An Elegant and Fast Algorithm for Partitioning Types

Jim E. Newton
jnewton@lrde.epita.fr
EPITA/LRE
Le Kremlin-Bicêtre, France

ABSTRACT

We present an improvement on the Maximal Disjoint Type Decomposition algorithm, published previously. The new algorithm is shorter than the previously best known algorithm in terms of lines of code, and performs better in many, but not all, benchmarks. Additionally the algorithm computes metadata which makes the Brzozowski derivative easier to compute–both easier in terms of accuracy and computation time. Another advantage of this new algorithm is its resilience limited `subtypep` implementations.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; *Type theory*.

ACM Reference Format:
Jim E. Newton. 2024. An Elegant and Fast Algorithm for Partitioning Types. In *Proceedings of the 16th European Lisp Symposium (ELS'23)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.5281/zenodo.7813576

1 INTRODUCTION

This paper discusses recent developments in a procedure introduced in a previous European Lisp Symposium paper from 2016, where Newton *et al.* [18] introduced *regular type expressions* (RTEs), and suggested a technique to compute membership of the corresponding regular languages. The paper mentioned several limitations which needed further study.

Newton *et al.* [18] developed the theory further applied only to Common Lisp [3], and generalized to other programming languages [17]: Clojure [10, 11], Scala [20, 21], and Python [24].

In Section 2.1, we briefly summarized the theory in order to set the stage for the contributions of this article.

1.1 Contributions

We introduce a new procedure for computing the MDTD (maximal disjoint type decomposition), Definition 2.1. Our new procedure has several advantages over previously known techniques.

(1) It is elegant, Section 3.2.
(2) It is provably correct, Section 3.4.
(3) It eases computation of Brzozowski derivative, Section 4.
(4) It is fast, Section 5.

ELS'23, April 24–25 2023, Amsterdam

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.7813576

1.2 Overview

Given an RTE (Section 2.1), we construct a deterministic finite automaton (DFA) whose language of acceptance (set of all accepted sequences) is the same as the accepting language of the RTE. A DFA consists of states and transitions. The transitions are labeled with Common Lisp type specifiers; see Figures 1 and 2. The construction process simply needs to be able to (1) construct a state with its transitions, and must (2) be repeated until all states have been created. There are two questions to be answered:

- Given a state, what are its transitions?
- What are all the states?

To determine the states and transitions, the Brzozowski derivative of an RTE, r with respect to type v, (Section 2.3) is employed. We compute a value denoted, $\partial_v r$, once for each transition, where r represents an RTE, and v represents a type. The recursive procedure to compute $\partial_v r$ (Section 4) relies heavily on knowledge of disjoint and subtype relations between types (represented in Common Lisp as so-called *type specifiers*). Each time we compute $\partial_v r$ for given values of r and v, we produce yet another RTE, and add it to the working list of RTEs for which we must again compute $\partial_v r$ (for other values of v). Every time we encounter an RTE which we have not seen before, we create a new state, and associate the RTE with that state. We label the transitions with the type used in computing $\partial_v r$. *E.g.*, if states 1 and 2 correspond to RTEs r_1 and r_2 respectively, and $r_2 = \partial_v r_1$ for some value of v, then we construct a transition from state 1 to state 2 labeled with the type v. The process eventually terminates.

The above flow description is not new. What is new and is the novel contribution we present in this article is how to tackle two computational challenges:

- For a given RTE, r, what is the set of types, v, for which we must compute $\partial_v r$? The MDTD Algorithm in Section 3, efficiently computes this set of types.
- Computing $\partial_v r$ relies on knowledge of disjoint and subtype relations between types. Often, programs rely on `subtypep` to decide such relations, but calls to `subtypep` can be compute intensive and may return inconclusive results, which we refer to an *inaccurate*.[1] Our proposed MDTD procedure circumvents reliance on inaccurate results of `subtypep` for these problematic cases.

2 RTE TO DFA FLOW

2.1 Regular Type Expressions

Traditional regular expressions are a DSL (domain specific language) for specifying sets of strings according to which sequences of characters appear in the string. The DSL provides mechanisms

[1] The Common Lisp specification refers to the second return value of `subtypep` as an *accurate* [indicator].

for specifying optionality, alternation, and repetition. Hazel [8, 25] has provided a regular expression library for Common Lisp, but such libraries are standard in most modern programming languages. We assume the reader understands traditional regular expressions. For a theoretical treatment, see Hopcroft [12].

The regular type expression (RTE) is an expression akin to the tradition regular expression, but which the DSL is written in terms of types (rather than characters), and algebraic combinations of these types to specify optionality, repetition, and alternation. Examples of RTEs are:

$$r_1 = \left(symbol \cdot \left(number^+ \cup string^+\right)\right)^+ \quad (1)$$

$$r_2 = (integer \cdot number) \cup (number \cdot integer) \quad (2)$$

Expression (1) means the set of all non-empty sequences of one or more occurrences of a symbol followed by either a number or a string. Sequences such as `(x 3 a "hello")` and `(y 3.1 b "hello" c "world")` are accepted, while sequences such as `(x a "hello")` and `(x 3 a "hello" c)` are rejected. Expression (2) represents the set of sequences of length two, which consist of two numbers, at least one of which is an integer.

The idea of RTE is reminiscent of Clojure Spec [14] and `Malli` [9]. Although Hickey [11] mentions Spec's existence, but we have found no other peer reviewed articles on either. Thus far, conversations between experts on public forums have lead us to contradictory conclusions that Spec is not based on finite automata theory at all, and other claims that it is based on NFA (non-deterministic finite automata) work by Might *et al.* [2, 13]. An NFA-based procedure (presumably using backtracking) would have at least polynomial complexity—our approach offers linear complexity. Grande [1], released a regular pattern matching library in Clojure called `seqexp`. According to an interview with Grande, the `seqexp` does not use a finite automata approach because of JVM limitations.

2.2 DSL for Regular Type Expressions

Expressions (1) and (2) (from Section 2.1) are represented as RTEs respectively as follows

The RTE, $r_1 = (symbol \cdot (number^+ \cup string^+))^+$, is represented in Common Lisp as:

```
(:cat symbol (:+ (:or (:+ number) (:+ string))))
```

The RTE, $r_2 = (integer \cdot number) \cup (number \cdot integer)$, is represented in Common Lisp as:

```
(:or (:cat integer number) (:cat number integer))
```

Keyword symbols such as, `:*`, `:+`, `:and`, `:or`, `:not`, represent the traditional regular expression operators, while leaf level objects represent Common Lisp type specifiers: `number`, `string`, `integer`.

2.3 Constructing a DFA from an RTE

The Common Lisp library, `rte`, is available on `quicklisp`, or directly from GitLab at https://gitlab.lrde.epita.fr/jnewton/regular-type-expression. Analogous to the classical case, an RTE can also be represented by a finite automaton. Whereas in the classical case, transitions are labeled by so-called letters from a fixed, finite alphabet; in our case, we label transitions with type specifiers. Each type specifier denotes the possibly infinite set of possible Common Lisp objects. Newton *et al.* [18] outlined a procedure for converting an RTE to a finite automaton using the Brzozowski derivative [5], in particular Newton follows closely the procedure outlined by Owens *et al.* [22]. The Brzozowski derivative, denoted $\partial_v r$, (read: derivative of r with respect to v) is a function which accepts an RTE, r, and a type specifier, v, and returns an RTE.

As explained in [18] and restated in Algorithm 1, a finite automaton can be constructed by computing the derivative of the given RTE, with respect to each element of a set, $\mathcal{A}$, of types, producing a new set of RTEs. Each of these RTEs represents an additional state in the finite automaton, and the transitions to the states are labeled by the (with-respect-to) type in the derivative computation. The procedure is repeated on the new RTEs, producing more states and transitions, including transitions to pre-existing states, *i.e.* loops to states we have seen already. Brzozowski [5] argues that this process terminates.

Algorithm 1: Construct DFA by Brzozowski derivative

Input: r : an RTE
Output: σDFA

```
1.1  begin
1.2      q_0 ← new State(r), T ← (), Q ← {q_0}, W ← {q_0}
1.3      while W ≠ ∅ do
1.4          q_1 ← any element from W
1.5          r ← q_1.rte
1.6          W ← W \ {q_1}
1.7          for v ∈ MDTD(1st(r)) do
1.8              d ← ∂_v r                          // canonicalize
1.9              if d = ∅ then
                     // avoid unsatisfiable transition
1.10                 continue
1.11             else if ∃ q_2 ∈ Q such that q_2.rte = d then
                     // transition to pre-existing state
1.12                 T ← (q_1, v, q_2) :: T
1.13             else
                                 // transition to a new state
1.14                 q_2 ← new State(d)
1.15                 T ← (q_1, v, q_2) :: T
1.16                 W ← q_2 :: W
1.17                 Q ← q_2 :: Q
             // compute final states, cf Owens [22]
                      // ⟦·⟧ explained in Section 4.
1.18     F ← {q ∈ Q | () ∈ ⟦q.rte⟧}
1.19     return (Q, q_0, F, T)
```

RTE to DFA construction is a generalization of classical DFA construction. Our particular DFA is a special case of that D'Antoni and Veanes [6] describe called *symbolic finite automata*. Algorithm 1 outlines the DFA construction using the Brzozowski derivative, and Figure 1 illustrates such a constructed DFA given an RTE. There are several parts of Algorithm 1 which deserve further explanation, making the topic interesting to research.

(1) The $\mathcal{A}$ needed for Algorithm 2 is computed as a call to $1^{st}(r)$ on line 1.7, discussed below.
(2) Canonicalization of an RTE on line 1.8, discussed below.

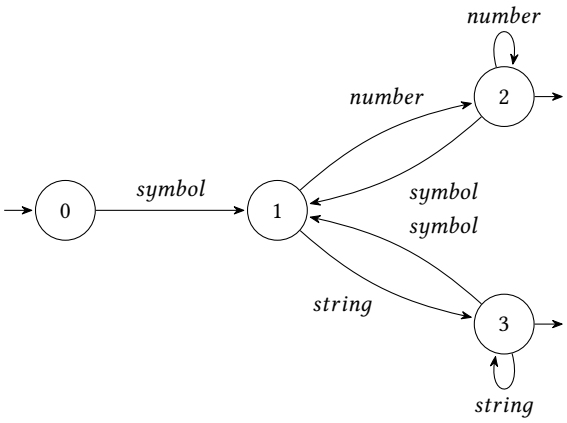


Figure 1: Deterministic finite automaton representing the expression: $(symbol \cdot (number^{+} \cup string^{+}))^{+}$

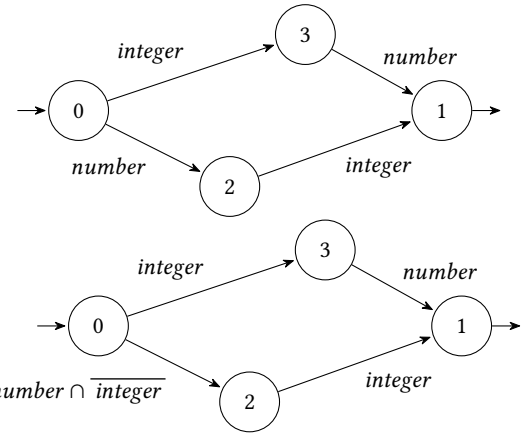


Figure 2: Finite automata representing: $(integer \cdot number) \cup (number \cdot integer)$**, (Top: non-deterministic, Bottom: deterministic)**

(3) Computation of the MDTD, a central contribution of this paper, and explained in Section 3.
(4) Computation of $\partial_v r$ on line 1.7, explained in Section 4.

On line 1.7, we call the MDTD function. The argument (as we'll in Section 3) is a set of types. We could pass the set of all type specifier mentioned in the RTE: {*symbol*, *string*} for Expression (1), and {*integer*, *number*} for Expression (2). This would give a correct answer, but most of the derivatives computed would be $\emptyset$, so most computation time would be wasted. Instead, an important optimization explained in [16] is to only consider *relevant* types. For example, Figure 1 shows that *number* is not relevant to the transitions at state 0. On line 1.7, $1^{st}(r)$ references a procedure for deciding a priori, which types are relevant. We do not discuss this optimization more in this paper as it has no effect on the MDTD algorithm.

Some amount of canonicalization is necessary (1.8). As mentioned above, Brzozowski [5] argues that this process eventually terminates provided a reasonable amount of *canonicalization* is performed on the computed expressions.

2.4 Determinism

Figure 2 illustrates the distinction between deterministic and non-deterministic finite automata. In order that the automaton be deterministic, we must be assure that the set of transitions leaving any given state contain no overlapping types. For example, state 0 in Figure 2 (Top) has exiting transitions *number* and *integer*, which are not disjoint types—a situation which we must avoid. On the other hand, state 0 in Figure 2 (Bottom) has transitions *integer* and $number \cap \overline{integer}$, which are indeed disjoint types.

Definition 2.1 (MDTD). Suppose we are given a finite set of type specifiers, $\mathcal{A} = \{A_1, A_2, \ldots A_n\}$ The set $\mathcal{X} = \{X_1, X_2, \ldots, X_m\}$ is called the *maximal disjoint type decomposition* of $\mathcal{A}$, if the following hold.

(1) **Union invariance**:
$X_1 \cup X_2 \cup \ldots \cup X_n = A_1 \cup A_2 \cup \ldots \cup A_n$.
(2) **Disjointness**:
If $X_i, X_j \in \mathcal{X}$, with $i \neq j$, then $X_i \cap X_j = \emptyset$.
(3) **Refinement**:
If $v \in \mathcal{X}$ and $\mu \in \mathcal{A}$, then either $v \subseteq \mu$ or $v \cap \mu = \emptyset$.

In Figure 2 (bottom) we express the MDTD (maximal disjoint type decomposition) of $\{\top, integer, number\}$ as $\{integer, number \cap \overline{integer}, \top \cap \overline{number}\}$. As is common convention, the graph omits the transition labeled $\top \cap \overline{number}$, because it leads to the so-called *sink* state, indicating a state of rejection as opposed to acceptance.

This kind of partition, illustrated in Figure 3, ensures that any object encountered in a candidate sequence is a member either of exactly one type in $\mathcal{X}$ or is a member of no type in the decomposition. Figure 3 expresses the MDTD as $\{X_1, X_2, \ldots, X_9\}$. Notice there is one X_i for each bounded disjoint area in Figure 3.

If $\top$, the universal type, is included in the input, $\mathcal{A}$, then the output $\mathcal{X}$ will also include the region outside A_1, *i.e.*, $\overline{A_1} \in \mathcal{X}$.

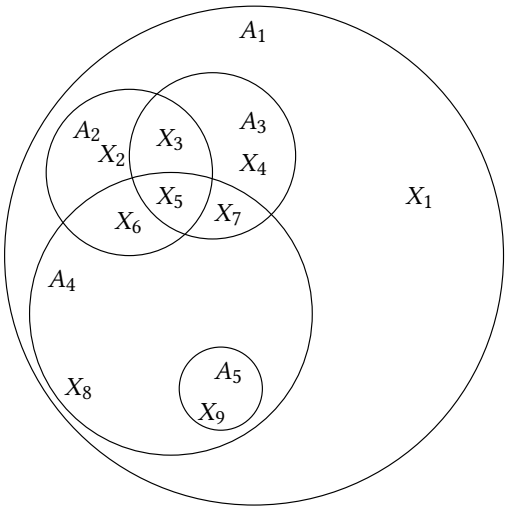
We have proven in [16] that a MDTD exists and is uniquely determined for any finite set of types.

3 A NEW MDTD PROCEDURE

We present the procedure shown in Algorithm 3 to compute the MDTD. The procedure returns two values, $\mathcal{X}$ and $\mathcal{S}$, where $\mathcal{X}$ is the actual set of types comprising the type decomposition, and $\mathcal{S}$ is metadata which can be reused to make the Brzozowski derivative (Section 4), more efficient and more accurate. The actual metadata is a mapping from each type specifier, v in $\mathcal{X}$, to two sets: a set of factors (super-types) of v and a set of disjoint types of v.

3.1 The subtypep function, in Common Lisp

In Common Lisp, programmatic reasoning about types is done in term of two so-called *type specifiers*. The type specifiers *t* and *nil* refer respectively to the universal type (containing all possible Common Lisp object) and the empty type (containing no objects). The intersection and union of two types can be specified using the *and* and *or* types; *e.g.*, `(or string (and integer (satisfies`



Disjoint Set	Derived Expression	Factors	Disjoint Types
X_1	$A_1\overline{A_2}\,\overline{A_3}\,\overline{A_4}$	A_1	A_2, A_3, A_4, A_5
X_2	$A_2\overline{A_3}\,\overline{A_4}$	A_1, A_2	A_3, A_4, A_5
X_3	$A_2A_3\overline{A_4}$	A_1, A_2, A_3	A_4, A_5
X_4	$A_3\overline{A_2}\,\overline{A_4}$	A_1, A_3	A_2, A_4, A_5
X_5	$A_2A_3A_4$	A_1, A_2, A_3, A_4	A_5
X_6	$A_2A_4\overline{A_3}$	A_1, A_2, A_4	A_3, A_5
X_7	$A_3A_4\overline{A_2}$	A_1, A_3, A_4	A_2, A_5
X_8	$A_4\overline{A_2}\,\overline{A_3}\,\overline{A_5}$	A_1, A_4	A_2, A_3, A_5
X_9	A_5	A_1, A_4, A_5	A_2, A_3

Figure 3: Example of Maximal Disjoint Type Decomposition: $\mathcal{X} = \{X_1, X_2, \ldots, X_9\}$ is the MDTD of $\mathcal{A} = \{A_1, A_2, \ldots, A_5\}$. Derived expressions are intersections of types from $\mathcal{A}$ or complements thereof.

evenp))). And types can be complemented (inverted) using the *not* type; *e.g.* (not integer).

In Common Lisp, the subtypep function can be used to determine the subtype relation. The behavior of subtypep can be understand by the following three cases:

(1) The Common Lisp expresison, (subtypep integer number), returns two values t,t; the first t indicates that the subtype relation is validated (*integer* $\subseteq$ *number*) while the second t indicates that the subtype relation was proven to be true.
(2) (subtypep number integer) returns two values nil,t; the nil indicates that the subtype does not hold (*integer* $\not\subseteq$ *number*) while the t indicates that the subtype relation was proven to be false.
(3) If Common Lisp cannot determine whether the subtype relation holds, subtypep returns nil,nil. (subtypep (satisfies oddp) integer) returns two values nil,nil; the first nil has no meaning because the second nil indicates that the subtype relation was neither proven nor disproven.

Algorithm 2: Compute MDTD of given $\mathcal{A}$.

Input: $\mathcal{A}$: a set of type designators
Output: $(\mathcal{X}, \mathcal{S})$: partition and metadata
begin
  $\mathcal{S} \leftarrow \{(\top, \{\top\}, \{\bot\})\}$ // working list of triples
  $\mathcal{X} \leftarrow \{\top\}$ // working list of disjoint types
  for $\mu \in \mathcal{A}$ **do**
    for $(\upsilon, f, d) \in \mathcal{S}$ **do**
      $\mathcal{S} \leftarrow \mathcal{S} \setminus \{(\upsilon, f, d)\}$
      if $\mu \cap \upsilon = \emptyset$ **then**
        $\mathcal{S} \leftarrow (\upsilon, f, \mu :: d) :: \mathcal{S}$ // υ, μ disjoint
      else if $\upsilon \subseteq \mu$ **then**
        $\mathcal{S} \leftarrow (\upsilon, \mu :: f, d) :: \mathcal{S}$ // $\upsilon \cap \overline{\mu} = \emptyset$
      else
        // $\mu \cap \upsilon$ and $\overline{\mu} \cap \upsilon$ partition υ
        $\upsilon_1 \leftarrow \mu \cap \upsilon$
        $\upsilon_2 \leftarrow \overline{\mu} \cap \upsilon$
        $\mathcal{X} \leftarrow (\mathcal{X} \setminus \upsilon) \cup \{\upsilon_1, \upsilon_2\}$
        $\mathcal{S} \leftarrow (\upsilon_1,\ \mu :: f,\ d) :: \mathcal{S}$ // $\upsilon_1 \subseteq \mu$
        $\mathcal{S} \leftarrow (\upsilon_2,\ f,\ \mu :: d) :: \mathcal{S}$ // υ_2, μ disjoint
  return $(\mathcal{X}, \mathcal{S})$

Algorithm 3: expand-1: Helper function for MDTD. A *triple* consists of a derived expression, list of factors, and list of disjoint types as in Figure 3 (Bottom).

Input: μ : a type designator
Input: (υ, f, d): a triple
Output: a set of one or two triples
begin
  if $\mu \cap \upsilon = \emptyset$ **then**
    return $\{(\upsilon, f, \mu :: d)\}$
  else if $\upsilon \subseteq \mu$ **then**
    return $\{(\upsilon, \mu :: f, d)\}$
  else
    return $\left\{\left(\mu \cap \upsilon, \mu :: f, d\right), \left(\overline{\mu} \cap \upsilon, f, \mu :: d\right)\right\}$

An expression such as (subtypep integer nil) as whether *integer* $\subseteq \emptyset$, *i.e.*, whether the integer type is empty. To ask about the disjoint relation, we ask whether the intersection is empty: (subtypep (and integer string) nil) asks whether (*integer* $\cap$ *string*) $\subseteq \emptyset$.

The Common Lisp specification allows subtypep to return nil,nil under several circumstances, most notably in cases involving the satisfies type in which case it is often impossible to determine, but also when it deems an accurate determination to be too costly in terms of computation time.

3.2 MDTD in Common Lisp

Algorithm 2 is restated more succinctly reduce and mapcan as in Figure 4. We pass the local function, expand, to reduce. The expand function uses mapcan to iterate a curried version of expand-1 (Algorithm 3) across $\mathcal{A}$. Each successive call to mapcan further refines

```
(defun mdtd (A)
  (labels ((expand-1 (mu triple) (...))
           (expand (acc mu)
             (mapcan (lambda (triple)
                       (expand-1 mu triple))
                     acc)))
    (let ((S (reduce #'expand A
                     :initial-value '((t (t) (nil))))))
      (values (mapcar #'car S) S))))
```

Figure 4: The `MDTD` code expressed using `mapcan`, `reduce`, and `expand-1` from Algorithm 3. The variables `S` and `M` correspond respectively to the variables $\mathcal{S}$ and $\mathcal{A}$ from Algorithm 3.

the *current* partition by intersecting appropriate type specifiers with μ, its complement $\overline{\mu}$, or both.

A subtle but crucial feature of our `mdtd` implementation is that even if `subtypep` returns *dont-know* (either during a subtype check or disjoint check) we nevertheless construct a well-formed partition of the space. This certainty is because in the worst case, algorithm lines 2.15, 2.16, and 3.7 partition the type, v, into $v_1 = v \cap \mu$ and $v_2 = v \cap \overline{\mu}$. If v and μ are disjoint (despite `subtypep` returning *dont-know*), then $v_1 \subseteq \emptyset$. If $v \subseteq \mu$, then $\overline{\mu} \cap v \subseteq \emptyset$. The consequence is that the computed set, $\mathcal{S}$ might contain type specifiers which specify the empty type, even though we have failed to detect the fact that the types are empty.

3.3 Sample Run

We detail the computation of the types in Figure 3. Figure 5 shows the computation tree. Each call to `mapcan` creates one horizontal level of the tree. The computation starts with the top type, denoted $\top$. Each subsequent level partitions each of the type specifiers in the previous level by intersecting with μ, $\overline{\mu}$ or both. If μ is disjoint with the type in question or a supertype of the type in question, the previous level's type is inherited to the next level.

The second level of the tree intersects $\top$ with A_1 and $\overline{A_1}$. The third level intersects each of $\left\{A_1, \overline{A_1}\right\}$ with A_2 and $\overline{A_2}$.

$$A_1 \cap A_2 = A_2$$
$$A_1 \cap \overline{A_2} = A_1 \cap \overline{A_2}\,.$$

Since $A_2 \subseteq A_1$, line 2.10 is reached. No intersection computation is necessary because the result would either be the empty type or the same type we started with:

$$\overline{A_1} \cap A_2 = \emptyset$$
$$\overline{A_1} \cap \overline{A_2} = \overline{A_1}$$

Similar reasoning continues with the fourth and fifth levels.

3.4 Sketch of Proof of Correctness

We do not present a formal proof, but rather we sketch an informal argument. A formal proof will come in a future publication. To sketch this proof, we'd like to show that the set of types specified by the bottom-most row of Figure 5 is indeed the MDTD of the types $\{A_1, A_2, A_3, A_4, A_5\}$. We need to show three things from Definition 2.1, which we address in Sections 3.4.1, 3.4.2, and 3.4.3.

3.4.1 Union invariance. By induction: To understand that the union of the types specified at level n (of Figure 5) is the same as the union of the types at level $n+1$, we see that each v at level n corresponds either to the same type at level $n+1$, or for some type μ, two types

$$\begin{aligned} v_1 &= v \cap \mu \\ v_2 &= v \cap \overline{\mu} \\ v_1 \cup v_2 &= (v \cap \mu) \cup (v \cap \overline{\mu}) \\ &= v \cap \left(\mu \cup \overline{\mu}\right) \\ &= v \cap \top = v \end{aligned}$$

So refining the partition moving from level-n to level $n+1$ preserves the union, which at the top level is $\top$, or A_1 in the case that $\top$ is not included in the MDTD input.

3.4.2 Disjointness. To understand that the types specified at each level are disjoint, we assume (an inductive proof) that the types at level n are disjoint and prove that the types at level $n+1$ are disjoint. We know this, because each type, v, at level n corresponds either to the same type at level $n+1$ or to two types, v_1 and v_2, where

$$\begin{aligned} v_1 &= v \cap \mu \\ v_2 &= v \cap \overline{\mu} \\ v_1 \cap v_2 &= (v \cap \mu) \cap (v \cap \overline{\mu}) \\ &= (v \cap v) \cap (\mu \cap \overline{\mu}) \\ &= v \cap \emptyset = \emptyset \end{aligned}$$

So we see that v_1 and v_2 are disjoint. If two types, v_1, v_2 at level $n+1$ are derived from the different level-n parents, v_3, v_4 respectively, then we know that v_3 and v_4 are disjoint by inductive hypothesis. Thus there exist μ_1 and μ_2 such that

$$\begin{aligned} v_1 &= v_3 \cap \mu_1 \\ v_2 &= v_4 \cap \mu_2 \\ v_1 \cap v_2 &= (v_3 \cap \mu_1) \cap (v_4 \cap \mu_2) \\ &= (v_3 \cap v_4) \cap (\mu_1 \cap \mu_2) \\ &= \emptyset \cap (\mu_1 \cap \mu_2) = \emptyset \end{aligned}$$

Thus all the types specified at level $n+1$ are disjoint.

3.4.3 Refinement. If $v \in \mathcal{X}$ and $\mu \in \mathcal{A}$, we see that μ is in either the third column (Factors) or the 4th column (Disjoint types) of the table in Figure 3. This fact is guaranteed because we know that the function in Algorithm 3 was called with μ as an argument. Algorithm 3 assures that μ is prepended either to the set of factors or the set of disjoint types.

4 COMPUTING BRZOZOWSKI DERIVATIVE

We saw in Algorithm 1 that the output of `MDTD` is used as input for the Brzozowski derivative on line 1.8. In this section, we show how the metadata collected in `MDTD` helps to compute $\partial_v r$

Recall the $\partial_v r$ is the Brzozowski derivative of RTE, r, with respect to type v. The value or $\partial_v r$ is another RTE.

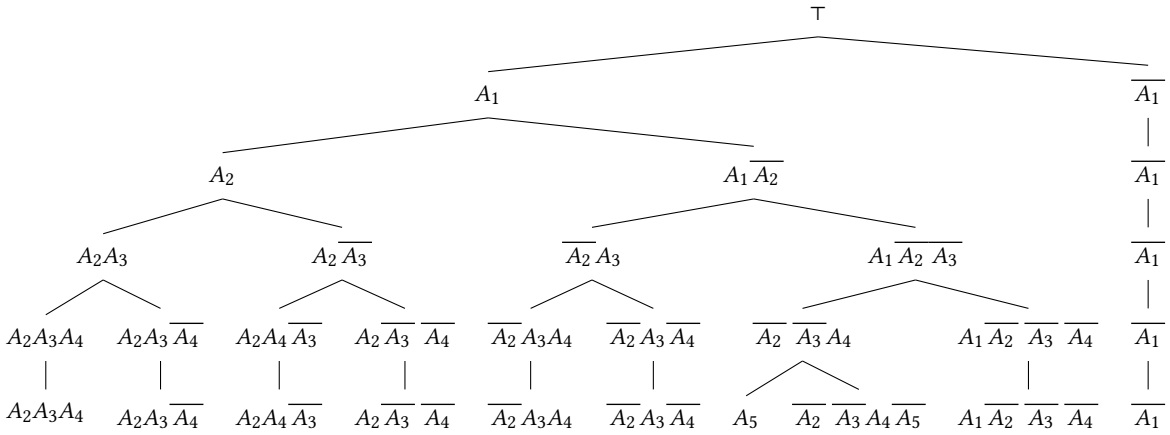


Figure 5: Computation tree computing the MDTD for the types in Figure 3

$$\partial_v\,\emptyset = \emptyset \tag{3}$$
$$\partial_v\,\varepsilon = \emptyset \tag{4}$$
$$\partial_v\,(\neg r) = \neg\partial_v\,r \tag{5}$$
$$\partial_v\,(r^*) = \partial_v\,r \cdot r^* \tag{6}$$
$$\partial_v\,(r \vee s) = \partial_v\,r \;\vee\; \partial_v\,s \tag{7}$$
$$\partial_v\,(r \wedge s) = \partial_v\,r \;\wedge\; \partial_v\,s \tag{8}$$
$$\partial_v\,(rs) = \begin{cases} (\partial_v\,r)s & \text{if } r \text{ not nullable} \\ (\partial_v\,r)s \vee \partial_v\,s & \text{if } r \text{ nullable} \end{cases} \tag{9}$$
$$\partial_v\,\mu = \varepsilon \quad \text{if } [\![v]\!] \subseteq [\![\mu]\!] \tag{10}$$
$$\partial_v\,\mu = \emptyset \quad \text{if } [\![v]\!] \cap [\![\mu]\!] = \emptyset \tag{11}$$
$$\partial_v\,\mu \quad \text{otherwise, no rule defined} \tag{12}$$

Figure 6: Recursive rules for computing Brzozowski derivative of an RTE. These rules are applied in computing $\partial_v\,r$ on line 1.8 of Algorithm 1. v (similarly μ) is a type specifier. $[\![v]\!]$ (similarly $[\![\mu]\!]$) represents the set of values comprising the specified type.

4.1 Computation Details

To compute $\partial_v\,r$, Owens *et al.* [22] suggest a recursive procedure. Newton *et al.* [18] generalized this procedure to work with Common Lisp types. Newton [16] further generalized the recursive rules of this procedure as shown in Figure 6. Rule 9 refers to *nullable*, meaning that the language of r contains the empty sequence. Owens *et al.* explain a simple decision procedure to determine whether a regular expression is *nullable*.

Some explanation is necessary to understand the notation and the implications of Figure 6. Recall that the notation $\partial_v\,r$ means that r is an RTE and v is a type. Programmatically, r is a data structure represented according to the rules of a DSL, which we summarize in Section 2.2.

The v in $\partial_v\,r$ specifies a type. Programmatically, v is represented by a Common Lisp type specifier.

Given a value of r and v, to compute $\partial_v\,r$, the rules in Figure 6 are applied recursively. There are several cases which terminate the recursion.

Rule (3): Every RTE represents a set of Common Lisp sequences. We use the symbol $\emptyset$ to represent the RTE which itself represents the empty set of sequences. Careful, the empty set of sequences is different from the set of empty sequences, denoted by ε. The derivative of $\emptyset$ is again $\emptyset$ regardless of v. The Common Lisp type specifier `nil` specifives the empty type, equivalently empty set.

Rule (4): The symbol ε represents the set of empty sequences. We sometimes represent this set as $\{()\}$; however in Common Lisp, ε includes the empty list, empty array, empty string, etc [15, 23]. The derivative of ε is $\emptyset$ regardless of v.

Rules (10) **and** (11): These rules represent the case where the RTE, μ, is known to represent specifically a set of singleton sequences,[2] *e.g.* the set of singleton sequences whose first (and only element) is an integer, or the set of sequences whose element is a string. In the notation of Figure 6, μ represents the RTE (in turn representing a set of sequences), while $[\![\mu]\!]$ represents the set comprising of the first elements of these sequences: the set of integers, or the set of strings, as opposed to the set of singleton lists of integers or set of singleton lists of strings. In Rule (11), we use $\emptyset$ to represent both the RTE containing no sequences, and also the empty type.

4.2 Complications with `subtypep`

In order to distinguish rules (10) and (11) programmatically, we must know whether one type is a subtype of another, given the type specifiers, or know whether the two specified types are disjoint. The Common Lisp function, `subtypep`, is an obvious implementation choice to make this run-time decision. However, `subtypep` sometimes returns `dont-know`. If this occurs during DFA construction, we cannot determine the value of the derivative. Thus, we must avoid this case.

[2] We can be assured that the μ represents a singleton sequence because we have eliminated all other possiblities in rules 3 through 9; *i.e.*, μ is not $\emptyset$, ε, *, negation, disjunction, conjunction, nor concatenation.

```
(defclass A1 () ())
(defclass A2 (A1) ())
(defclass A3 (A1) ())
(defclass A23 (A2 A3) ()) ; X3 U X5
(defclass A4 (A1) ())
(defclass A423 (A4 A23) ()) ; X5 U X6 U X7
(defclass A5 (A4) ())

(subtypep '(and A3 (not A2) (not A4))
          'A1)          ; returns T, T
(subtypep 'A3 'A2)      ; returns NIL, T
(subtypep '(and (and A3 (not A2) (not A4)) A1)
          nil)          ; returns NIL, NIL
```

Figure 7: Common Lisp code defining classes analogous to Figure 3, also demonstrating successful and unsuccessful calls to `subtypep`.

This weakness of `subtypep` is a significant problem for the Brzozowski derivative computation, a limitation which we alleviate with our proposed MDTD procedure. As an illustration of the problem, suppose that we have an RTE, r, representing a singleton sequence whose element has type X_4 from Figure 3, and suppose we need to compute $\partial_v r$ where $v = A_1$. We need to determine whether $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right) \subseteq A_1$ or whether $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right) \cap A_1 = \emptyset$. It is not syntactically obvious which (if either) is the case; there is no mention of A_1 within $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right)$. The human (or a sufficiently intelligent `subtypep`) can of course answer this question by reasoning that A_3 is mentioned and $A_3 \subseteq A_1$. This reasoning only works if A_3 and A_1 are specified by very simple type specifiers, such as a class name. If on the other hand, either or both of A_3, A_1 are type specifiers involving Boolean combination types such as `(and ...)`, `(or ...)`, `(not ...)`, or `(satisfies ...)`, such reasoning would be less obvious and more compute intensive.

The Common Lisp code in Figure 7 defines classes which have the same disjoint and subtype relations as in Figure 3. The code contains three calls to `subtypep` to determine whether see how SBCL and CLISP handle these calls to `subtypep`. The SBCL [15] implementation responds `T, T` for the first, indicating that the subtype relation, $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right) \subseteq A_1$. The second example returns `NIL, T`, indicating that $A_3 \not\subset A_2$. However the third call which asks whether $\left(\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right) \cap A_1\right) \subseteq \emptyset$ returns `NIL, NIL` indicating *dont-know*. We get the same results in CLISP [7].

If we use `subtypep` to answer these questions, `subtypep` is allowed (by the Common Lisp specification) to return *dont-know*. However, we do not need to rely on `subtypep` in this case, because as we also see in the third column row X_7 of Figure 3 (Bottom) that A_1 is guaranteed by construction to be a supertype of $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right)$.

Even if the `subtypep` implementation in your particular implement of Common Lisp is intelligent enough to determine this subtype relation, doing so would necessarily be computation intensive. Our MDTD procedure avoids this redundant complexity.

5 PERFORMANCE ANALYSIS

We have taken as benchmarks, the performance analysis presented in [16, Ch 10]. In that work, Newton analyzed (ad nauseam) performance characteristics of various MDTD algorithms on various genre of input types, without conclusive results. We repeated some of those performance comparisons with the procedure presented in this paper. The experiments are summarized here. We considered the following algorithms, a subset of those presented in [16].

(1) `mdtd-bdd` – A primitive base-line algorithm using BDDs [4] as data structure to designate a type.
(2) `mdtd-graph` – A graph based algorithm also described in [19], using Common Lisp type specifiers (s-expressions) as type designators.
(3) `mdtd-bdd-graph` – Same algorithm as `mdtd-graph` but using BDDs as type designators.
(4) `mdtd-padl` – The procedure in Algorithm 2.

The reference benchmarks were divided into so-called *pools*. A pool is a set of type specifiers, chosen with similar characteristics; *e.g.*, a set of `(member ...)` types, or a set of floating point range types, or all predefined subtypes of number.

We show the results for several pools:

- **MEMBER types** – Types such as

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

- **CL combinations** – Unions and intersections of types whose name come from the `common-lisp` package. Examples include

```
(or print-not-readable structure-class)
(and simple-string bignum)
(or standard-char double-float)
(or class storage-condition)
```

- **Real number ranges** – numerical ranges of `integer`, `real`, and `float`. Examples include

```
(INTEGER 60 (79))
(REAL 1/36 47/9)
(FLOAT 55.142532 60.722794)
```

- **Subtypes of NUMBER** – Subtypes of `number` and Boolean combinations of them. Examples include

```
short-float
(and short-float (not unsigned-byte))
(or short-float unsigned-byte)
unsigned-byte
(and number (not bit))
rational
```

- **CL types** – Symbols from `common-lisp` package which designate types. Examples include

```
arithmetic-error
function
simple-condition
array
```

Running a benchmark consists of selecting successively larger sets of type specifiers from the pool in question, and calling the `mdtd` function, measuring the execution time. Figure 8 shows the benchmark results. See the legend on the bottom-right of the figure for the color scheme.

We see that for a small number of input types, the `mdtd-padl` algorithm performs poorly compared to the others, in time ranges of less than 0.1 millisecond. However, for times greater than 1 millisecond, the algorithm performs well. In the top two plots in Figure 8, `mdtd-padl` is the best performing, at least in the asymptotic case. In the bottom-most plot, CL types, we see that `mdtd-padl` performs worse by an order of magnitude. However, for most of the cases, in the middle of the figure, the performance is very good but outperformed by the BDD-based algorithm.

6 CONCLUSION

6.1 Results

In this work, we have introduced a new algorithm for computing the Maximal Disjoint Type Distribution of a given set of type specifiers. Our experiments show that the procedure is usually faster than previously reported procedures, and also provides data which makes the Brzozowski derivative easier to compute. While the improvements we have discussed here have also been applied to our RTE implements in Clojure, Scala, and Python, we have only addressed herein the aspects relating to Common Lisp.

Our MDTD procedure alleviates some of the consequences of the incompleteness and compute intensity of `subtypep`. The fact that certain dependence on direct calls to `subtypep` is elided, has the effect of eliding certain unnecessary computations, potentially making the Brzozowski derivative computation faster than it otherwise might be. In addition to computation speed, we also enable the algorithm to produce a correct (even if suboptimal) result despite having a less powerful implementation of `subtypep` in your Common Lisp implementation. The MDTD algorithm is guaranteed to compute a set of types which are disjoint; however, they may not be provably inhabited.

6.2 Perspectives

We see in Figure 8 that the BDD-based MDTD procedure outperforms `mdtd-padl`, but not significantly so. We would like to refactor the BDD-based procedure to use the approach of `mdtd-padl` but applied to BDDs rather to s-expression based type specifiers.

We have not yet extensively investigated the application of our algorithm to type-system related computations on JVM languages such as Clojure and Scala. Sometimes questions of subtype-ness and habitation/vacuity cannot be answered about JVM-based types, because we do not know at computation time which shared libraries may be dynamically loaded later in the running program. Our current model in the RTE implementation in Clojure and Scala uses a so-called *world-view*. An closed world-view means that we assume no new classes will be added, and an open world-view means we never know the entire list of subclasses of a given class. A open world-view is predicted to result in larger DFAs with more unsatisfiable transitions. However, we do not yet have data to confirm or measure this effect.

REFERENCES

[1] Seqexp: regular expressions for sequences, 2014. URL https://github.com/cgrand/seqexp.

[2] David nolen on parsing with derivatives, 2016. URL https://www.youtube.com/watch?v=FKiEsJiTMtI.

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

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

[5] Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN 0004-5411. doi: 10.1145/321239.321249. URL http://doi.acm.org/10.1145/321239.321249.

[6] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification, 29th International Conference (CAV'17)*. Springer, July 2017. URL https://www.microsoft.com/en-us/research/publication/power-symbolic-automata-transducers-invited-tutorial/.

[7] Bruno Haible. Gnu clisp, 2010. URL https://clisp.sourceforge.io.

[8] Philip Hazel. PCRE - Perl Compatible Regular Expressions, 2015. URL www.pcre.org.

[9] Mikko Heikkilä. Malli, metosin, 2022. URL https://github.com/metosin/malli.

[10] Rich Hickey. The Clojure Programming Language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

[11] Rich Hickey. A History of Clojure. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi: 10.1145/3386321. URL https://doi.org/10.1145/3386321.

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

[13] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 189–195, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308656. doi: 10.1145/2034773.2034801. URL https://doi.org/10.1145/2034773.2034801.

[14] Alex Miller. spec guide, 2022. URL https://clojure.org/guides/spec.

[15] William H. Newman. Steel Bank Common Lisp User Manual, 2015. URL http://www.sbcl.org.

[16] Jim Newton. *Representing and Computing with Types in Dynamically Typed Languages*. PhD thesis, Sorbonne University, November 2018.

[17] Jim Newton and Adrien Pommellet. A portable, simple, embeddable type system. In *European Lisp Symposium*, Online, Everywhere, May 2021.

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

[19] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic Manipulation of Common Lisp Type Specifiers. In *European Lisp Symposium*, Brussels, Belgium, April 2017.

[20] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Sigplan Notices - SIGPLAN*, volume 40, pages 41–57, 10 2005. doi: 10.1145/1103845.1094815.

[21] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification, 2004.

[22] Scott Owens, John Reppy, and Aaron Turon. Regular-expression Derivatives Re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009. ISSN 0956-7968.

[23] Christophe Rhodes. User-extensible Sequences in Common Lisp. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 13:1–13:14, New York, NY, USA, 2009. ACM. ISBN 978-1-59593-618-9. doi: 10.1145/1622123.1622138. URL http://doi.acm.org/10.1145/1622123.1622138.

[24] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. ISBN 1906966141, 9781906966140.

[25] Edmund Weitz. *Common Lisp Recipes: A Problem-solution Approach*. Apress, 2015. ISBN 978-1-4842-1177-9.

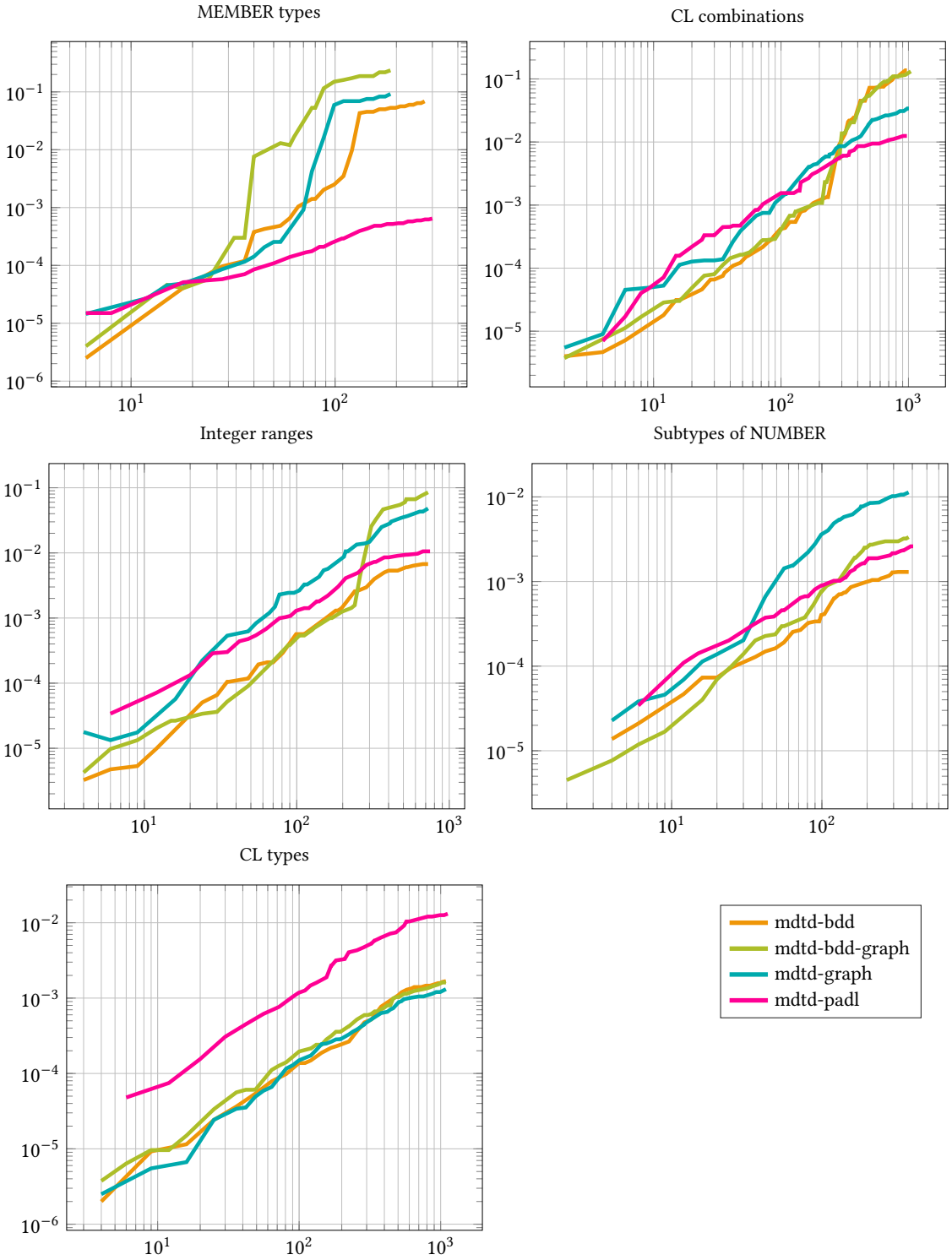


Figure 8: Results of benchmark experiments: Lower is better. Each graph has number of given types as x-axis, and average computation time in seconds, as y-axis. The pink/magenta curve indicates the results for `mdtd-padl`, that being the algorithm described in this paper.