and only if $A < B$ is false and $B < A$ is false.

Typically this is done by alphabetizing them. However, type specifiers in Common Lisp cannot necessarily be compared alphabetically. So the solution we use is to compare them alphabetically when possible. And tree-comparing the s-expressions otherwise. For example `integer` type specifier is itself a string, and the `(member 1 2 3)`. If the type specifiers being compared are of different types, then alphabetize them according to their type names: `list` precedes `symbol`. And for each type which might be encountered the natural ordering is used:

`list`: compare the first two elements which are not equal

`number`: compare with numerical `<`, `=`, or `>`.

`symbol`: compare the package name alphabetically, and if the symbols belong to the same package compare their printable names.

The Common Lisp code for the function used to compare two type specifiers is given in Appendix F.

7.3 Optimized BDD construction

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 Anderson [And99] is to intercept the BDD constructor function. This constructor should assure that it never returns two BDD which are `equal` but not `eq` to each other.

The BDD constructor takes three arguments: a type specifier (also called a label), and two BDDs called the left and right subtree. Several optimizations are in place to reduced the total number of trees.

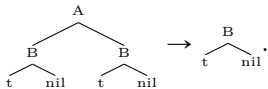
Table 3 lists the reductions which are performed by the BDD constructor function. The table contains a **Frequency** column which shows an estimated result of how often this case occurs in a typical run. It can be seen in the final line of the table that a new BDD allocation need only occur about 1.7% of the time.

Reduction	Frequency	Section
eq children	26%	7.3.1
found in table	63%	7.3.2
reduce and test eq	2.5%	7.3.3
found in table	7.0%	7.3.3
compare to child	0.5%	7.3.4
reduce to t or nil	0.03%	7.3.5
allocate bdd	1.7 %	7.3.2

Table 3: BDD reductions

7.3.1 Equal right and left subtrees

The most notable optimization is that if the left and right subtrees are identical then simply return one of them, without allocating a new tree [And99].



7.3.2 Caching BDDs

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 tree can be returned. For this to work, we associate each new tree created with an integer, and create a hash key which is a list (a triple) of the type specifier followed by two integers corresponding to the left and right trees.

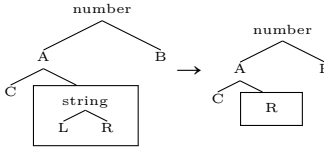
7.3.3 Reduction in the presence of subtypes

When the nodes of the BDD represent types, other optimizations can be 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 4. Each of these optimizations follows a similar pattern: when constructing a BDD with label X , search in either the left or right subtree to find a subtree, $\begin{matrix} Y \\ \swarrow \searrow \\ L \quad R \end{matrix}$, whose label is Y having left and right subtrees L and R . If X and Y have a particular relation, then the $\begin{matrix} Y \\ \swarrow \searrow \\ L \quad R \end{matrix}$ tree reduces either to L or R .

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

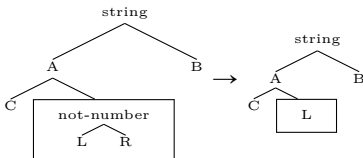
Table 4: BDD optimizations

Case 1: If $X \cap Y = \emptyset$ and $\begin{matrix} Y \\ \swarrow \searrow \\ L \quad R \end{matrix}$ appears in $left(X)$, then $\begin{matrix} Y \\ \swarrow \searrow \\ L \quad R \end{matrix}$ reduces to R .

For example: If $X = number$ and $Y = string$, we have 

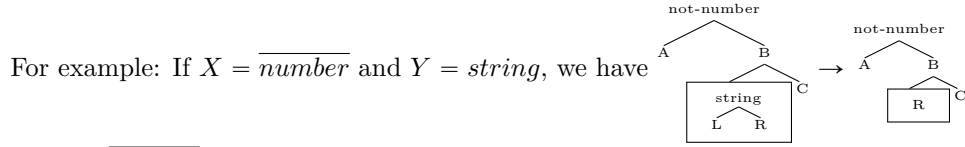
because $(number \cap string = \emptyset)$.

Case 2: If $X \cap \bar{Y} = \emptyset$ and $\begin{matrix} Y \\ \swarrow \searrow \\ L \quad R \end{matrix}$ appears $left(X)$, then $\begin{matrix} Y \\ \swarrow \searrow \\ L \quad R \end{matrix}$ reduces to L .

For example: If $X = string$ and $Y = \overline{number}$, we have 

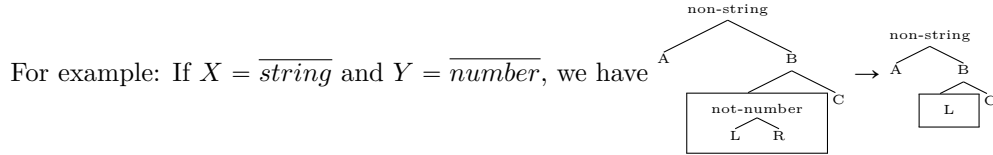
because $string \cap \overline{\overline{number}} = \emptyset$.

Case 3: If $\overline{X} \cap Y = \emptyset$ and $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ appears *right*(X), then $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ reduces to R .



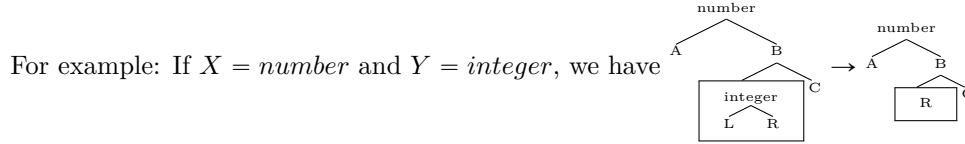
because $\overline{\text{number}} \cap \text{string} = \emptyset$.

Case 4: If $\overline{X} \cap \overline{Y} = \emptyset$ and $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ appears *right*(X), then $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ reduces to L .



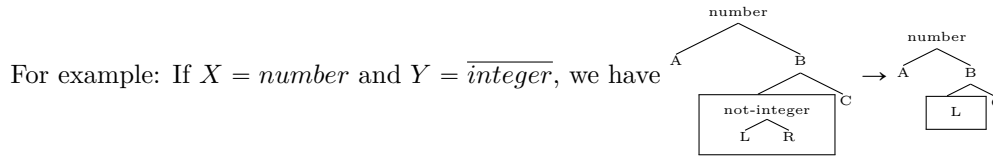
because $\overline{\text{string}} \cap \overline{\text{number}} = \emptyset$.

Case 5: If $Y \subset X$ and $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ appears in *right*(X), then $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ reduces to R .



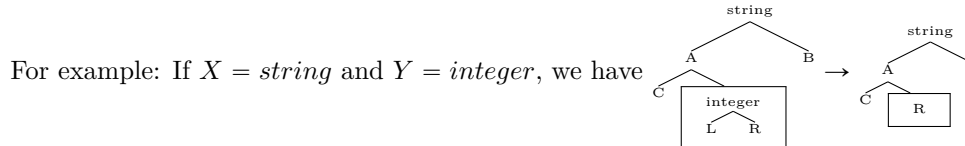
because $\text{integer} \subset \text{number}$.

Case 6: If $\overline{Y} \subset X$ and $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ appears in *right*(X), then $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ reduces to L .



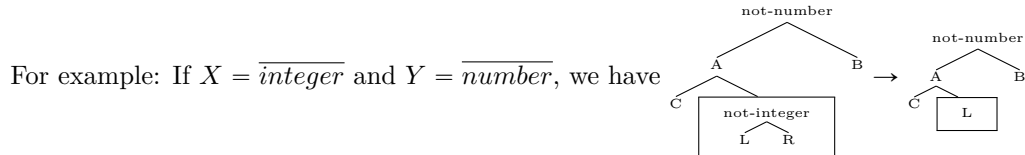
because $\overline{\text{integer}} \subset \text{number}$.

Case 7: If $Y \subset \overline{X}$ and $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ appears in *left*(X), then $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ reduces to R .



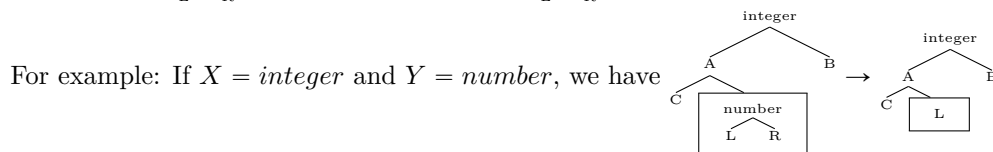
because $\text{integer} \subset \overline{\text{string}}$.

Case 8: If $\overline{Y} \subset \overline{X}$ and $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ appears in *left*(X), then $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ reduces to L .



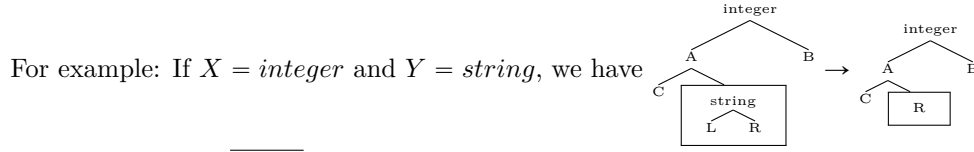
because $\overline{\text{integer}} \subset \overline{\text{number}}$.

Case 9: If $X \subset Y$ and $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ appears in *left*(X), then $\underset{L}{\overset{Y}{\curvearrowright}}\underset{R}{}$ reduces to L .



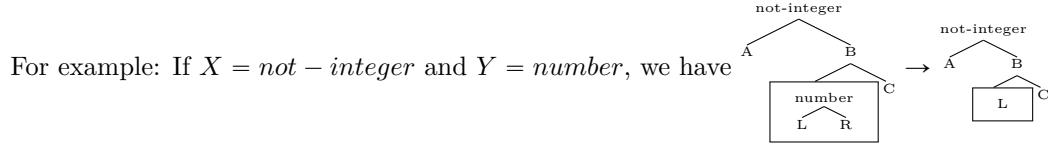
because $\text{integer} \subset \text{number}$.

Case 10: If $X \subset \bar{Y}$ and $\begin{matrix} Y \\ L \quad R \end{matrix}$ appears in $left(X)$, then $\begin{matrix} Y \\ L \quad R \end{matrix}$ reduces to R .



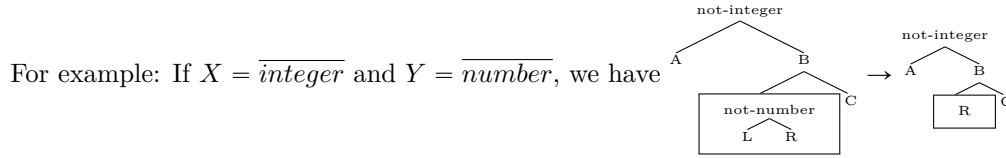
because $integer \subset \overline{string}$.

Case 11: If $\bar{X} \subset Y$ and $\begin{matrix} Y \\ L \quad R \end{matrix}$ appears in $right(X)$, then $\begin{matrix} Y \\ L \quad R \end{matrix}$ reduces to L .



because $\overline{\overline{integer}} \subset number$.

Case 12: If $\bar{X} \subset \bar{Y}$ and $\begin{matrix} Y \\ L \quad R \end{matrix}$ appears in $right(X)$, then $\begin{matrix} Y \\ L \quad R \end{matrix}$ reduces to R .

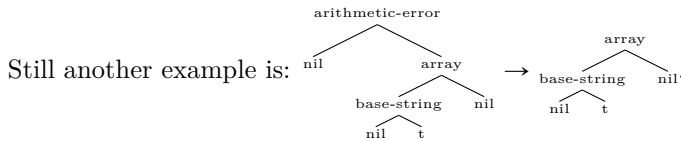
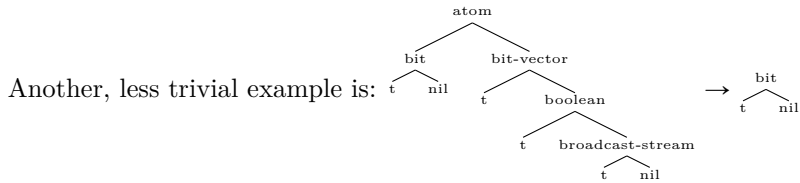


because $\overline{\overline{integer}} \subset \overline{\overline{number}}$.

7.3.4 Reduction to subtree

The list of reductions described in Section 7.3.3 fails to apply on several important cases. Most notably there is no reduction on a case where the root node itself needs to be eliminated. For

example, since $vector \subset array$ we would like the following reduction:

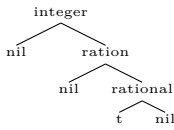


The solution in place in this case is that before constructing a new tree, we first ask whether the resulting tree is type-equivalent to either the left or right subtrees. If so, we simply return the appropriate subtree without allocating the parent tree.

Because of the way the caching is done, this expensive type-equivalence check need only be done once per triple. Thereafter, the result is in the hash table, and it will be discovered as discussed in Section 7.3.2.

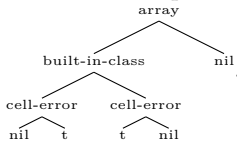
7.3.5 More complex type relations

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 identically (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. To call the `subtypep` function we must generate a serialization of the BDD which is conforming with the Common Lisp type specifier syntax. See Section 7.5 for details of this serialization.

Another example of a BDD which reduces to `nil` is:



Again this check is expensive, but the expense is mitigated in that the result is cached.

7.4 Other BDD related optimizations

It was observed doing profiling analysis that by far the largest consumer of computation time in BDD operations while calculating disjoint type decomposition is the `subtypep` function. The function is called orders of magnitude more often than any other function. We observed that there seem to be three sources of calls to this function: direct calls from within `smarter-subtypep`, calls from `disjoint-types-p`, and recursive calls from within `subtypep` itself.

Since we have not attempted to modify the implementation of the Common Lisp we are using (sbcl), we have not attempted to change the implementation of `subtypep` to make it more intelligent. So the only way to reduce the number of recursive calls to the function, is simply to reduce the total number of calls to the function.

7.4.1 Calls to smarter-subtypep

The main entry point to the `subtypep` function in our program is through the `smarter-subtypep` function. This function was written to *patch* some of the shortcomings of `subtypep`. The architecture of `smarter-subtypep` is that it first calls `subtypep` and returns the values it returns as long as the second value is `t`. However, if the second value is `nil`, meaning that `subtypep` was not able to determine the subtype relation, when we call a helper function, which tries several different formulations, and memoizes its result if possible—possible means that it is called within a dynamic scope which proves a particular dynamic variable used as a memoization table. The code is shown in Appendix G.

7.4.2 Calls from disjoint-types-p

A well known technique to determine whether two types are disjoint is to ask whether their intersection is empty, such as with the following function `poor-disjoint-types-p`.

```
(defun poor-disjoint-types-p (T1 T2)
  (subtypep '(and ,T1 T2) nil))
```

The function, like `subtypep`, returns a second value indicating whether the Boolean semantic of the first return can be trusted. Unfortunately, `subtypep` called in this way returns `nil`, `nil` far too often. The consequence is that algorithms such as the one described in Sections 8.1 and 8.2 must assume worst case that the types intersect.

TODO document cases where `disjoint-types-p` finds results which `subtypep` is not able to find.

The function, `disjoint-types-p`, whose code is given in Appendix H is the function we have developed to answer this question. The function incorporates both memoization techniques, as well as several set theoretical relations to determine the disjoint relation in the case `subtypep` is unable to determine the result. It makes use of the function `dispatch:specializer-intersections` which returns the list of least specific subclasses of two given classes.

7.5 Serializing a BDD

The BDD can be serialized to a CL type-specifier in many different ways. Two such serializations are as DNF (disjunctive normal form) and ITENF (if then else normal form). Consider the type specifier: `(or (and sequence (not array)) number (and (not sequence) array))`

```
TEST> (bdd-to-dnf (bdd '(or (and sequence (not array))
                           number
                           (and (not sequence) array))))
(or (and ARRAY (not SEQUENCE))
    NUMBER
    (and (not ARRAY) SEQUENCE))
TEST> (bdd-to-expr (bdd '(or (and sequence (not array))
                             number
                             (and (not sequence) array))))
(or (and ARRAY (not SEQUENCE))
    (and (not ARRAY) (or NUMBER (and (not NUMBER) SEQUENCE))))
TEST>
```

The DNF is an OR of ANDs in which each AND expression is a simple type expression or the NOT of such. The IFENF is of the form `(or (and X Y) (and (not X) Z))`.

The Common Lisp code for `bdd-to-dnf` can be found in Appendix I.

The IFENF closely matches the tree structure of the BDD. The IFENF is intended to be the most efficient to calculate.

The Common Lisp code for `bdd-to-dnf` can be found in Appendix J.

7.6 Applying BDDs to type checking

The BDD can be used to re-implement `typep`. Such an implementation, guarantees each type check is done only once, even if the same type specifier component appears multiple times in the given type specifier. *E.g.*, `sequence` appears twice in the type specifier `(or (and sequence (not array)) number (and (not sequence) array))`, but when checking an object against this type specifier using the BDD based approach, we are guaranteed that the object is only checked once to be a sequence. This may not be important for a type like `sequence` which is fast to check, but some user defined types may be arbitrarily expensive to check, such as `(satisfies F)`.

7.6.1 Run-time type checking

Similar semantics to `CL:TYPEP` but takes a BDD or a Common Lisp type-specifier. If a Common Lisp type specifier is given as second argument, it is interpreted as the corresponding BDD object, via a call to the function `BDD`. Returns `T` if the object is an element of the specified type, Returns `NIL` otherwise.


```

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

```

7.6.2 Compile time

This function has the same calling syntax as `CL:TYPEP`, but uses a BDD based algorithm.

```

(defun bdd-typep (obj type-specifier)
  (bdd-type-p obj (bdd type-specifier)))

(define-compiler-macro bdd-typep (obj type-specifier)
  (typecase type-specifier
    ((cons (eql quote))
     (bdd-with-new-hash
      (lambda (&aux (bdd (bdd (cadr type-specifier))))
        '(funcall ,(bdd-to-if-then-else-3 bdd (gensym)) ,obj))))
    (t
     '(typep ,obj ,type-specifier))))

```

The compiler macro checks call-site syntax for something like `(bdd-typep my-object '(or (and sequence (not array)) number (and (not sequence) array)))` and expands (at compile time) to something like the following.

```

(funcall
 (lambda (#:g738)
  (labels ((#:g742 ()
            (if (typep #:g738 'sequence)
                t
                nil)))
         (#:g741 ()
          (if (typep #:g738 'number)
              t
              (#:g742))))
        (#:g740 ()
         (if (typep #:g738 'sequence)
             nil
             t)))
       (#:g739 ()
        (if (typep #:g738 'array)
            (#:g740)
            (#:g741))))
      (#:g739)))
 my-object)

```

If the compiler works hard enough at in-lining local function calls it can transform this code to the following:

```
(funcall (lambda (x)
  (if (typep x 'array)
      (if (typep x 'sequence)
          nil
          t)
      (if (typep x 'number)
          t
          (if (typep x 'sequence)
              t
              nil))))))
my-object)
```

One disadvantage of expanding the code as shown here to nested `if` calls, is that the code size may be large. Even if the execution time is linear as a function of the number of variables, the code size is exponential in worst case. Notice that the function calls in the `labels` based implementation, are all in tail position, and can thus be optimized away.

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

Notice in the if-then-else tree that no possible value of `x` will lead to `(typep x 'sequence)` being evaluated more than once.

Caveat: reordering type specifiers within AND and OR. CL spec example of `(and integer (satisfies oddp))` does not work if re-ordered. We have no good solution for this. Possible non-solutions, as none of them actually work.

- always *sort* `(satisfies ...)` to the end. Does not work because `deftype` might contain a `satisfies`, no compliant way to expand a `deftype`, no way to order `(satisfies F)` vs checks `(satisfies G)`
- re-interpret `(satisfies F)` as `(satisfies (ignore-errors F))`
- others?

8 Type decomposition using BDDs

Using the BDD data structure along with the algorithms described in Section 7 we can efficiently represent and manipulate Common Lisp type specifiers. We may now programmatically represent Common Lisp types largely independent of the actual type specifier representation. For example in Common Lisp the 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 7.1. Moreover, unions, intersections, and relative complements of Common Lisp type specifiers can be calculated using the reduction BDD manipulation rules as well.

8.1 Improving the RTE algorithm using BDDs

We may revisit the algorithms described in Sections 4, 5, and 6, but this time use the BDD as the data structure to represent the Common Lisp type specifications rather than using the s-expression.

To decompose a set of types using the BDD approach we start with the list of type specifiers, eliminate the ones which specify the empty type, and proceed as follows.

Seed the set, S , with one of the types. Iterate p through the remaining types, represented as BDDs. For each p , iterate q through the elements of S . Calculate a slice-set of at most three elements by calculating $\{p \cap q, p \cap \bar{q}, \bar{p} \cap q\} \setminus \{\emptyset\}$, discarding any that are the empty type and accumulating the non-empty types. After q has traversed completely through S , replace S by the union of the slice-sets, and proceed with the next value of p . When calculating this union of slice-sets, it is important to guard against duplicate elements, as some slice-sets may contain common elements. After p finishes iterating through the given set of types, S remains the type decomposition.

The Common Lisp function is implemented elegantly using the `reduce` function. The Common Lisp code can be found in Appendix C.

Using BDDs in this algorithm allows certain checks to be made easily. For example, two types are equal if they are the same object (pointer comparison). A type is empty if it is identically the empty type (pointer comparison). Two BDDs which share equal common subtrees, actually share the objects (shared pointers).

8.2 Improving the graph based algorithm using BDDs

We re-implemented the graph algorithm described in Section 5. The implementation is roughly 110 lines of Common Lisp code—roughly 3 times the size of the brute force algorithm described in Section 4.

The skeleton of the code is straightforward and is shown in Appendix D.

9 Performance of type decomposition

Sections 4, 5, and 6 explained three different algorithms for calculating type decomposition. We look here at some performance characteristics of the three algorithms. Section 9.1 describes content and construction of data sets used both for optimizing and tuning the algorithm from Section 8.2 and also for comparing the relative performances for the various algorithms. Section 9.2 describes how the tuning parameters were selected for the optimized BDD based graph algorithm. Section 9.3 describes the relative performance of the `rte` algorithm from Section 4 using (1) its s-expression based implementation and also (2) its BDD based implementation; the graph based algorithm from Section 8.2 including (3) its s-expression based implementation and (4) its BDD based implementation; and (5) the SAT like algorithm from Section 6.

9.1 Performance Test Setup

It has been observed that different sets of initially given types evoke different performance behavior from the various algorithms. It is still an area of research to explain and characterize this behavior. It would be ideal if from a given set of types, it were possible to predict with some amount of certainty the time required to calculate the type decomposition. This is not currently possible. What we have done instead put together several sets of types which can be used as *pools* or starting sets for the decomposition.

9.1.1 Subtypes of number

This pool contains the type specifiers for all the subtypes of `CL:NUMBER` whose symbol name comes from the "CL" package:

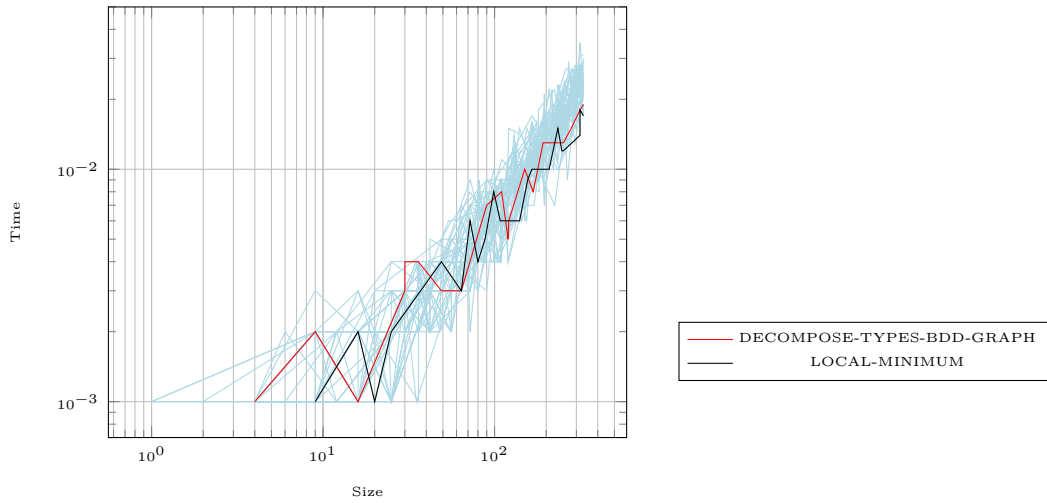


Figure 24: Tuning algorithm with subtypes of number

(short-float array-total-size float-radix ratio `rational` bit nil
array-rank integer long-float real double-float bignum signed-byte
float unsigned-byte single-float char-code number float-digits
fixnum char-int `complex`)

This pool was used to generate the graph shown in Figure 24.

9.1.2 Subtypes of condition

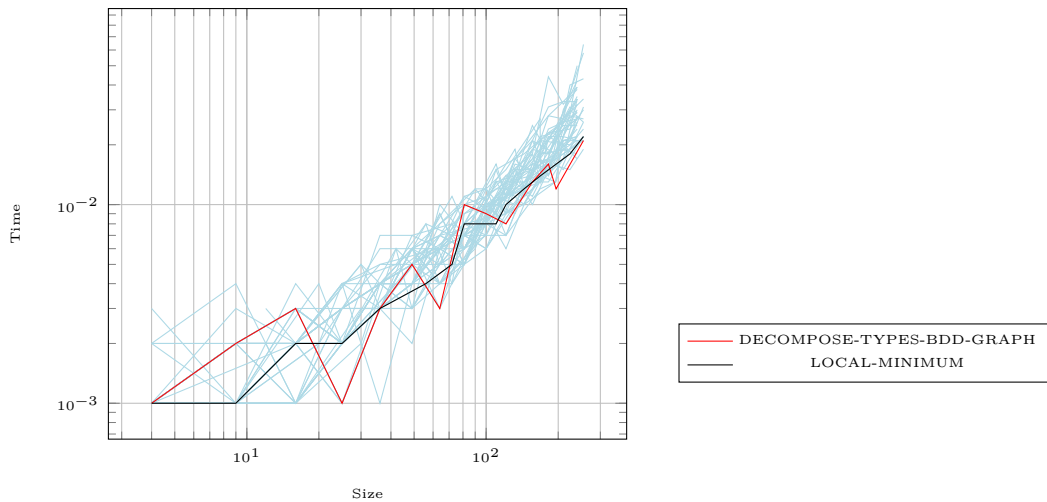


Figure 25: Tuning algorithm with subtypes of condition

This pool contains the type specifiers for all the subtypes of `CL:CONDITION` whose symbol name comes from the "CL" package:

(simple-error storage-condition file-error control-error
serious-condition condition division-by-zero nil parse-error
simple-type-error `error` package-error program-error stream-error
unbound-variable undefined-function floating-point-inexact
cell-error floating-point-overflow floating-point-invalid-operation)

```

simple-warning print-not-readable type-error
floating-point-underflow style-warning end-of-file unbound-slot
reader-error simple-condition arithmetic-error warning)

```

This pool was used to generate the graph shown in Figure 25.

9.1.3 Subtypes of number or condition

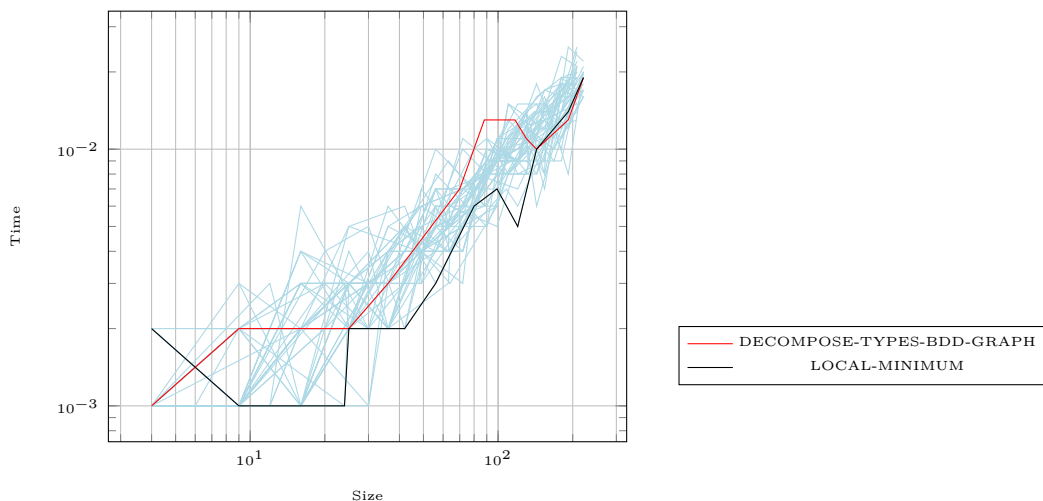


Figure 26: Tuning algorithm with subtypes of number or condition

This pool contains the type specifiers for all the subtypes of `CL:NUMBER` and all the subtypes of `CL:CONDITION` whose symbol name comes from the "CL" package. It is the union of the two sets from Section 9.1.1 and 9.1.2.

This pool was used to generate the graph shown in Figure 26.

9.1.4 Subtypes of t

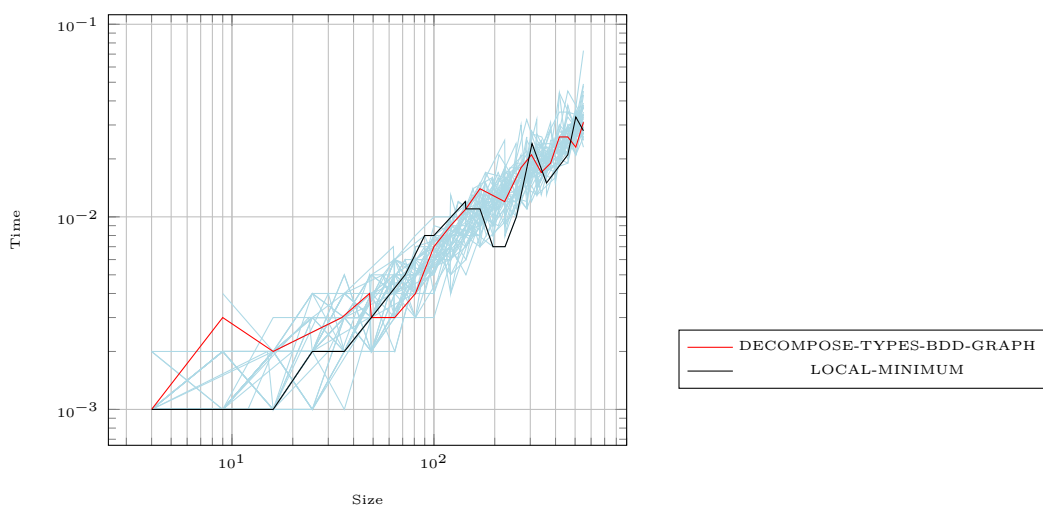


Figure 27: Tuning algorithm with subtypes of t

This pool contains the type specifiers for all the subtypes of CL:T whose symbol name comes from the "CL" package:

```
(simple-error short-float storage-condition file-error
array-total-size float-radix ratio character restart package
rational control-error t vector method serious-condition atom
generic-function condition bit readtable division-by-zero nil
parse-error null base-string base-char simple-type-error
synonym-stream error stream package-error array-rank pathname-host
standard-object integer simple-base-string keyword boolean
program-error pathname-directory file-stream stream-error
unbound-variable sequence undefined-function long-float real cons
floating-point-inexact double-float concatenated-stream bit-vector
standard-method cell-error floating-point-overflow hash-table
method-combination pathname-name floating-point-invalid-operation
simple-warning bignum signed-byte compiled-function float array
unsigned-byte single-float symbol pathname-device char-code
print-not-readable type-error function simple-array
floating-point-underflow simple-string number simple-bit-vector
style-warning standard-char echo-stream standard-class
logical-pathname float-digits structure-object pathname-version
two-way-stream fixnum built-in-class end-of-file unbound-slot
extended-char reader-error char-int string-stream pathname
random-state standard-generic-function simple-condition class list
structure-class arithmetic-error pathname-type broadcast-stream
warning complex simple-vector string)
```

This pool was used to generate the graph shown in Figure 27.

9.1.5 Subtypes in SB-PCL

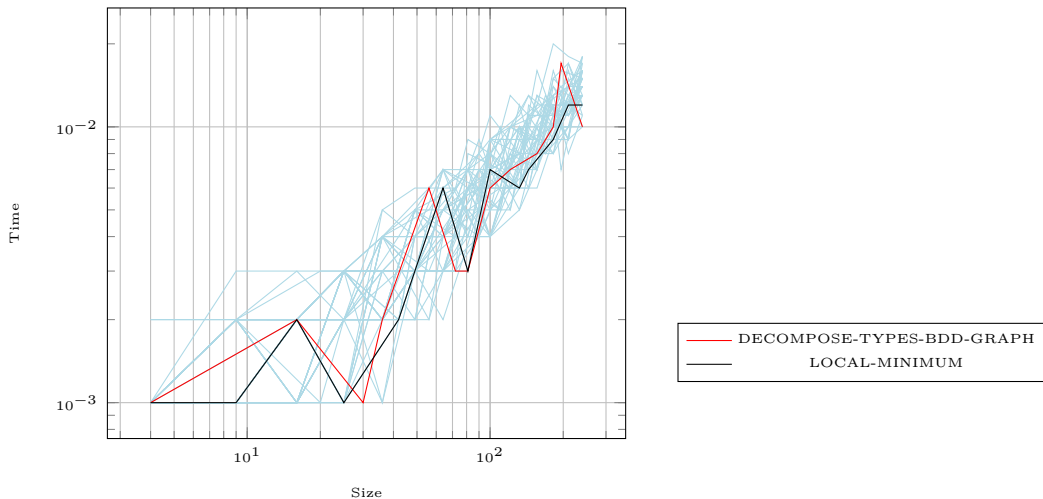


Figure 28: Tuning algorithm with PCL types

This pool contains the type specifiers for all the types whose symbol is in the "SB-PCL" package:

```
(sb-pcl:system-class sb-mop:standard-slot-definition
sb-mop:funcallable-standard-class sb-mop:standard-reader-method
sb-mop:specializer sb-mop:standard-writer-method
sb-mop:slot-definition sb-mop:eql-specializer
sb-mop:standard-accessor-method sb-mop:forward-referenced-class)
```

```

sb-mop:standard-direct-slot-definition
sb-mop:effective-slot-definition sb-mop:funcallable-standard-object
sb-mop:direct-slot-definition
sb-mop:standard-effective-slot-definition)

```

This pool was used to generate the graph shown in Figure 28.

9.1.6 Specified Common Lisp types

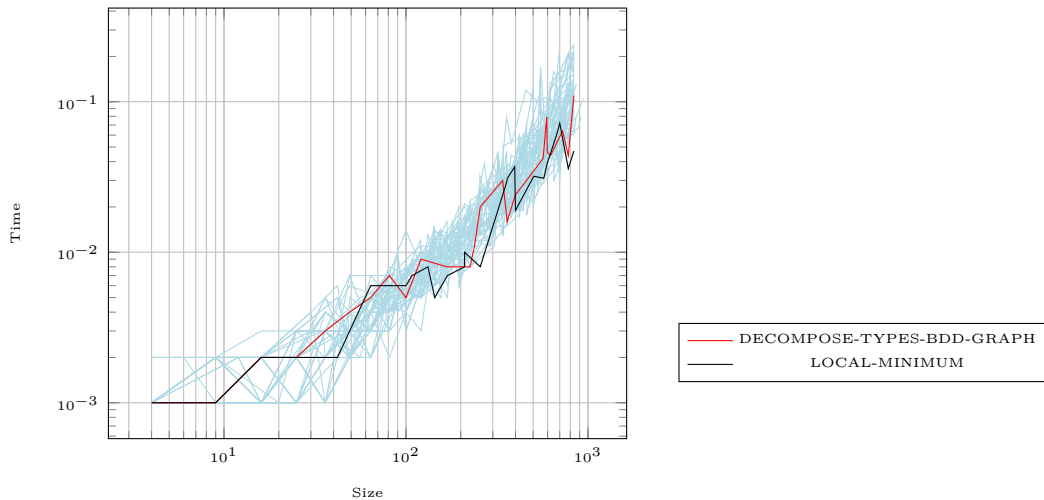


Figure 29: Tuning algorithm with specified CL types

This pool contains the type specifiers for all the types directly described by the Common Lisp specification. These are the 97 types listed in **Figure 4-2. Standardized Atomic Type Specifiers** from the Common Lisp specification [Ans94, Section 4.2.3 Type Specifiers], which every compliant Common Lisp implementation must support.

(arithmetic-error	function	simple-condition
array	generic-function	simple-error
atom	hash-table	simple-string
base-char	integer	simple-type-error
base-string	keyword	simple-vector
bignum	list	simple-warning
bit	logical-pathname	single-float
bit-vector	long-float	standard-char
broadcast-stream	method	standard-class
built-in-class	method-combination	standard-generic-function
cell-error	nil	standard-method
character	null	standard-object
class	number	storage-condition
compiled-function	package	stream
complex	package-error	stream-error
concatenated-stream	parse-error	string
condition	pathname	string-stream
cons	print-not-readable	structure-class
control-error	program-error	structure-object
division-by-zero	random-state	style-warning
double-float	ratio	symbol
echo-stream	rational	synonym-stream
end-of-file	reader-error	t
error	readtable	two-way-stream

```

extended-char      real          type-error
file-error        restart       unbound-slot
file-stream       sequence      unbound-variable
fixnum            serious-condition  undefined-function
float             short-float    unsigned-byte
floating-point-inexact signed-byte  vector
floating-point-invalid-operation simple-array  warning
floating-point-overflow simple-base-string
floating-point-underflow simple-bit-vector )

```

This pool was used to generate the graph shown in Figure 29.

9.1.7 Intersections and Unions

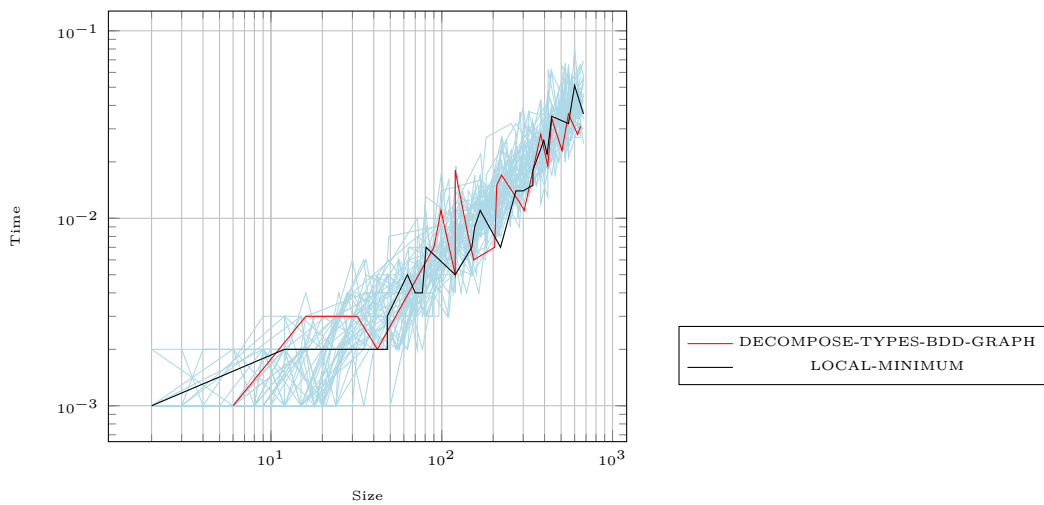


Figure 30: Tuning algorithm with AND and OR combinations

This pool contains the type specifiers which are AND and OR combinations of the types in Section 9.1.6. Starting from this list, we randomly generated type specifiers using `and` and `or` combinations of names from this list such as the following:

```

(arithmetic-error function
 (and arithmetic-error function)
 (or arithmetic-error function)
 array
 (or function array)
 sequence
 (or function sequence)
 ...)

```

This pool was used to generate the graph shown in Figure 30.

9.1.8 Subtypes of fixnum using member

This pool contains type specifiers of various subtypes of `FIXNUM` all of the same form, using the `(member ...)` syntax.

```

((member 2 6 7 9) (member 0 2 7 8 9) (member 0 2 5 6)
 (member 0 1 2 4 6 8 10) (member 0 2 3 4 5 6 8 9) (member 1 2 3 4 5 6 10)
 (member 3 5 6 7 8) (member 0 1 3 5 8 9) (member 1 2 4 5 8 10)
 (member 0 2 5 6 8 9 10) (member 0 2 3 4) (member 1 4 5 6 7 9 10)

```

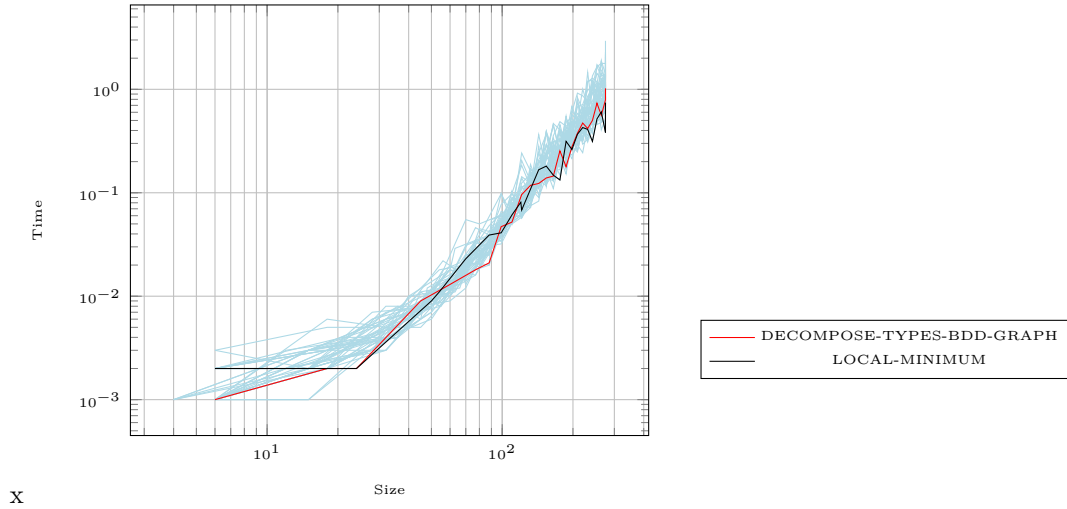



Figure 31: Tuning algorithm with MEMBER types

```
(member 0 2 3 4 5 7 9) (member 2 4 5) (member 3 4 5 9)
(member 0 1 2 3 6 10) (member 0 1 3 4 5 6 9 10) (member 1 2 8 10)
(member 1 3 6 7 8 10) (member 0 1 2 4 10) (member 0 1 2 3 4 6 7 9)
(member 0 1 2 4 5 6 7 8 10) (member 0 4 5 7 9 10) (member 1 3 4 6 9)
(member 1 3 6 7 8 9 10))
```

This pool was used to generate the graph shown in Figure 31.

9.2 Tuning the BDD based graph algorithm

Section 5 explains how to represent the problem of set decomposition as a graph problem. The section explains how to construct the graph, and several deconstruction operations. What is not explained is how to deconstruct the graph efficiently in terms of execution time. From the high level view once the graph has been constructed, the algorithm proceeds by visiting each node and apply some of a set of possible operations. In fact one in particular of these operations, the subset extraction, there are two variations described in 5.2.2 and 5.2.3.

The natural question which arises is what is the best way to assemble these pieces into an efficient algorithm? In this section (Section 9.2) we describe some experimentation we did to attempt to determine an approach which is reasonable for a wide range of data sets.

Break subset: `strict` *vs.* `relaxed`, explained in Sections 5.2.2 and 5.2.3.

Break loop: `yes` *vs.* `no`, explained in Section 5.2.5.

Iteration topology: \forall node \forall operation *vs.* \forall operation \forall node, explained in Section 5.3.

Recursive: `yes` *vs.* `no`, explained in Section 5.2.6 and 5.3

Node visitation order: `"SHUFFLE"`, `"INCREASING-CONNECTIONS"`, `"DECREASING-CONNECTIONS"`, `"BOTTOM-TO-TOP"`, `"TOP-TO-BOTTOM"`, explained in Section 5.3.

Inclusion: `implicit` *vs.* `explicit`, explained in Section 5.2.1.

Empty set: `late-check`, *vs.* `correct-label` *vs.* `correct-connection`, explained in Section 5.2.6.

Section 9.1 details several pools which were constructed to test various aspects of the disjoint type decomposition algorithm.

Parameter	value
Break subset	relaxed
Break loop	no
Iteration topology	operations per node
Recursive	yes
Node visitation order	BOTTOM-TO-TOP
Inclusion	not tested
Empty set	not tested

Figure 32: Experimentally determined best parameter values

We have devised test suits which run on each of the data pools testing the performance of the graph based algorithm as a function of each of the parameters described above, with the exception of *inclusion* and *empty-set*. At this point in our experimentation we have not yet included these parameter in our performance testing.

Some of the parameter options don't make sense together. For example, we cannot use *break subset=relaxed* along with *break loop=no* because this would result in some graphs which cannot be reduced at all. Of the remaining possible combinations we have constructed 45 different variations of the input parameters. Figures 24 through 31 show the performance results in terms of input size *vs.* calculation time. The red curves indicate the default performance of the algorithm (having been tuned by the analysis explained below). The black curve indicates the minimum time (best possible) performance for this data pool. (The following paragraphs explain what we mean by *best performance*). The plots show that although the default parameters do not yield the best results on any one of the data sets, the results do seem reasonably good.

For each of the 8 test pools, 45 curves were plotted. In each case we wanted to determine the *fitness* of the 45 curves. There are potentially many different ways of judging the curves for *fitness*.

As all of our sampled data is positive (calculation time is always non-negative), we elected to derive a *norm* for the curves. *I.e.*, we can sort the curves from best to worst by sorting them in increasing order of norm. Small norm means fast calculation time over the range sampled, and large norm means slow calculation time. But even then, there are many ways of calculating a norm of sampled functions. A simple Google search of the terms "metrics for sampled functions" finds 726,000 results of scholarly papers.

Three possible norms are the following:

- Average value sum of y-values divided by number of samples.
- Average value based on numerical integral divided by size of domain
- RMS (root means square) distance from the extrapolated minimum curve.

The most reasonable of these for our needs is the average. The other two were dismissed because they don't give equal weighting to all samples. We believed that the latter of the two would give inappropriately large weighting to large values of calculation time *vs.* sample size.

Once the norm of each curve was calculated, (45 for each pool, over 8 pools), we calculated a *student score* for each curve in the same pool. If μ is the average norm for a pool and σ is the standard deviation of the norms, then the student score, Z , is defined as follows. $Z(f) = \frac{\|f\| - \mu}{\sigma}$. By this definition the smallest (most negative) Z indicates the *best* curve, *i.e.*, the set of parameters resulting in the least calculation time within the pool. Using the student score, allows us to compare results from different pools, as each score has already been *normalize* by the standard deviation of its respective pool.

Once each curve has been scored (via the student score), we then proceeded to determine which sets of parameters lead to the best score. We considered, one by one, each of the parameters, and for each possible value of the parameter, we calculated at the average student score (across all 8

pools). For example: there are two possible values for *break sub*: *strict* and *relaxed*. The average student score for *break sub=strict* was 0.62518245 and for *break sub=relaxed* was -0.28656462. From this we infer that *relaxed* is a better choice than *strict*. Figure 32 shows the experimentally determined best values for the parameters of the graph based algorithm.

Given this experimentally determined set of default values for the parameters, we have a candidate default behavior for the function. The performance of this default function can be seen relative to the other possible parameterizations can be seen in Figures 24 through 31. Each figure shows the performance of the default function and the performance of the per-pool best function along with the 43 other functions for that data pool. The red curve indicates the default performance of the algorithm. The black curve indicates the minimum time (best possible) performance for this data pool. The plots show that although the default parameters do not yield the best results on any one of the data sets, the results do seem reasonably good.

Ranking Results		
Parameter	Value	Average Score
Break subset	relaxed	-0.28656462
	strict	0.62518245
Break loop	no	-0.49688414
	yes	0.2650052
Iteration topology	node first	0.0048906407
	operation first	0.042273182
Recursive	yes	-0.29216018
	no	0.14136852
Node visitation order	BOTTOM-TO-TOP	-0.2333628
	DECREASING-CONNECTIONS	-0.17516118
	SHUFFLE	0.10092279
	INCREASING-CONNECTIONS	0.12181535
	TOP-TO-BOTTOM	0.27254203

9.3 Analysis of Performance Tests

Figures 33, 37, 39, 41, 43, 45, and 47 contrast the the five effective algorithms in terms of execution time vs sample size. Sample size is the integer product of the number of input types multiplied by the number of output types. This particular plot was chosen as it heuristically seems to be the best to demonstrate the relative performance for the different data sets.

The type specifiers used in Figure 33 are those designating all the subtypes of `NUMBER` whose symbol name comes from the "CL" package as explained in Section 9.1.1.

The type specifiers used in Figure 35 are those designating all the subtypes of `CONDITION` whose symbol name comes from the "CL" package as explained in Section 9.1.2.

The type specifiers used in Figure 37 are the set of symbols in the "CL" package specifying types which are a subtype of either `NUMBER` or `CONDITION` as explained in Section 9.1.3.

The type specifiers used in Figure 39 are those designating all the subtypes of `CL:T` whose symbol name comes from the "CL" package as explained in Section 9.1.4.

The type specifiers used in Figure 41 are those designating all the types named in the `SB-PCL` package as explained in Section 9.1.5.

The type specifiers used in Figure 43 are those explained in Section 9.1.6.

The type specifiers used in Figure 45 are those explained in Section 9.1.7.

The type specifiers used in Figure 47 are those explained in Section 9.1.8.

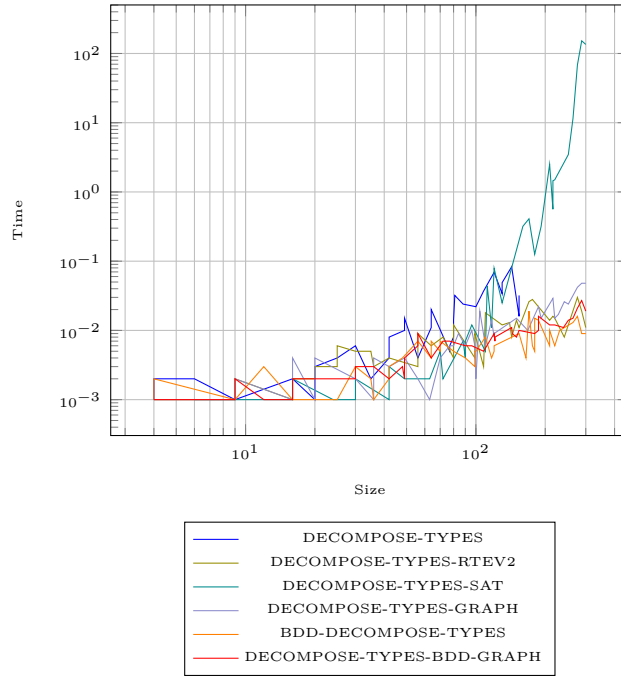


Figure 33: Performance using subtypes of number

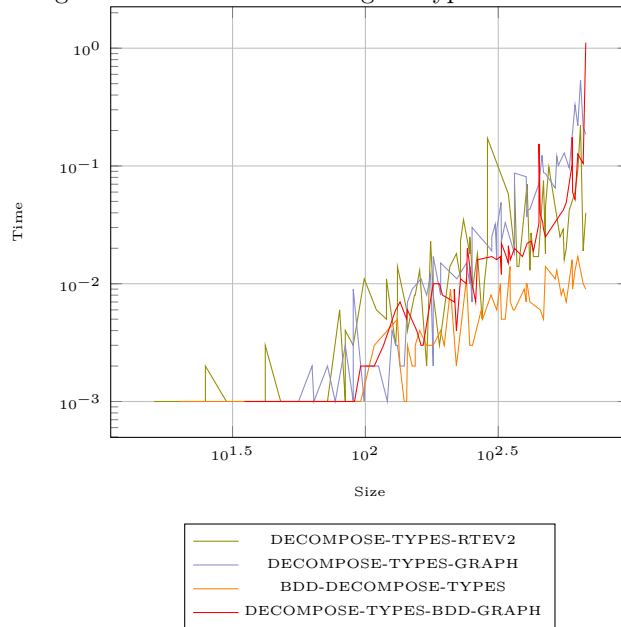


Figure 34: Performance using subtypes of number (best three)

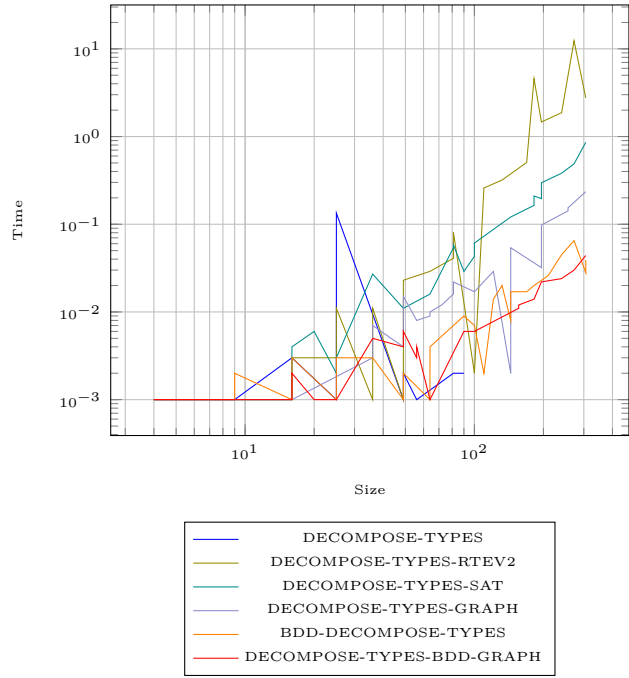


Figure 35: Performance using subtypes of condition

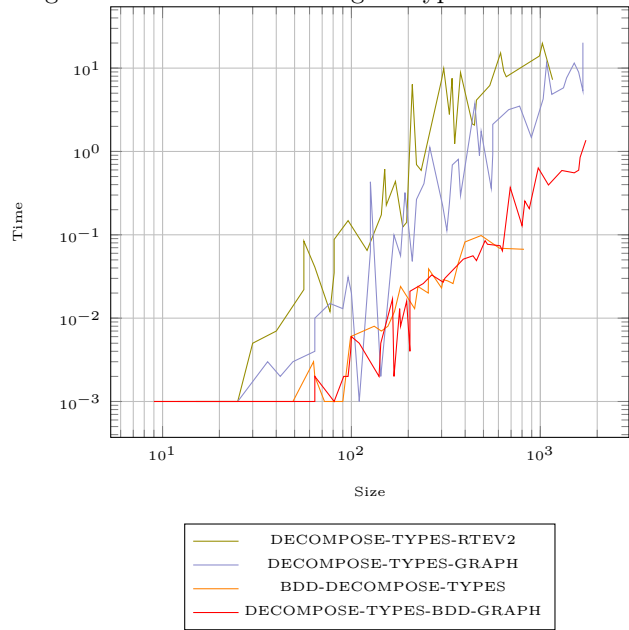


Figure 36: Performance using subtypes of condition (best three)

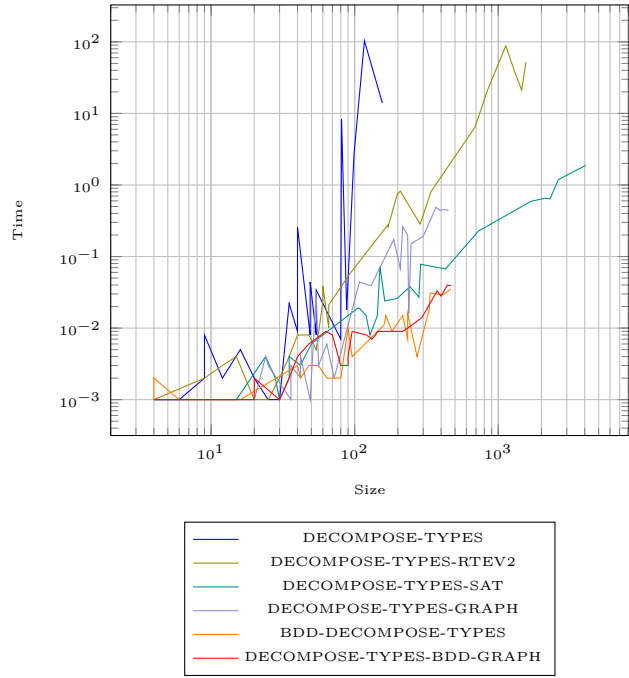


Figure 37: Performance using subtypes of number-or-condition

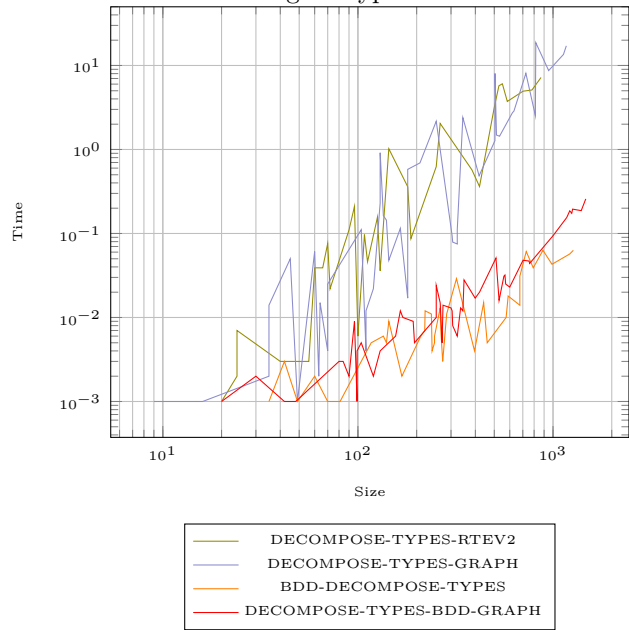


Figure 38: Performance using subtypes of number-or-condition (best three)

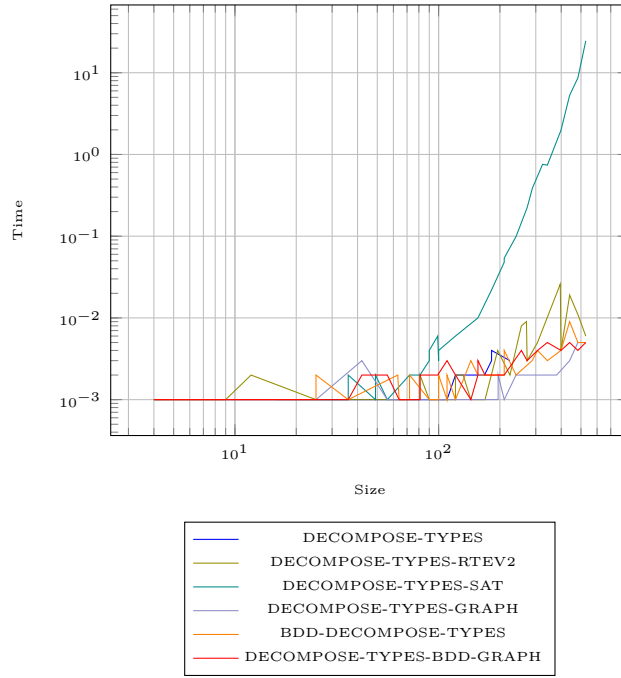


Figure 39: Performance using subtypes of t

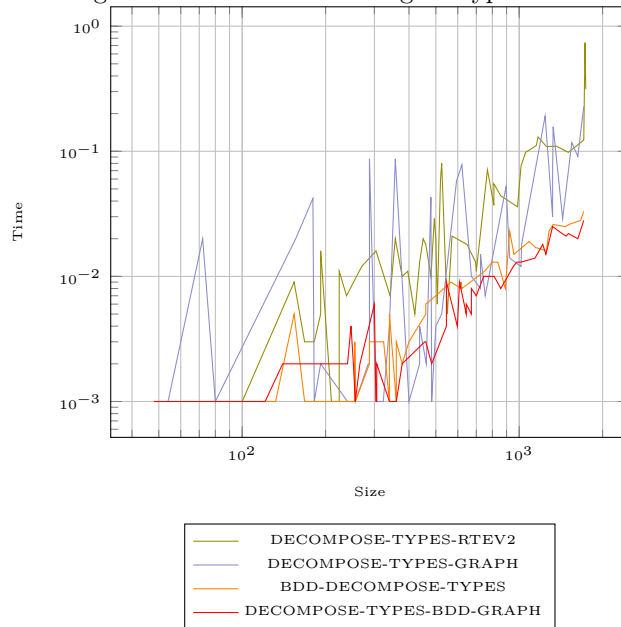


Figure 40: Performance using subtypes of t (best three)

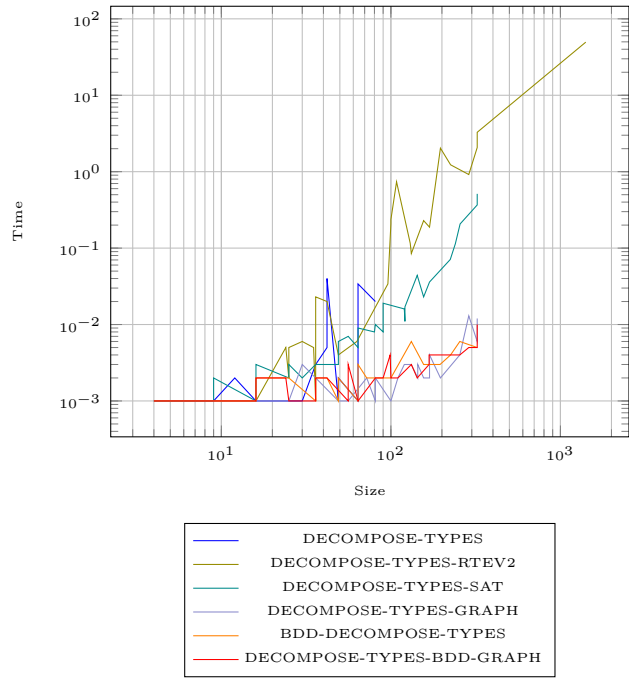


Figure 41: Performance using PCL types

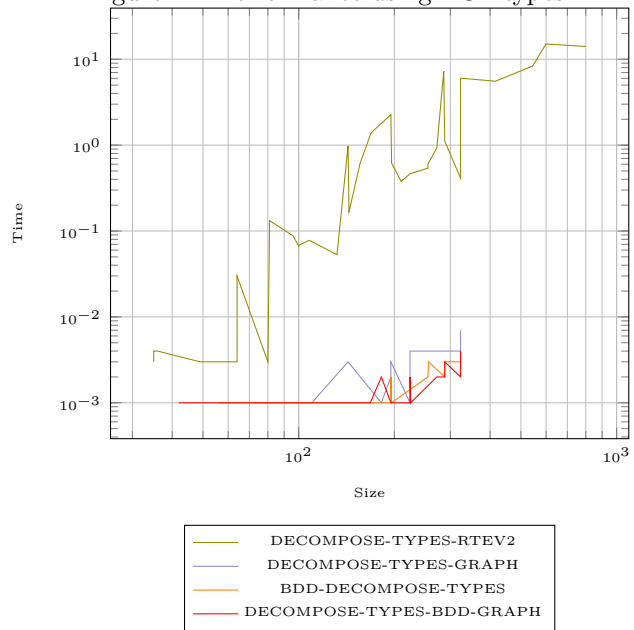


Figure 42: Performance using PCL types (best three)

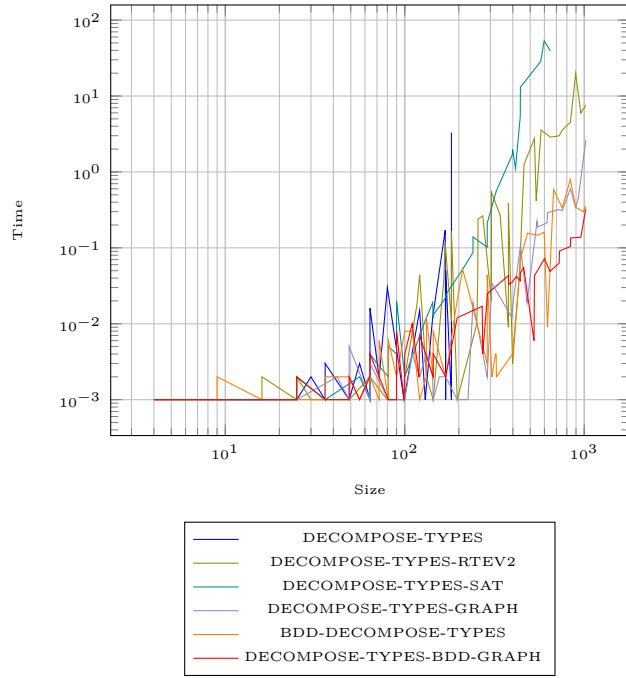


Figure 43: Performance using CL types

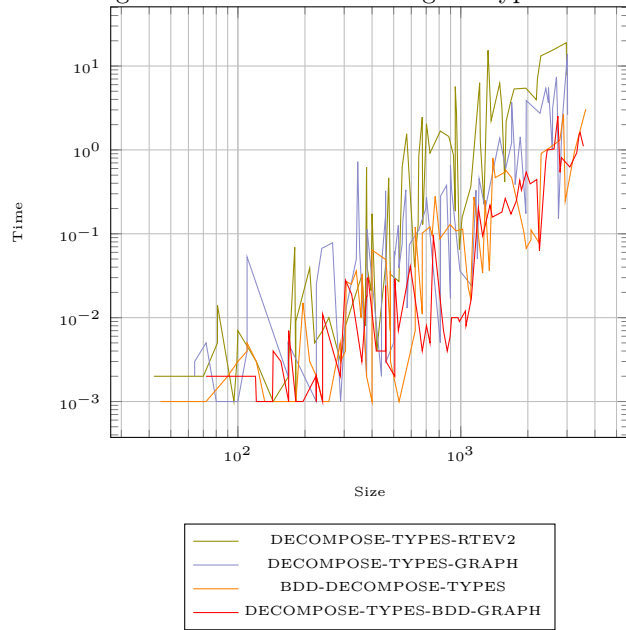


Figure 44: Performance using CL types (best three)

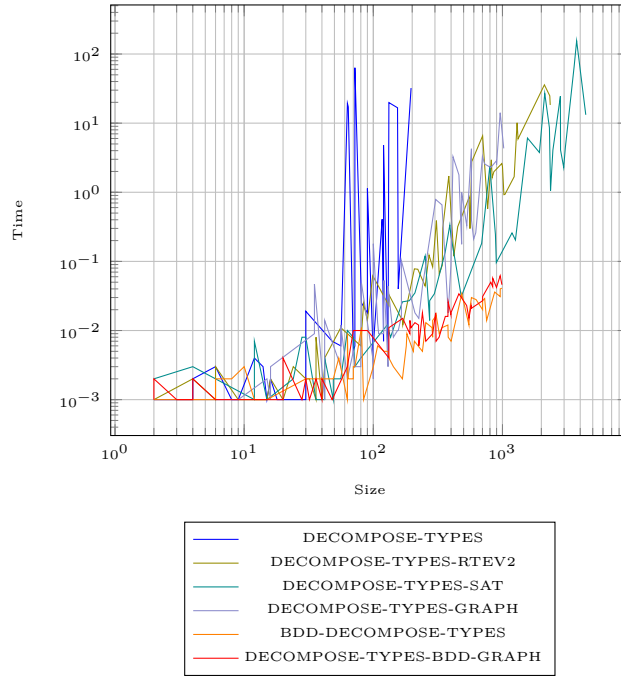


Figure 45: Performance using CL combos

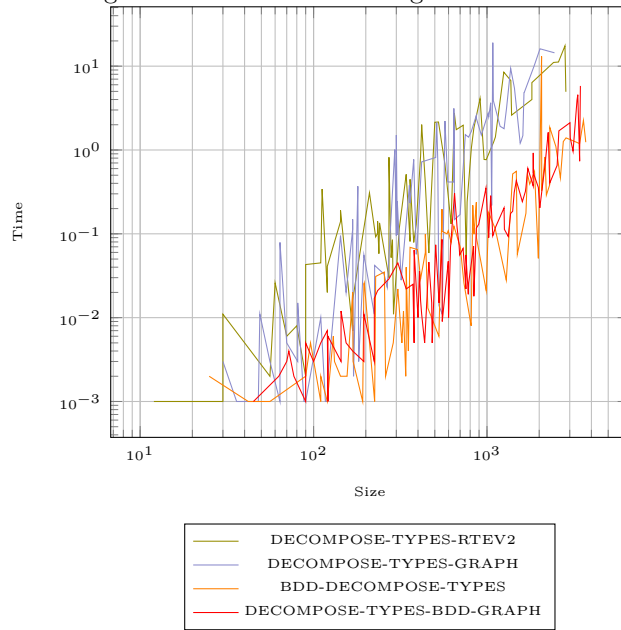


Figure 46: Performance using CL combos (best three)

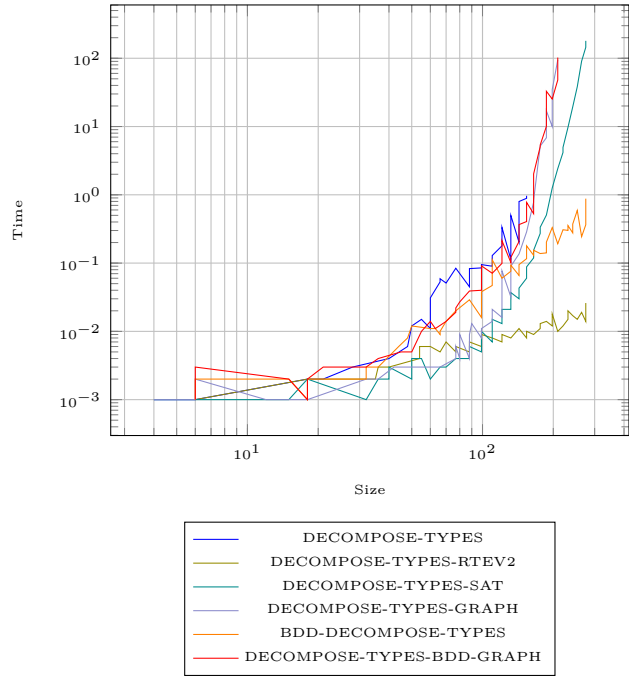


Figure 47: Performance using MEMBER types

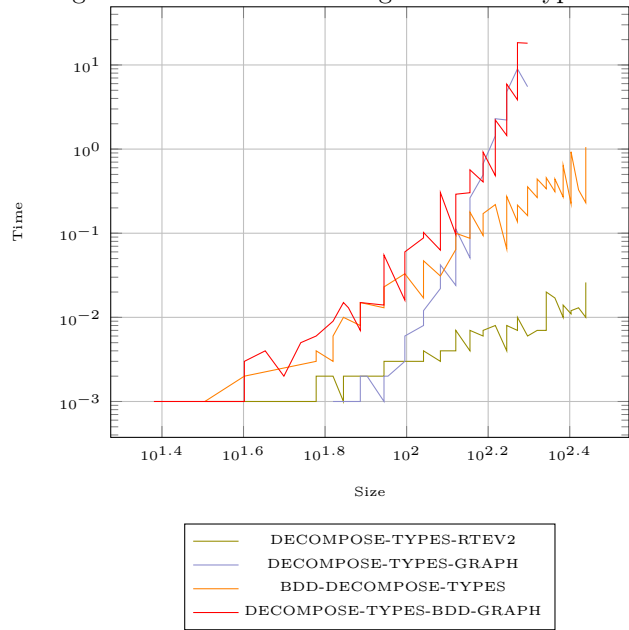


Figure 48: Performance using MEMBER types (best three)

10 Related work

The average and worst case of decision diagrams has been discussed in published works. For example Butler *et al.* [SIHB97] discusses these measurements for symmetric multiple-valued functions. Bryant [Bry86] discusses the effect of variable ordering on diagram sizes. Gröpl [GPS01] examines the worst case size of qOBDDs, which are quasi-reduced BDDs; *i.e.* he omits some of the reduction steps which we use.

As stated in Section 7.1, Castagna [Cas16] discusses the use of BDDs as a tool for type manipulation. The usage therefor is in functional languages significantly different than Common Lisp. A significant motivation for our research was (is) to investigate how these techniques might be useful when applied to a more pragmatic type system such as in Common Lisp.

11 Known issues

Need to discuss where such an algorithm as described in Sections 1 is applicable. Is there an application other than in RTE? One non-conclusive argument is that any reader of Castagna's paper who has a type system with subtyping may need the optimizations discussed here.

Another potential application is that the Common Lisp `subtypep` is known to be slow. Do Common Lisp implementations use s-expressions as type specifiers to perform the type calculus? If so, this could be made more efficient using BDDs. The application is not trivial because despite my current efforts, my BDD calculus does depend on `subtypep` in some non-trivial cases.

12 Future work

There are many potential directions this research could lead next.

12.1 Common Lisp Pattern Matching

Section 7.6.2 discusses approaches for in-lining certain Common Lisp calls such as `typep` using BDD related techniques. It is interesting to ask how much further this can be taken. There are other aspects of functional-language-like pattern matching which could be implemented atop something like Common Lisp `typecase`, and therein compile such constructs to efficient code using techniques which are extensions of those discussed in Section 7.6.2.

12.2 More realistic test cases

Section 9 details work we have done attempting to characterize the performance difference between the various algorithms and various tweaks of algorithms used for type decomposition. The test cases used in the section are explained in the section but are admitted contrived. Having an implementation of pattern matching (as described in Section 12.1) in Common Lisp which takes advantage of type decomposition algorithms, could potentially give us a better set of data for testing performance of the various algorithms.

12.3 Variable Ordering

It is known that the size of an ROBDD depends on the ordering chosen for the variables. Furthermore, it is known that finding the *best* ordering is NP-hard. However, we would like to experiment with attempting to incrementally improve the size by changing the ordering, thus proving an intuition of effort *vs.* improvement. The concept of the experiment would be to extend the techniques used in Section ?? to also measure results of sampling different orderings of variables to measure ROBDD size reduction. This sampling based search could be made as computationally light or intensive as desired. The current guess is that more search time would result in better orderings

for a chosen Boolean expression. However, it is not known whether easy searches may incrementally improve worst case size for a number of variables, or whether only every time and resource consuming searches would be able to yield significant improvements.

12.4 Short-circuiting BDD operations

Section 7.3.3 explains an elegant but sometimes costly operation involving BDDs which is necessary to model the Common Lisp subtype relation sufficiently with BDDs. We believe this (and some other) BDD operations should be optimized. Sometimes we calculate the intersection or relative complement of two given types for the purpose of finding out whether such operation results in the empty type. We believe there are cases when the calculation might be abandoned once it has been determined that the result is not the empty type. *I.e.* it is not necessary to know the exact type, rather that it is different from the empty type.

12.5 Closed forms for average ROBDD size

In Section ?? we derive a formula for the worst case ROBDD size as a function of the number of Boolean variables, and in Section ?? we experimentally develop graphs of the average and median sizes. We would like to derive formulas for the expected ROBDD size, but it currently unclear how to do this.

12.6 0-Sup-BDDs

Minato [Min93] suggests using a different set of reduction rules than those discussed in Section ?. The resulting graph is referred to as a Zero-Suppressed BDD, or 0-Sup-BDDs. Minato claims this data structure offers certain advantages over ROBDDs in modeling sets and expressing set related operations. We see potentially application for this in type calculations, especially when types are viewed as sets (Section 1.2). Additionally, Minato claims that 0-Sup-BDDs provide advantages when the number of input variables is unknown, which is the case we encounter when dealing with Common Lisp types, because we do not have a way of finding all user defined types.

We would like to experiment with 0-Sup-BDD based implementations of our algorithms and contrast the performance results with those found thus far.

12.7 Hidden cost

When using ROBDDs or presumably 0-Sup-BDDs, one must incorporate a hash table of all the BDDs encountered so far (or at least within a particular dynamic extent). The hash table was mentioned in Section 7. The hash table is used to assure structural identity. However, the hash table can become extremely large, even if its lifetime is short. Section ?? discusses the characterization of the worst case size of an ROBDD as a function of the number of Boolean variables. However, this characterization ignores the transient size of the hash table. So one might argue that the size estimations in ?? are misleading in practice.

We would like to continue our experimentation and analysis to provide ways of measuring or estimating the hash table size, and potentially ways of decreasing the burden incurred. For example, we suspect that most of the hash table entries are never re-used. Even though both Andersen [And99] and Minato [Min93] claim the necessity to enforce structural identity, it is not clear whether in our case, the run time cost associated with this memory burden, outweighs the advantage gained by structural identity. Furthermore, the approach used by Castagna [Cas16] seems to favor laziness over caching, lending credence to our suspicion.

12.8 Lazy union BDDs

Castagna [Cas16] mentions the use of a lazy union strategy for representing type expressions as BDDs. However, in this article, we have only implemented the strategy described by Andersen

[And99]. The Andersen approach involves allocating a hash table to memoize all the BDDs encountered in order to both reduce the incremental allocation burden when new Boolean expressions are encountered, and also to allow occasional pointer comparisons rather than structure comparisons. Castagna suggests that the lazy approach can greatly reduce memory allocation. We would like to investigate which of these two approaches gives better performance, or allows us to solve certain problems. Certainly it would be good to attain heuristics to describe situations which one or the other optimization approach is preferable.

Additionally, from the description given by Castagna, the lazy union approach implies that some unions involved in certain BDD related Boolean operations can be delayed until the results are needed, at which time the result can be calculated and stored in the BDD data structure. A question which naturally arises: can we implement a fully functional BDD which never stores calculated values. The memory footprint of such an implementation would *probably* be smaller, while incremental operations would be slower. It is not clear whether the overall performance would be better or worse.

12.9 Improving the SAT based disjoint type algorithm using BDDs

Sections 8.1 and 8.2 describe re-implementing previously presented type decomposition algorithms using the BDD data structure. For completeness, we should also re-implement the SAT based algorithm using the BDD data structure.

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.
- [Bak92] Henry G. Baker. A decision procedure for Common Lisp’s SUBTYPEP predicate. Lisp and Symbolic Computation, 5(3):157–190, 1992.
- [Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, 35:677–691, August 1986.
- [Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv., 24(3):293–318, September 1992.
- [Cas16] Giuseppe Castagna. Covariance and contravariance: a fresh look at an old issue. Technical report, CNRS, 2016.
- [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.
- [GPS01] Clemens Gröpl, Hans Jürgen Prömel, and Anand Srivastav. On the evolution of the worst-case OBDD size. Inf. Process. Lett., 77(1):1–7, 2001.
- [JEH01] Jeffrey D. Ullman Johh E. Hopcroft, Rajeev Motwani. Introduction to Automata Theory, Languages, and Computation. Addison Wesley, 2001.
- [Knu09] Donald E. Knuth. The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional, 12th edition, 2009.
- [Min93] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In Proceedings of the 30th International Design Automation Conference, DAC ’93, pages 272–277, New York, NY, USA, 1993. 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.
- [New16] Jim Newton. Report: Efficient dynamic type checking of heterogeneous sequences. Technical report, EPITA/LRDE, 2016.
- [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.
- [SIHB97] Tsutomu Sasao, Robert J. Barton III, David S. Herscovici, and Jon T. Butler. Average and worst case number of nodes in decision diagrams of symmetric multiple-valued functions. IEEE Transactions on Computers, 46:491–494, 1997.

A Identities of Unary Union and Intersection

In Section 3.1 we introduced definitions for unary union and unary intersection of sets of sets. In this appendix we present proofs of identities which were previously stated without proof in Section 3.1. These proofs are provided as an appendix for completeness and because the proofs themselves are tedious, involving several cases. We feel that the proofs offer little insight into the relations. On the other hand, we do offer in this appendix several examples of the identities intended to provide some intuitive understanding.

As a reminder, we repeat the definitions from Section 3.1.

Definition 11. We define $\bigcup : 2^{2^U} \mapsto 2^U$ by the following rule. If V is a set of subsets of U , then

$$\bigcup V = \begin{cases} \emptyset & \text{if } V = \emptyset & (1) \\ X & \text{if } |V| = 1 \text{ and } V = \{X\} & (2) \\ \bigcup_{X \in V} X & \text{if } |V| > 1 & (3) \end{cases}$$

Definition 12. We define $\bigcap : 2^{2^U} \mapsto 2^U$ by the following rule. If V is a set of subsets of U , then

$$\bigcap V = \begin{cases} U & \text{if } V = \emptyset & (4) \\ X & \text{if } |V| = 1 \text{ and } V = \{X\} & (5) \\ \bigcap_{X \in V} X & \text{if } |V| > 1 & (6) \end{cases}$$

Theorem 6. If $V, V' \subset 2^U$, then

$$\bigcup(V \cup V') = (\bigcup V) \cup (\bigcup V')$$

Proof. We proceed tediously with three cases. Some of the cases themselves have sub-cases.

1. $|V \cup V'| = 0$,
2. $|V \cup V'| = 1$, and
3. $|V \cup V'| > 1$.

Case 1: $|V \cup V'| = 0 \implies V \cup V' = \emptyset \implies V = \emptyset$ and $V' = \emptyset$.

$$\begin{aligned} \bigcup(V \cup V') &= \bigcap \emptyset \\ &= \emptyset && \text{by 1} \\ &= \emptyset \cup \emptyset \\ &= (\bigcup \emptyset) \cup (\bigcup \emptyset) && \text{by 1} \\ &= (\bigcup V) \cup (\bigcup V') \end{aligned}$$

Case 2: $|V \cup V'| = 1$. There are two sub-cases to consider: (2a) $V = V'$ and (2b) $V \neq V'$.

Sub-case 2a: $V = V'$

$$\begin{aligned} \bigcup(V \cup V') &= \bigcup V \\ &= (\bigcup V) \cup (\bigcup V) \\ &= (\bigcup V) \cup (\bigcup V') \end{aligned}$$

Sub-case 2b: $V \neq V'$ implies that either V or V' has cardinality 1 and the other is \emptyset .
Without loss of generality, assume $V' = \emptyset$, so $V \cup V' = V$.

$$\begin{aligned}
\bigcup(V \cup V') &= \bigcup V \\
&= (\bigcup V) \cup \emptyset \\
&= (\bigcup V) \cup (\bigcup \emptyset) && \text{by 1} \\
&= (\bigcup V) \cup (\bigcup V')
\end{aligned}$$

Case 3: $|V \cup V'| > 1$. There are four sub-cases consider: (3a) $V = \emptyset$ or $V' = \emptyset$, (3b) $|V| = 1$ and $|V'| = 1$, (3c) exactly one of $|V| = 1$ or $|V'| = 1$, and (3d) $|V| > 1$ and $|V'| > 1$.

Sub-case 3a: $V = \emptyset$ or $V' = \emptyset$. Without loss of generality, assume $V' = \emptyset$.

$$\begin{aligned}
\bigcup(V \cup V') &= \bigcup V \\
&= (\bigcup V) \cup \emptyset \\
&= (\bigcup V) \cup (\bigcup \emptyset) && \text{by 1} \\
&= (\bigcup V) \cup (\bigcup V')
\end{aligned}$$

Sub-case 3b: $|V| = 1$ and $|V'| = 1$. Denote $V = \{v\}$ and $V' = \{v'\}$.

$$\begin{aligned}
\bigcup(V \cup V') &= \bigcup(\{v\} \cup \{v'\}) \\
&= \bigcup(\{v, v'\}) \\
&= v \cup v' && \text{by 3} \\
&= (\bigcup\{v\}) \cup (\bigcup\{v'\}) && \text{by 2} \\
&= (\bigcup V) \cup (\bigcup V')
\end{aligned}$$

Sub-case 3c: Exactly one of $|V| = 1$ or $|V'| = 1$. Without loss of generality, assume $|V| > 1$, and $|V'| = 1$. Denote $V' = \{v'\}$, so that $\bigcup V' = \bigcup\{v'\}$.

$$\begin{aligned}
\bigcup(V \cup V') &= \bigcup_{\beta \in (V \cup V')} \beta && \text{by 3} \\
&= \bigcup_{\beta \in (V \cup \{v'\})} \beta \\
&= (\bigcup_{\beta \in V} \beta) \cup v' \\
&= (\bigcup V) \cup v' && \text{by 3} \\
&= (\bigcup V) \cup (\bigcup\{v'\}) && \text{by 2} \\
&= (\bigcup V) \cup (\bigcup V')
\end{aligned}$$

Sub-case 3d: $|V| > 1$ and $|V'| > 1$.

$$\begin{aligned}
\bigcup(V \cup V') &= \bigcup_{\beta \in (V \cup V')} \beta && \text{by 3} \\
&= (\bigcup_{\beta \in V} \beta) \cup (\bigcup_{\beta \in V'} \beta) \\
&= (\bigcup V) \cup (\bigcup V') && \text{by 3}
\end{aligned}$$

□

Theorem 6 states that the \cap function preserves unions. *I.e.* a union in 2^{2^U} is mapped to a union in 2^U . One might be tempted to extrapolate to assume that intersections are also preserved. But they are not. *I.e.*, one might be tempted to guess that $\bigcup(V \cap V')$ is identical to $(\bigcup V) \cap (\bigcup V')$. A simple counterexample suffices to disprove this.

Example 24. Let $V = \{\{1, 2\}\}$, and $V' = \{\{1, 3\}\}$. We see that

$$\bigcup(V \cap V') = \bigcup(\{\{1, 2\}\} \cap \{\{1, 3\}\}) = \bigcup \emptyset = \emptyset$$

As well

$$\bigcap(V \cap V') = \bigcap(\{\{1, 2\}\} \cap \{\{1, 3\}\}) = \bigcap \emptyset = \emptyset$$

while

$$(\bigcup V) \cap (\bigcup V') = (\bigcup \{\{1, 2\}\}) \cap (\bigcup \{\{1, 3\}\}) = \{1, 2\} \cap \{1, 3\} = \{1\}$$

So we can conclude that in the general case there exist V and V' for which

$$\bigcup(V \cap V') \neq (\bigcup V) \cap (\bigcup V')$$

However, the equation

$$\bigcup(V \cap V') = (\bigcup V) \cap (\bigcup V')$$

does have solutions. *E.g.*, $V = \emptyset, V' = \emptyset$ satisfies the equation.

Theorem 7. *If $V, V' \subset 2^U$, then*

$$\bigcap(V \cup V') = (\bigcap V) \cap (\bigcap V')$$

Proof. This proof follows the proof of Theorem 6 very closely in form. A critical difference however, can be found in several cases where \emptyset in Theorem 6 has been replaced with U here.

We proceed tediously with three cases. Some of the cases themselves have sub-cases.

1. $|V \cup V'| = 0$,
2. $|V \cup V'| = 1$, and
3. $|V \cup V'| > 1$.

Case 1: $|V \cup V'| = 0 \implies V \cup V' = \emptyset \implies V = \emptyset$ and $V' = \emptyset$.

$$\begin{aligned} \bigcap(V \cup V') &= \bigcap \emptyset \\ &= U && \text{by 4} \\ &= U \cap U \\ &= (\bigcap \emptyset) \cap (\bigcap \emptyset) && \text{by 4} \\ &= (\bigcap V) \cap (\bigcap V') \end{aligned}$$

Case 2: $|V \cup V'| = 1$. There are two sub-cases to consider: (2a) $V = V'$ and (2b) $V \neq V'$.

Sub-case 2a: $V = V'$

$$\begin{aligned} \bigcap(V \cup V') &= \bigcap V \\ &= (\bigcap V) \cap (\bigcap V) \\ &= (\bigcap V) \cap (\bigcap V') \end{aligned}$$

Sub-case 2b: $V \neq V'$ implies that either V or V' has cardinality 1 and the other is \emptyset . Without loss of generality, assume $V' = \emptyset$, so $V \cup V' = V$.

$$\begin{aligned}
\bigcap(V \cup V') &= \bigcap V \\
&= (\bigcap V) \cap U \\
&= (\bigcap V) \cap (\bigcap \emptyset) && \text{by 4} \\
&= (\bigcap V) \cap (\bigcap V')
\end{aligned}$$

Case 3: $|V \cup V'| > 1$. There are four sub-cases consider: (3a) $V = \emptyset$ or $V' = \emptyset$, (3b) $|V| = 1$ and $|V'| = 1$, (3c) exactly one of $|V| = 1$ or $|V'| = 1$, and (3d) $|V| > 1$ and $|V'| > 1$.

Sub-case 3a: $V = \emptyset$ or $V' = \emptyset$. Without loss of generality, assume $V' = \emptyset$.

$$\begin{aligned}
\bigcap(V \cup V') &= \bigcap V \\
&= (\bigcap V) \cap U \\
&= (\bigcap V) \cap (\bigcap \emptyset) && \text{by 4} \\
&= (\bigcap V) \cap (\bigcap V')
\end{aligned}$$

Sub-case 3b: $|V| = 1$ and $|V'| = 1$. Denote $V = \{v\}$ and $V' = \{v'\}$.

$$\begin{aligned}
\bigcap(V \cup V') &= \bigcap(\{v\} \cup \{v'\}) \\
&= \bigcap(\{v, v'\}) \\
&= v \cap v' && \text{by 6} \\
&= (\bigcap\{v\}) \cap (\bigcap\{v'\}) && \text{by 5} \\
&= (\bigcap V) \cap (\bigcap V')
\end{aligned}$$

Sub-case 3c: Exactly one of $|V| = 1$ or $|V'| = 1$. Without loss of generality, assume $|V| > 1$, and $|V'| = 1$. Denote $V' = \{v'\}$, so that $\bigcap V' = \bigcap\{v'\}$.

$$\begin{aligned}
\bigcap(V \cup V') &= \bigcap_{\beta \in (V \cup V')} \beta && \text{by 6} \\
&= \bigcap_{\beta \in (V \cup \{v'\})} \beta \\
&= (\bigcap_{\beta \in V} \beta) \cap v' \\
&= (\bigcap V) \cap v' && \text{by 6} \\
&= (\bigcap V) \cap (\bigcap\{v'\}) && \text{by 5} \\
&= (\bigcap V) \cap (\bigcap V')
\end{aligned}$$

Sub-case 3d: $|V| > 1$ and $|V'| > 1$.

$$\begin{aligned}
\bigcap(V \cup V') &= \bigcap_{\beta \in (V \cup V')} \beta && \text{by 6} \\
&= (\bigcap_{\beta \in V} \beta) \cap (\bigcap_{\beta \in V'} \beta) \\
&= (\bigcap V) \cap (\bigcap V') && \text{by 6}
\end{aligned}$$

□

Even after seeing the proofs, one may still wonder how this works with actual examples. In particular, these identities do in fact hold, especially in the cases: $V' = \emptyset$, $V' = 2^U$, $V' = \{\emptyset\}$, and $V' = \{U\}$.

First, we look at $V' = \emptyset$.

$$\begin{aligned}
 \bigcup V &= \bigcup (V \cup \emptyset) \\
 &= (\bigcup V) \cup (\bigcup \emptyset) \\
 &= (\bigcup V) \cup \emptyset \\
 &= \bigcup V \\
 \\
 \bigcap V &= \bigcap (V \cup \emptyset) \\
 &= (\bigcap V) \cap (\bigcap \emptyset) \\
 &= (\bigcap V) \cap U \\
 &= \bigcap V
 \end{aligned}$$

Next, we look at $V' = 2^U$.

$$\begin{aligned}
 U &= \bigcup 2^U \\
 &= \bigcup (V \cup 2^U) \\
 &= (\bigcup V) \cup (\bigcup 2^U) \\
 &= (\bigcup V) \cup U \\
 &= \bigcup U \\
 \\
 \emptyset &= \bigcap 2^U \\
 &= \bigcap (V \cup 2^U) \\
 &= (\bigcap V) \cap (\bigcap 2^U) \\
 &= (\bigcap V) \cap \emptyset \\
 &= \emptyset
 \end{aligned}$$

Next, we look at $V' = \{U\}$ in the special case where $U \in V$.

$$\begin{aligned}
\bigcup V &= \bigcup(V \cup \{U\}) \\
&= (\bigcup V) \cup \bigcup\{U\} \\
&= (\bigcup V) \cup U \\
&= U \\
\bigcap V &= \bigcap(V \cup \{U\}) \\
&= (\bigcap V) \cap \bigcap\{U\} \\
&= (\bigcap V) \cap U \\
&= \bigcap V
\end{aligned}$$

Next, we look at $V' = \{\emptyset\}$ in the special case where $\emptyset \in V$.

$$\begin{aligned}
\bigcup V &= \bigcup(V \cup \{\emptyset\}) \\
&= (\bigcup V) \cup \bigcup\{\emptyset\} \\
&= (\bigcup V) \cup \emptyset \\
&= \bigcup V \\
\bigcap V &= \bigcap(V \cup \{\emptyset\}) \\
&= (\bigcap V) \cap \bigcap\{\emptyset\} \\
&= (\bigcap V) \cap \emptyset \\
&= \emptyset
\end{aligned}$$

The notation might be considered deceptive. We have already seen that these identities hold:

$$\begin{aligned}
\bigcup(V \cup V') &= (\bigcup V) \cup (\bigcup V') \\
\bigcap(V \cup V') &= (\bigcap V) \cap (\bigcap V')
\end{aligned}$$

By extrapolation from the notation, one might wonder whether there is any relation between

$$\{\bigcup(V \cup V'), \bigcap(V \cup V')\}$$

and

$$\{(\bigcup V) \cap (\bigcup V'), (\bigcap V) \cup (\bigcap V')\}$$

In particular one might wonder whether any of them are identically equal.

One might guess that some of the following identities hold. In fact, they do not as we will show.

$$\bigcup(V \cap V') \stackrel{?}{=} (\bigcup V) \cap (\bigcup V') \tag{7}$$

$$\bigcup(V \cap V') \stackrel{?}{=} (\bigcup V) \cup (\bigcup V') \tag{8}$$

$$\bigcap(V \cap V') \stackrel{?}{=} (\bigcap V) \cup (\bigcap V') \tag{9}$$

$$\bigcap(V \cap V') \stackrel{?}{=} (\bigcap V) \cap (\bigcap V') \tag{10}$$

We have already seen in Example 24 that 7 is not an equality. The same values of V and V' can be used which were used in Example 24.

Example 25. As in Example 24, let $V = \{\{1, 2\}\}$, and $V' = \{\{1, 3\}\}$. We see that the values of the left hand sides of each of 7, 8, 9, and 10 are all \emptyset .

$$\bigcup(V \cap V') = \bigcup(\{\{1, 2\}\} \cap \{\{1, 3\}\}) = \bigcup \emptyset = \emptyset$$

$$\bigcup(V \cap V') = \bigcup(\{\{1, 2\}\} \cap \{\{1, 3\}\}) = \bigcup \emptyset = \emptyset$$

while the right hand sides of 7, 8, 9, and 10 are either $\{1\}$ or $\{1, 2, 3\}$.

$$\begin{aligned} (\bigcup V) \cap (\bigcup V') &= (\bigcup \{\{1, 2\}\}) \cap (\bigcup \{\{1, 3\}\}) \\ &= \{1, 2\} \cap \{1, 3\} \\ &= \{1\} \end{aligned}$$

$$\begin{aligned} (\bigcup V) \cup (\bigcup V') &= (\bigcup \{\{1, 2\}\}) \cup (\bigcup \{\{1, 3\}\}) \\ &= \{1, 2\} \cup \{1, 3\} \\ &= \{1, 2, 3\} \end{aligned}$$

$$\begin{aligned} (\bigcap V) \cup (\bigcap V') &= (\bigcap \{\{1, 2\}\}) \cup (\bigcap \{\{1, 3\}\}) \\ &= \{1, 2\} \cup \{1, 3\} \\ &= \{1, 2, 3\} \end{aligned}$$

$$\begin{aligned} (\bigcap V) \cap (\bigcap V') &= (\bigcap \{\{1, 2\}\}) \cap (\bigcap \{\{1, 3\}\}) \\ &= \{1, 2\} \cap \{1, 3\} \\ &= \{1\} \end{aligned}$$

So we can see that none of the supposed identities hold.

B Finitely many Boolean combinations

The following is an argument that given a finite set V of sets, that the set of all Boolean combinations of these sets is itself finite. To prove this, we first prove a more general result (Theorem 8), and then show the lesser result as an immediate consequence (Corollary 2).

Theorem 8.

Proof. TBD

□

Corollary 2.

Proof. TBD

□

C Code implementing RTE algorithm

The following Common Lisp code implements the algorithm explained in Section 8.1.


```

(defun decompose-types-bdd-graph (type-specifiers)
  (let* ((node-id 0)
        (bdds (sort-unique (mapcar #'bdd type-specifiers)))
        (graph (loop :for bdd :in bdds
                    :collect (list :bdd bdd
                                   :supers nil
                                   :subs nil
                                   :touches nil
                                   :id (incf node-id)))))

    ;; setup all the :supers, :subs, and :touches lists
    (loop :for tail :on graph
          :do (loop :for node2 :in (cdr tail)
                  :with node1 = (car tail)
                  ;; populate the :supers, :subs, and :touches fields
                  :do
                    ... ))

    (let ((changed nil)
          disjoint-bdds)
      (labels ((disjoint! (node)
                (setf changed t)
                (setf graph (remove node graph))
                (push (getf node :bdd) disjoint-bdds))
              (break-sub! (sub super)
                (setf (getf super :bdd) (bdd-and-not (getf super :bdd)
                                                       (getf sub :bdd))
                     changed t)
                (remf sub (getf super :subs))
                (remf super (getf sub :supers)))
              (break-touch! (node-x node-y)
                (setf changed t)
                (remf node-x (getf node-y :touches))
                (remf node-y (getf node-x :touches))
                (let* ((bdd-x (getf node-x :bdd))
                      (bdd-y (getf node-y :bdd))
                      (supers-xy (union (getf node-x :supers)
                                         (getf node-y :supers)))
                      (touches-xy (intersection (getf node-x :touches)
                                                 (getf node-y :touches)))
                      (node-xy (list :bdd (bdd-and bdd-x bdd-y)
                                     :supers supers-xy
                                     :subs nil
                                     :touches (set-difference touches-xy supers-xy)
                                     :id (incf node-id))))
                  (push node-xy graph))))
        (setf changed t)
        (while changed
          (setf changed nil)
          (dolist (node graph)
            (when (disjoint-condition node)
              (disjoint! node)))
          (dolist (node graph) ;; strict subset
            (dolist (super (getf node :supers))
              (when (strict-subset-condition node)
                (break-sub! node super)))
            (dolist (neighbor (getf node :touches)) ;; touching connections
              (when (touching-condition node neighbor)

```



```

                (break-touch! node neighbor))))))
    (mapcar #'bdd-to-dnf disjoint-bdds))))

```

E Code to generate worst case ROBDD of N Boolean variables

The following Common Lisp code was used to construct the ROBDDs shown in Section ?? and is discussed in Section ??.

```

(defun bdd-make-worst-case (vars &key (basename (format nil "/tmp/bdd-worst--D"
                                                         (length vars))))
  (let* ((leaves (list *bdd-true* *bdd-false*))
         (size 2) ;; length of leaves
         (row-num (1- (length vars)))
         (rows (list leaves)))

    ;; build up the bottom
    (while (< (* size (1- size)) (expt 2 row-num))
      (let (bdds)
        (map-pairs (lambda (o1 o2)
                     (push (bdd-node (car vars) o1 o2) bdds))
                   (reduce #'append rows :initial-value ()))
          (push bdds rows)
          (incf size (* size (1- size)))
          (pop vars)
          (decf row-num)))

      ;; build the belt with exactly (expt 2 row-num) elements,
      ;; and 2* (expt 2 row-num) arrows.
      ;; so two cases, does the previous row below have enough elements
      ;; to support the arrows?
      (let* ((n row-num)
             (m (expt 2 n))
             (p (length (car rows)))
             (needed m)
             (remaining (- m (* p (1- p))))
             bdds)

        (block create-links-to-n+1
          ;; First construct as many as possible, but not too many nodes
          ;; pointing to row n+1. Assuming that row n=2 contains p
          ;; number of nodes, this will create a maximum of p(p-1)
          ;; nodes. If p*(p-1) >= 2^n then this is sufficient,
          ;; otherwise, remaining denotes how many additional need to be
          ;; created in BLOCK create-remaining.
          (map-pairs (lambda (left right)
                       (cond
                        ((plusp needed)
                         (push (bdd-node (car vars) left right) bdds)
                         (decf needed))
                        (t
                         (return-from create-links-to-n+1))))
                     (car rows)))

        (block create-remaining
          ;; Next we create any remaining nodes that are needed. This

```

```

;; has some effect only in the case that p*(p-1) < 2^n, which
;; means that the previous block create-links-to-n+1 failed to
;; create 2^n nodes, because row n+1 doesn't have enough
;; elements. So the strategy is to create links to as many of
;; the existing nodes row n+2, n+3 ... as necessary, skipping
;; any pair which has already been created in the previous
;; block.
(map-pairs (lambda (right left &aux (bdd (bdd-node (car vars) left right)))
  (cond
    ;; if there's already a bdd in bdds pointing to
    ;; these two nodes, this skip this pair. we
    ;; don't want duplicate nodes.
    ((member bdd bdds :test #'eq))
    ((plusp remaining)
     (push bdd bdds)
     (decf remaining))
    (t
     (return-from create-remaining))))
  (reduce #'append rows :initial-value ()))

(push bdds rows)
(pop vars))

;; build the top
(while vars
  (let (bdds
        (ptr (car rows)))
    (assert (or (= 1 (length ptr))
                (evenp (length ptr)))) (ptr)
           "expecting either 1 or even number as length, not ~D" (length ptr))
    (while ptr
      (push (bdd-node (car vars) (pop ptr) (pop ptr)) bdds))
    (push bdds rows))
  (pop vars))

;; the top row has one item, that element is the worst
;; case bdd for the given variables
(bdd-view (car (car rows)) :basename basename)
(car (car rows))
(values
 (bdd-to-expr (car (car rows)))
 (bdd-count-nodes (car (car rows)))))

(defun map-pairs (f objs)
  (let* ((size (length objs))
         (size^2 (* size size))
         (vec (make-array size :initial-contents objs))
         (prim size))
    ;; first we make sure that all the objects get 'used if possible
    ;; by calling F with adjacent pairs first f(0 1), f(2 3), f(4 5)
    ;; ...
    (loop for i from 0 below (1- size) by 2
          do (funcall f (svref vec i) (svref vec (1+ i))))
    ;; then we continue by calling the missed adjacent pairs
    ;; f(1 2), f(3 4), f(5 6) ...
    (loop for i from 1 below (1- size) by 2
          do (funcall f (svref vec i) (svref vec (1+ i))))
    ;; then we continue by calling the adjacent pairs in reverse order
    ;; f(1 0), f(2 1), f(3 2), f(4 3) ...

```

```

(loop for i from 0 below (1- size)
      do (funcall f (svref vec (1+ i)) (svref vec i)))
(loop until (= 1 (gcd prim size))
      do (incf prim))
(do ((n 0 (1+ n))
     (b prim (+ b prim)))
    ((= n size^2) nil)
    (multiple-value-bind (q r) (truncate b size)
      (let ((i1 (mod q size)))
        (cond
         ((= i1 r)) ;; don't call the function on the same index at
                    ;; the same time. f(1 1)
         ((= (1+ i1) r)) ;; skip adjacent pairs because they've
                        ;; been handled already above
         ((= i1 (1+ r))) ;; skip adjacent pairs because they've
                        ;; been handled already above
         (t
          (funcall f (svref vec i1) (svref vec r))))))))))

```

F Code for comparing two type specifiers

The following Common Lisp code is explained in Section 7.2.

```

(defun bdd-cmp (t1 t2)
  (cond
   ((equal t1 t2)
    '=)
   ((null t1)
    '<)
   ((null t2)
    '>)
   ((not (eql (class-of t1) (class-of t2)))
    (bdd-cmp (class-name (class-of t1)) (class-name (class-of t2))))
   (t
    (typecase t1
     (list
      (let (value)
        (while (and t1
                    t2)
              (eq '= (setf value (bdd-cmp (car t1) (car t2)))))
          (pop t1)
          (pop t2))
        (cond
         ((and t1 t2)
          value)
         (t1 '>)
         (t2 '<)
         (t  '=))))))
    (symbol
     (cond
      ((not (eql (symbol-package t1) (symbol-package t2)))
       (bdd-cmp (symbol-package t1) (symbol-package t2)))
      ((string< t1 t2) '<)
      (t '>))))
    (package
     (bdd-cmp (package-name t1) (package-name t2)))
    (string
     ;; know they're not equal, thus not string=

```

```

      (cond
        ((string< t1 t2) '<)
        (t '>)))
(number
 (cond ((< t1 t2) '<)
       (t '>)))
(t
 (error "cannot compare a ~A with a ~A"
       (class-of t1) (class-of t2))))))

```

G Code for smarter-subtypep

The following is to `smarter-subtypep` which was mentioned in Section 7.4.1.

```

(defun --smarter-subtypep (t1 t2)
  (declare (optimize (speed 3) (compilation-speed 0)))
  (cond
    ((typep t1 '(cons (member eql member))) ; (eql obj) or (member obj1 ...)
     (list (every #'(lambda (obj)
                     (declare (notinline typep))
                     (typep obj t2))
             (cdr t1))
           t))
    ;; T1 <: T2 <=> not(T2) <: not(T1)
    ((and (typep t1 '(cons (eql not)))
          (typep t2 '(cons (eql not))))
     (multiple-value-list (smarter-subtypep (cadr t2) (cadr t1))))
    ;; T1 <: T2 <=> not( T1 <= not(T2))
    ((and (typep t2 '(cons (eql not)))
          (smarter-subtypep t1 (cadr t2)))
     '(nil t))
    ;; T1 <: T2 <=> not( not(T1) <= T2)
    ((and (typep t1 '(cons (eql not)))
          (smarter-subtypep (cadr t1) t2))
     '(nil t))
    ;; (subtypep '(and cell-error type-error) 'cell-error)
    ((and (typep t1 '(cons (eql and)))
          (exists t3 (cdr t1)
                  (smarter-subtypep t3 t2)))
     '(t t))
    ;; this is the dual of the previous clause, but it appears
    ;; sbcl gets this one right so we comment it out
    ;; ((and (typep t2 '(cons (eql or)))
            (exists t3 (cdr t2)
                      (smarter-subtypep t1 t3)))
     (values t t))
    (t
     '(nil nil))))

(defun -smarter-subtypep (t1 t2 &aux (t12 (list t1 t2)))
  (declare (optimize (speed 3) (compilation-speed 0)))
  (cond
    ((null *subtype-hash*)
     (--smarter-subtypep t1 t2))
    ((nth-value 1 (gethash t12 *subtype-hash*))
     (gethash t12 *subtype-hash*))
    (t
     (setf (gethash t12 *subtype-hash*)

```

```

(--smarter-subtypep t1 t2))))))

(defun smarter-subtypep (t1 t2)
  "The sbcl subtypep function does not know that (eql :x) is a subtype
of keyword, this function SMARTER-SUBTYPEP understands this."
  (declare (optimize (speed 3) (compilation-speed 0)))
  (multiple-value-bind (T1<=T2 OK) (subtypep t1 t2)
    (cond
      (OK
       (values T1<=T2 t))
      (t
       (apply #'values (-smarter-subtypep t1 t2))))))

```

H Code for disjoint-types-p

The following is the code for `disjoint-types-p` as explained in Section 7.4.2.

```

(defun disjoint-types-p (T1 T2 &aux X Y (t12 (list T1 T2)))
  "Two types are considered disjoint, if their intersection is empty,
i.e., is a subtype of nil."
  (declare (notinline subsetp))
  (flet ((calculate ()
          (multiple-value-bind (disjointp OK) (subtypep (cons 'and t12) nil)
            (cond
              (OK
               (list disjointp t))
              ((and (symbolp T1)
                    (symbolp T2)
                    (find-class T1 nil)
                    (find-class T2 nil))
               (list (not (dispatch:specializer-intersections (find-class T1)
                                                                (find-class T2)))
                     t))
              ((subsetp '((t t) (nil t))
                        (list (setf X (multiple-value-list (smarter-subtypep T1 T2)))
                              (multiple-value-list (smarter-subtypep T2 T1)))
                         :test #'equal)
               ;; Is either T1<:T2 and not T2<:T1
               ;; or T2<:T1 and not T1<:T2 ?
               ;; if so, then one is a proper subtype of the other.
               ;; thus they are not disjoint.
               (list nil t))
              ((and (typep T1 '(cons (eql not)))
                    (typep T2 '(cons (eql not)))
                    (smarter-subtypep t (list 'and (cadr T1) (cadr T2)))
                    (disjoint-types-p (cadr T1) (cadr T2)))
               (list nil t))
              ;; T1 ^ T2 = 0 ==> !T1 ^ T2 != 0 if T1!=1 and T2 !=0
              ;; !T1 ^ T2 = 0 ==> T1 ^ T2 != 0 if T1!=0 and T2 !=1
              ((and (typep T1 '(cons (eql not)))
                    (not (void-type-p (cadr T1)))
                    (not (void-type-p T2))
                    (disjoint-types-p (cadr T1) T2))
               (list nil t))
              ;; T1 ^ T2 = 0 ==> T1 ^ !T2 != 0 if T1!=0 and T2!=1
              ;; T1 ^ !T2 = 0 ==> T1 ^ T2 != 0 if T1!=0 and T2!=0
              ((and (typep T2 '(cons (eql not)))
                    (not (void-type-p T1))
                    (not (void-type-p (cadr T2)))
                    (disjoint-types-p T1 (cadr T2)))
               (list nil t))
              ;; e.g., (disjoint-types-p (not float) number) ==> (nil t)
            ))))

```

```

;;      (disjoint-types-p (not number) float) ==> (t t)
((and (typep T1 '(cons (eql not)))
      (setf Y (multiple-value-list (smarter-subtypep (cadr T1) T2)))
      (setf X (multiple-value-list (smarter-subtypep T2 (cadr T1))))
      (subsetp '((t t) (nil t)) (list X Y) :test #'eql))
 (list (car X) t))
;; e.g., (disjoint-types-p float (not number)) ==> (t t)
;;      (disjoint-types-p number (not float)) ==> (nil t)
((and (typep T2 '(cons (eql not)))
      (setf Y (multiple-value-list (smarter-subtypep T1 (cadr T2))))
      (setf X (multiple-value-list (smarter-subtypep (cadr T2) T1)))
      (subsetp '((t t) (nil t)) (list X Y) :test #'eql))
 (list (car Y) t))
((or (smarter-subtypep T1 T2)
     (smarter-subtypep T2 T1))
 (list nil t))
(t
 (list nil nil))))))
(apply #'values
 (cond ((null *disjoint-hash*)
        (calculate))
       ((nth-value 1 (gethash t12 *disjoint-hash*))
        (gethash t12 *disjoint-hash*))
       (t
        (progn (setf (gethash t12 *disjoint-hash*)
                     (calculate)))))))

```

I Code for bdd-to-dnf

The following is the code for `bdd-to-dnf` which is explained in Section 7.5.

```

(defun bdd-to-dnf (bdd)
  (slot-value bdd 'dnf))

(defmethod slot-unbound (class (bdd bdd-node) (slot-name (eql 'dnf)))
  (setf (slot-value bdd 'dnf)
        (-bdd-to-dnf bdd)))

(defun -bdd-to-dnf (bdd)
  (declare (type bdd bdd))
  (labels ((wrap (op zero forms)
            (cond ((cdr forms)
                   (cons op forms))
                  (forms
                   (car forms))
                  (t
                   zero))))
    (prepend (head dnf)
             (typecase dnf
               ((cons (eql or))
                (wrap
                 'or nil
                 (mapcar (lambda (tail)
                           (prepend head tail))
                         (cdr dnf))))
               ((cons (eql and))
                (wrap 'and t (remove-super-types (cons head (cdr dnf))))))
             ((eql t)
              head)
             ((eql nil)
              nil)))

```

```

        nil)
      (t
        (cons 'and (remove-supers (list head dnf))))))
    (disjunction (left right)
      (cond
        ((null left)
         right)
        ((null right)
         left)
        ((and (typep left '(cons (eql or)))
              (typep right '(cons (eql or))))
         (cons 'or (nconc (copy-list (cdr left)) (cdr right))))
        ((typep left '(cons (eql or)))
         (wrap 'or nil (cons right (cdr left))))
        ((typep right '(cons (eql or)))
         (wrap 'or nil (cons left (cdr right))))
        (t
         (wrap 'or nil (list left right))))))

    (let ((left-terms (prepend (bdd-label bdd) (bdd-to-dnf (bdd-left bdd))))
          (right-terms (prepend '(not ,(bdd-label bdd)) (bdd-to-dnf (bdd-right bdd)))))
      (disjunction left-terms
                   right-terms)))

```

J Code for bdd-to-expr

The following is the code for `bdd-to-expr` which is explained in Section 7.5.

```

(defun bdd-to-expr (bdd)
  (slot-value bdd 'expr))

(defmethod slot-unbound (class (bdd bdd-node) (slot-name (eql 'expr)))
  (setf (slot-value bdd 'expr)
        (cond
          ((and (eq *bdd-false* (bdd-left bdd))
                (eq *bdd-true* (bdd-right bdd)))
           '(not ,(bdd-label bdd)))
          ((and (eq *bdd-false* (bdd-right bdd))
                (eq *bdd-true* (bdd-left bdd)))
           (bdd-label bdd))
          ((eq *bdd-false* (bdd-left bdd))
           '(and (not ,(bdd-label bdd)) ,(bdd-to-expr (bdd-right bdd))))
          ((eq *bdd-false* (bdd-right bdd))
           '(and ,(bdd-label bdd) ,(bdd-to-expr (bdd-left bdd))))
          ((eq *bdd-true* (bdd-left bdd))
           '(or ,(bdd-label bdd)
                (and (not ,(bdd-label bdd)) ,(bdd-to-expr (bdd-right bdd)))))
          ((eq *bdd-true* (bdd-right bdd))
           '(or (and ,(bdd-label bdd) ,(bdd-to-expr (bdd-left bdd)))
                (not ,(bdd-label bdd))))
          (t
           '(or (and ,(bdd-label bdd) ,(bdd-to-expr (bdd-left bdd)))
                (and (not ,(bdd-label bdd)) ,(bdd-to-expr (bdd-right bdd)))))))

```



```

      (and (not Z4)
           (or (and Z5 (and Z6 Z7))
               (and (not Z5)
                    (or (and Z6 (not Z7)) (not Z6))))))
    (and (not Z2)
         (or (and Z3
              (or (and Z4
                   (or (and Z5 (or (and Z6 (not Z7)) (not Z6)))
                       (and (not Z5) (and (not Z6) Z7))))
                (and (not Z4)
                     (or (and Z5 (and (not Z6) (not Z7)))
                         (and (not Z5) (and Z6 Z7))))))
           (and (not Z3)
                (or (and Z4
                     (or (and Z5 (not Z6))
                         (and (not Z5) (or Z6 (and (not Z6) Z7))))
                 (and (not Z4)
                      (or (and Z5 (or Z6 (and (not Z6) (not Z7))))
                          (and (not Z5) (and Z6 (not Z7))))))))))
    (and (not Z1)
         (or (and Z2
              (or (and Z3
                   (or (and Z4
                        (or (and Z5 (and (not Z6) Z7))
                            (and (not Z5) (or (and Z6 (not Z7)) (not Z6))))
                    (and (not Z4)
                         (or (and Z5 (or (and Z6 (not Z7)) (not Z6)))
                             (and (not Z5) (and Z6 Z7))))))
                (and (not Z3)
                     (or (and Z4
                          (or (and Z5 (and Z6 (not Z7)))
                              (and (not Z5) (or Z6 (and (not Z6) (not Z7))))))
                    (and (not Z4)
                         (or (and Z5 (or Z6 (and (not Z6) (not Z7))))
                             (and (not Z5) (and (not Z6) Z7))))))))))
           (and (not Z2)
                (or (and Z3
                     (or (and Z4
                          (or (and Z5 (or Z6 (and (not Z6) Z7)))
                              (and (not Z5)
                                   (or (and Z6 Z7) (and (not Z6) (not Z7))))))
                    (and (not Z4)
                         (or (and Z5 (and Z6 (not Z7)))
                             (and (not Z5) (not Z6))))))
                (and (not Z3)
                     (or (and Z4
                          (or (and Z5 (and Z6 Z7))
                              (and (not Z5) (and (not Z6) (not Z7))))))
                    (and (not Z4)
                         (or (and Z5 (or (and Z6 Z7) (not Z6)))
                             (and (not Z5) Z6))))))))))

```