Recognizing Heterogeneous Sequences in a Simple Embeddable Type System

Jim Newton[0000-0002-1595-8655]

EPITA Research Lab, 94270 Le Kremlin Becêtre, FRANCE jnewton@lrde.epita.fr

Abstract. Heterogeneously typed sequences are frequently used in dynamically typed programming languages, but can also be used in statically typed languages supporting run-time type reflection. Such sequences often exhibit type patterns such as repetition, alternation, and optionality. The programmer needs a mechanism to declare or query adherence to these regular patterns. The theory of finite automata alphabets characterizes such patterns into so-called regular languages, but does not exactly meet this challenge, because the set of potential elements of the sequences, the alphabet, is infinite. In this article, we present a generalization of regular expressions called rational type expressions as a means of declaring regular patterns in heterogeneous sequences. We present procedures for constructing a symbolic finite automata using an embeddable type system. We provide a solution for a certain class of subtype decidability relations where subtypeness can be ensured by construction. We demonstrate the generality and portability of the system by providing implementations in Common Lisp, Clojure, Scala, and Python.

1 Introduction

This paper studies rational type expressions (RTEs) and the construction from RTE to symbolic finite automata (σDFA). RTEs can be used to specify patterns in the types of the sequence elements, such as $(\mathit{int} \cdot \mathit{string} \cdot \mathit{even})^*$, and σDFAs are used to efficiently decide the language induced by the RTE. The main challenges in the system are: (1) how to define types and (2) how to construct σDFA from an RTE. For the first challenge, we manipulate types using "a Simple Embeddable Type System" from Newton and Pommellet [27]. For the second challenge, we use Brzozowski style derivative-based construction, and we solve the challenge of overlapping types using Maximal Disjoint Type Decomposition.

Before looking into the theory and implementation, we introduce the rational type expression (RTE) and the deterministic, complete, symbolic, finite automaton (σDFA) simply with an extended example. Our goal is to declaratively describe a set of sequences (rational language) in a programming language based on regular patterns in the types of the sequence elements, and to efficiently decide membership of these rational languages at run-time.

1.1 Illustrative Examples

The syntax of so-called *rational type expressions* [26] should be intuitive to anyone already familiar with regular expressions.

$$r_0 = (int \cdot string \cdot even)^* \tag{1}$$
$$r_1 = (int \cdot string^* \cdot even)^* \tag{2}$$
$$r_2 = (int \cdot string \cdot string^* \cdot even)^* . \tag{3}$$

To avoid limiting ourselves to a single programming language, we define *type* portably [27]. RTE, r_0 (1), represents the set of sequences of arbitrary (finite) length, in a given programming language, each of which consists of zero or more occurrences: integer, string, even integer. r_1 expands r_0 to allow the string to occur 0 or more times. Finally, r_2, requires the string occur at least once.

Analogous to classical finite automata [16], RTEs correspond to *symbolic* finite automata [9] (σDFA) over a possibly infinite alphabet, Σ—labeled by subsets of Σ. Σ is the set of all values representable in the programming language. RTEs, r_0 and r_2, are implemented in Figure 1 [left] and [right] respectively, r_0 being represented by a deterministic automaton and r_2 non-deterministic. The

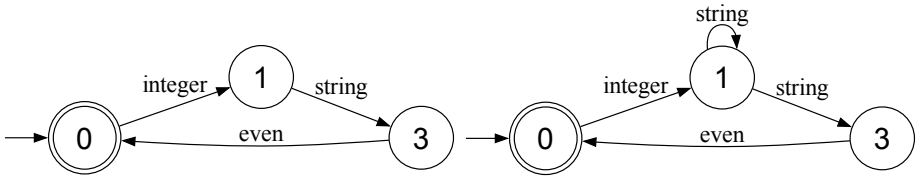


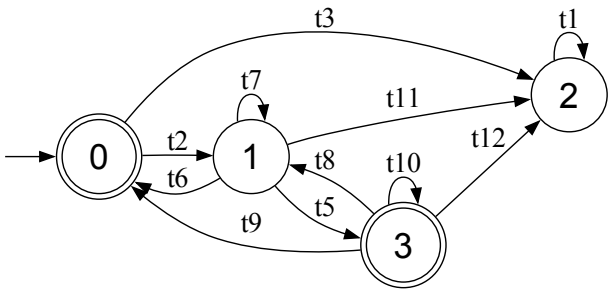
Fig. 1: σDFA for RTEs: $r_0 = (int \cdot string \cdot even)^*$ [(1) left] and non-deterministic automaton for $r_2 = (int \cdot string \cdot string^* \cdot even)^*$ [(3) right]

expense of constructing finite automata (classical or symbolic) can be mitigated by precomputing at compile-time (as in Common Lisp [2]) or using memoization. Once the σDFA is constructed, the time complexity of the membership decision is linear in length of sequence in question, constant in memory complexity, and no longer a function of the potentially large representation of the RTE itself.

Much of the work in σDFA construction focuses on preventing overlap between transitions, thus enforcing determinism. The fact that the subtype relation is not always decidable makes it more difficult to guaranteed determinism.

Why is the subtype relation important? Knowing the subtype relation is critical for proving other relations between types. For types, ν and μ, we prove $\nu = \mu$ by proving $(\nu \subseteq \mu) \wedge (\mu \subseteq \nu)$. ν is provably vacuous, if we prove $\nu \subseteq \emptyset$. ν and μ are disjoint, if we prove $\nu \cap \mu \subseteq \emptyset$.

Our solution to the subtype problem, as it relates to σDFA construction, is not to create a subtype predicate which always returns the correct result, as



$t_1 = \Sigma$ $\quad t_7 = string \cap \overline{even}$

$t_2 = int$ $\quad t_8 = (int \cap \overline{even}) \cup (string \cap \overline{even})$

$t_3 = \overline{int}$ $\quad t_9 = \overline{int} \cap \overline{string} \cap even$

$t_4 = string$ $\quad t_{10} = (int \cap even) \cup (string \cap even)$

$t_5 = string \cap even$ $\quad t_{11} = \overline{string} \cap \overline{even}$

$t_6 = \overline{string} \cap even$ $\quad t_{12} = \overline{int} \cap \overline{string} \cap \overline{even}$

Fig. 2: σDFA for $r_1 = (int \cdot string^* \cdot even)^*$ (2) deterministic, complete.

that would be impossible. Rather our solution is pragmatic. SETS (Section 4.1) proposes a subtype ($\subseteq$) predicate which always returns *true*, *false*, or *dont-know*, and a membership predicate ($\in$) which always returns *true* or *false*.

As an example of a problematic type, consider the type *even*. This so-called $satisfies$ type is constructed using a predicate, which returns `true`, given an even integer, and `false` otherwise. The system is unable to predict the set of values for which an application-specific predicate returns `true`.

In the σDFA in Figure 2, the outgoing transitions for each state are labeled with mutually disjoint types. *E.g.*, state ① has four exiting transitions: t_5, t_6, t_7, and t_{11} which are mutually disjoint and a partition: $t_5 \cup t_6 \cup t_7 \cup t_{11} = \Sigma$.

Unsatisfiable transitions are common and lead to problems in minimization. Whereas classical finite automata are always uniquely minimizable, σDFAs are not. The system cannot determine that $t_5 = \emptyset$, nor that $t_4 = t_7$. State ③ *could* be eliminated, as well as associated transitions, because its label, t_5, is not satisfiable. However, we cannot determine programmatically that this elimination is possible, without a sufficiently intelligent oracle.

One might suppose that the fact that we cannot always determine whether arbitrary types are disjoint will inevitably lead to non-deterministic transitions, but it does not. For σDFAs, undecidable does not imply non-deterministic. Why? Because independent of whether $\mu \subseteq \nu$, or whether μ and ν are disjoint, it is certain that $\mu \cap \nu$, $\mu \cap \overline{\nu}$, $\overline{\mu} \cap \nu$, and $\overline{\mu} \cap \overline{\nu}$ are mutually disjoint and a partition of Σ. Non-satisfiable transitions yield *undetectable* useless states, but not non-deterministic σDFA. These redundant transitions may result in needless computation at compile-time and at run-time, but computations which lead to correct, perhaps sub-optimal results.

1.2 Our Contribution

Our contributions in this article are as follows. We

1. Adapt the theoretical description of D'Antoni and Veanes [9] to practical programming languages: (Section 6)
2. Provide a simple and comprehensible algorithm for the maximal disjoint type decomposition, `MDTD`, which avoids, by construction, the problem of undecidability of subtyping.
3. Construct correct σDFAs from RTEs even in presence of undecidable subtype relations.

1.3 Previous Work

In [21], Newton used DFA to recognize certain heterogeneous Common Lisp sequences based on the types.

The `clojure-rte` implementation is reminiscent of the Clojure Spec library [20] and `metosin` [13]. Although Hickey [15, 14] mentions Spec's existence, we have otherwise found no peer reviewed articles on either. 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. Bonnaire [5] presents *typed Clojure*, without rigorously defining a Clojure *type*, nor anything as ambitious as Common Lisp's `SUBTYPEP`.

The Brzozowski [8] derivative and an algorithm to compute it dates to 1964. Owens *et al.* [31] presented a *modern* version. Owens noted that practical obstacle to using this approach is the large computation time of generating large finite automata related to excessively large alphabets. Our approach ameliorates this problem by considering sets rather than individual values.

D'Antoni and Veanes [9] argue that symbolic finite automata retain many of the good properties of their finite-alphabet counterparts. In the same work they discuss a decomposition of types which they refer to as *Minterms(...)* which they describe as *the set of maximal satisfies Boolean combinations*. D'Antoni's set is a less optimized version of our MDTD algorithm (Section 5.5) which attempts to minimize the number of terms by taking advantage of known subtype and disjoint relations.

Grigore [12] addresses subtype decidability in Java [11]. Kennedy and Pierce [19] identify several type system restrictions which make the question decidable. They investigate restricting these capabilities in different ways in Java [11], Scala [29, 30], C#, and (curiously without citation) .NET Intermediate Language.

In Common Lisp [2], `SUBTYPEP` can be compute-intensive in general. An implementation may return *dont-know* when estimated *too compute intensive*. [3, 33] Common Lisp is specified (without explanation) to signal an error if run-time type membership check is performed on non-trivial function types. Presumably this is motivated by the fact that `SUBTYPEP` may return *dont-know*, but the type-membership check, `typep`, must return `true` or `false`, which would be impossible in certain membership checks for functions which themselves require

subtype checks. SETS (Section 4.1) circumvents this problem by avoiding support for function types, other than that already supported in the host language.

Hosoya, Vouillon, and Pierce [17] defined regular expression types and the XDuce language. XDuce allows static XML types to be defined recursively and hierarchically. XDuce enables programs to consume and manipulate XML [6] documents while the type checker assures adherence to the XML schema.

A notable distinction of the RTE vs XDuce is that our implementation adds such type checking ability to an existing dynamic type system.

2 Symbolic Finite Automata

We consider finite automata as shown in Figure 2, called *symbolic deterministic complete finite automata* (σDFA). A σDFA differs from a classical finite automaton in two significant ways. (1) The alphabet, Σ, may be infinite; and (2) Each transition is labeled by a symbol representing a possibly infinite subset of Σ.

Definition 1 (symbolic finite automaton, transition,label).
A symbolic finite automaton *is a structure:* $A = (\Sigma, Q, q_0, F, T)$ *where:*

1. Σ *is a possibly infinite set of* objects.
2. Q *is a finite set of* states.
3. $q_0 \in Q$ *is the* initial *state.*
4. $F \subseteq Q$ *is the set of* accepting *states.*
5. $T \subseteq Q \times 2^{\Sigma} \times Q$ *is a set of transitions.*

A transition, $(q, \nu, r) \in T$*, may also be denoted* $q \xrightarrow{\nu} r$*. We refer to* ν *as the label.*

A symbolic finite automaton is called complete *and* deterministic*, and denoted* σ*DFA, if in addition, for each* $q \in Q$*, the set labels of transitions having origin* q *is a partition of* Σ*.* △

A transition $q \xrightarrow{\nu} r$ is called *satisfiable* if ν is inhabited ($\nu \neq \emptyset$); otherwise, it is called *non-satisfiable*. It is called *indeterminate*, if our subtype procedure cannot determine whether $\nu \subseteq \emptyset$.

3 Rational Expressions

We take some definitions directly from classical finite automata theory. A *sequence* has finite length, $n \geq 0$, and is a function mapping $\{0, 1, \ldots, n-1\} \to \Sigma$. A *language* is a set of sequences. The set of all sequences is called Σ^*. The symbol () represents the empty sequence. The language, ε, contains only (); $\varepsilon = \{()\}$. Concatenating two sequences, s, t forms a new sequence, $s \cdot t$. Similarly, two languages, L_1 and L_2 can be concatenated to form $L_1 \cdot L_2 = \{s \cdot t \mid s \in L_1, t \in L_2\}$. If $x \in \Sigma$, and s is a sequence of length $n \geq 0$, then the *cons*, $x :: s = (x) \cdot s$, is a sequence of length $n + 1$. Finally, the Kleene star of a language, L^*, is the set of all possible finite concatenations of zero or more sequences from L.

There is a unique subset of Σ^* which is *recognized* by a given σDFA, called the *language* of the σDFA. We may define a regular expression (not uniquely) as an expression tree corresponding to the language of the σDFA. Given a regular expression, there are many σDFAs which recognize the language.

Definition 2 (rational type expression). *A* rational type expression *(RTE), r, is an expression which represents (generates) a language $[\![r]\!] \subseteq \Sigma$. $\emptyset$ is an RTE with $[\![\emptyset]\!] = \emptyset$. If ν designates a subset of Σ, then ν is an RTE with $[\![\nu]\!] = \{(a) \mid a \in \nu\}$, including $[\![\Sigma]\!] = \{(a) \mid a \in \Sigma\}$. Let f and g be RTEs, with $F = [\![f]\!]$ and $G = [\![g]\!]$, then $f \cdot g$, $f \vee g$, $f \wedge g$, $\neg f$, and f^* are RTS with $[\![f \cdot g]\!] = F \cdot G$, $[\![f \vee g]\!] = F \cup G$, $[\![f \wedge g]\!] = F \cap G$, $[\![\neg f]\!] = \Sigma^* \setminus F$, and $[\![f^*]\!] = F^*$.* $\triangle$

4 Programming Language Supporting RTEs

We pragmatically restrict Σ to the set of objects representable in a given programming language. Σ^* denotes the set of finite, non-circular, sequences with components in Σ. We suppose that the programming language supports a set, Υ_0, of built-in types and user-defined types, such as `int` and `string`, which allow an application program to perform run-time type membership checks. We suppose that the program language provides a mechanism for performing subtype checks between types in terms of the symbols in Υ_0.

4.1 A Simple Embeddable Type System

Newton and Pommellet [27] formally defined *Simple Embeddable Type System* (SETS) which we use here. SETS is simple enough to be implemented in a wide range of programming languages. The simplicity and flexibility of SETS is demonstrated in that it is built into Common Lisp, and we provide SETS implementations in Clojure [22], Scala [25], and Python [24], each as part of the larger RTE implementation. We summarize SETS here.

Definition 3 (type). *A type is any set of values, i.e. any subset of Σ.* $\triangle$

We distinguish a *type* from a *type designator*. We typically denote a type designator by the symbol, ν, and the corresponding type by $[\![\nu]\!]$. The set of all types is 2^Σ, while the set of all type designators is denoted, Υ, and is recursively defined as follows: $\Upsilon_0 \subseteq \Upsilon$, with semantics defined by the host language. $\Sigma, \emptyset \in \Upsilon$, with obvious semantics. Union, intersection, and complement types are defined with semantics corresponding to the associated set operations: $\nu_1 \cup \nu_2 \in \Upsilon$, $\nu_1 \cap \nu_2 \in \Upsilon$, and $\overline{\nu_1} \in \Upsilon$. Singleton types: $\forall a \in \Sigma$, $\{a\} \in \Upsilon$. Predicate types, for any total, pure, function, $f : \Sigma \to \{true, false\}$, implemented in the host language, $\mathfrak{Sat}(f) \in \Upsilon$ and denotes the set $\{x \in \Sigma \mid f(x) = true\}$.

SETS and RTE are related as follows: if $\nu \in \Upsilon$, then the RTE, ν, is the set of all singleton sequences (a) where $a \in [\![\nu]\!]$.

A programming language specific API for SETS must implement type designators and $\in$, $\subseteq$ procedures. The membership predicate, $\in$, must be decidable

and return always *true*, *false*. However, as the subtype relation is sometimes undecidable (or excessively compute intensive) $\subseteq$ must return *true*, *false*, or *dont-know* in a programming language specific way. To reiterate, we require that the $\subseteq$ predicate in SETS not loop forever—either it returns *true* or *false* indicating that the subtype relation has been proven or disproven, or it gives up and returns *dont-know*.

As an example: transition, $① \xrightarrow{string \cap even} ③$, in Figure 2, is indeterminate. The SETS $\subseteq$ procedure cannot determine that $(string \cap \mathfrak{Sat}(evenp)) \subseteq \emptyset$. Nevertheless, the transition behaves correctly. Given the string, `"hello"`, at run-time, the SETS $\in$ procedure will determine that "$hello$" $\notin$ $(string \cap \mathfrak{Sat}(evenp))$.

5 Construction of σDFA

In this section we describe σDFA construction using a generalization of the Brzozowski [8, 31]. Other construction algorithms, such as Thompson, are possible but not discussed in this article due to lack of space.

5.1 Brzozowski Derivative Construction of an σDFA

While Brzozowski/Owens [8, 31] defined $\partial_a r$ (the derivative of regular expression, r, with respect to letter $a \in \Sigma$), we present $\partial_\nu r$ for type $\nu \in \Upsilon$.

Definition 4 (derivative, $\partial_\nu r$). *If $S \subseteq \Sigma^*$, and $\nu \in \Upsilon$ is a type designator designating type $[\![\nu]\!]$, then the* derivative *of S* with respect to *ν is defined as:*
$\partial_\nu S = \{s \in \Sigma^* \mid h :: s \in S \text{ for some } h \in [\![\nu]\!]\}$ $\triangle$

Consider the RTE from Equation (2): $r_1 = (int \cdot string^* \cdot even)^*$, and let S be the language of r_1. Each sequence is either () or begins with an *int*.

$\partial_{int} S$ is the set generated by

$$\partial_{int} (int \cdot string^* \cdot even)^* = string^* \cdot even \cdot (int \cdot string^* \cdot even)^*$$

because if we take the subset of S containing sequences starting with an *int*, which is $S \setminus \{()\}$, then strip off the head (an *int*) from each sequence. Each of the remaining tails consists of zero or more *string* followed by *even*, then followed by zero or more occurrences of $(int \cdot string^* \cdot even)$.

We compute the derivative of an RTE by recursively applying reduction rules (4) through (12). Variables, ν and μ represent type designators, $[\![\nu]\!], [\![\mu]\!] \subseteq \Sigma$; while r and s represent RTEs. In particular, we introduce subtype based rules (10), (11) and (12) which replace equivalence based rules which Owens [31] succinctly stated. The symbol μ represents a type, but placing it in the position of an RTE in $\partial_\nu \mu$ is abusive notation, meaning an RTE representing a set of singleton sequences, whose first (and only) elements are of type μ.

$$\partial_\nu \emptyset = \partial_\nu \varepsilon = \emptyset \quad (4)$$

$$\partial_\nu (\neg r) = \neg \partial_\nu r \quad (5)$$

$$\partial_\nu (r^*) = \partial_\nu r \cdot r^* \quad (6)$$

$$\partial_\nu (r \vee s) = \partial_\nu r \ \vee \ \partial_\nu s \quad (7)$$

$$\partial_\nu (r \wedge s) = \partial_\nu r \ \wedge \ \partial_\nu s \quad (8)$$

$$\partial_\nu (rs) = \begin{cases} (\partial_\nu r)s & \text{if } () \notin [\![r]\!] \\ (\partial_\nu r)s \vee \partial_\nu s & \text{if } () \in [\![r]\!] \end{cases} \quad (9)$$

$$\partial_\nu \mu = \varepsilon \quad \text{if } [\![\nu]\!] \subseteq [\![\mu]\!] \quad (10)$$

$$\partial_\nu \mu = \emptyset \quad \text{if } [\![\nu]\!] \cap [\![\mu]\!] = \emptyset \quad (11)$$

$$\partial_\nu \mu \quad \text{otherwise, no rule defined} \quad (12)$$

Correctness of Rule (5) is discussed in Appendix 1.

5.2 Constructing States and Transitions

Given an RTE, r, we may construct all the states of the corresponding σDFA by an iterative procedure outlined in Algorithm 2 (Appendix 2). The first iteration is to compute $\partial_\nu r$ for each ν in a specially computed partition of Σ. The purpose of the MDTD procedure, Algorithm 1 (Section 5.5) is to determine this partition, as a function of the RTE in question. Once the set of derivatives is computed, we associate each RTE with a new state. We repeat this process computing the derivatives for each newly discovered RTE and associating each newly discovered RTE with a new state. The process terminates because sometimes the RTE computed will be equivalent to an RTE already encountered in a previous iteration, and eventually an iteration will produce no new RTEs. Brzozowski [8, 31] argues that the set of all such derivatives is finite.

Given the set of states, the transitions of the σDFA are formed as such: if states q_1 and q_2 correspond to RTEs r_1 and r_2, and $r_2 = \partial_\nu r_1$ then we construct a transition, $q_1 \xrightarrow{\nu} q_2$. *I.e.*, the type designator becomes the label of the transition. If there is no transition directly from q_1 to q_2, this means there is an implicit, unsatisfiable, transition $q_1 \xrightarrow{\emptyset} q_2$.

Finally, we flag the *accepting* states. A state, q_i is accepting if $() \in [\![r_i]\!]$. Owens [31] provides a simple decision procedure which we omit in this article.

5.3 Dealing with Overlapping Types

Our generalization of the Brzozowski construction encounters a challenge which the classical Brzozowski construction does not. In particular, we must assure that the only recursion-terminating cases that occur are (4), (10) or (11), and never (12). To accomplish this, it suffices to assure that the set of types visited by ν for a given RTE, never partially overlap each other. Otherwise stated, the set of types which label transitions from any given state must partition Σ.

Rather than calculating the derivative at each state with respect to each (possibly intersecting) type mentioned in the RTE, instead we calculate a disjoint set of types, then compute the derivatives with respect to this potentially larger set of disjoint types. The exact set of disjoint types, and the manner to compute it is discussed in Section 5.5.

The derivative rules, (4) through (12), are similar to those Owens [31] mentions; we have replaced equivalence checks with subtype checks. When computing

$\partial_\nu \mu$ we must consider several cases: $\nu \subseteq \mu$, in which case rule (10) applies and $\partial_\nu \mu$ reduces to ε; or $\nu \cap \mu = \emptyset$, in which case rule (11) applies and $\partial_\nu \mu$ reduces to $\emptyset$. If $\nu \not\subseteq \mu$ and $\nu \cap \mu \neq \emptyset$, then we define no rule to compute the derivative. The algorithm must avoid any such an attempted computation.

A caveat of our enhanced algorithm is that we must *ask* whether $\nu \subseteq \mu$ or whether $\nu \cap \mu = \emptyset$. This poses a challenge because the query might return *dont-know*. One might naively think that if we could avoid the subtype problem by deconstructing the types ν and μ (as discussed in Section 1) to $\{\nu \cap \mu, \nu \cap \overline{\mu}, \overline{\nu} \cap \mu, \overline{\nu} \cap \overline{\mu}\}$, then the undecidability problem would be averted. Unfortunately, we would still need to pose question of subtypeness to determine which if any of those intersections is empty.

We do not attempt to solve the undecidability problem, rather we solve the problem of computing the derivative in our `MDTD` algorithm (Section 5.5), which assures knowledge of subtype and disjointness *by construction*. *I.e.*, every time we need to compute $\partial_\nu \mu$, we know by construction whether $\nu \subseteq \mu$ or $\nu \cap \mu = \emptyset$.

Our extension step has two positive effects on the algorithm. 1) it enforces determinism, *i.e.*, we ensure that all the transitions *leaving* a state specify disjoint types, and 2) it forces our treatment of the problem to comply with the assumptions required by the Brzozowski/Owens algorithm.

5.4 Avoiding Redundant Run-time Type Checking

It should be noted that cases such as state ① in Figure 2 are common. The labels of the transitions exiting ① have redundant leaf-level type designators. *E.g.*, $t_5 = \mathit{string} \cap \mathit{even}$, $t_6 = \overline{\mathit{string}} \cap \mathit{even}$, $t_7 = \mathit{string} \cap \overline{\mathit{even}}$, and $t_{11} = \overline{\mathit{string}} \cap \overline{\mathit{even}}$ all involve both *string* and *even*; some labels such as $t_8 = \left(\mathit{int} \cap \overline{\mathit{even}}\right) \cup \left(\mathit{string} \cap \overline{\mathit{even}}\right)$ mention the same type name multiple times within itself. If at run-time these type tests are made in a naive, straightforward way; *i.e.*, if the system simply subjects a given object in turn to each of the types for a membership test, then unfortunately the same type tests will occur multiple times in order to identify the correct state transitions. This test can be done efficiently, either by using Binary Decision Diagrams [7, 28], or by a simpler approach using a simple if-then-else tree. Each level of an if-then-else tree test membership to exactly one atomic type, say ν, and `true` branches to a subtree which factors out ν by replacing it and all its supertypes with `true`, otherwise branches to a subtree which factors out ν by replacing it and all its subtypes with `false`. The tree traversal continues until a leaf-node is encountered; the leaf-node indicates the correctly selected destination state in the σDFA.

5.5 Maximal Disjoint Type Decomposition (MDTD)

In [21] Newton presents and analyzes several algorithms to compute the `MDTD` (Maximal Disjoint Type Decomposition). We present here a much cleaner and more understandable algorithm than previously published. The new version of the algorithm has two other advantages which we consider a significant contribution. (1) it is provably correct (planned for future publication because of lack

Input: $\mathcal{M}$: a set of type designators
Output: $\mathcal{S}$: a set of triples as described in Section 5.5

```
1.1  begin
1.2  |  S ← {(⊤, {⊤}, {⊥})}                          // working list of triples
1.3  |  for μ ∈ M do
1.4  |  |  for (ν, f, d) ∈ S do
1.5  |  |  |  S ← S \ {(ν, f, d)}                     // remove triple from S
1.6  |  |  |  if μ ∩ ν = ∅ then
1.7  |  |  |  |  S ← (ν, f, μ :: d) :: S             // ν and μ are disjoint
1.8  |  |  |  else if ¬μ ∩ ν = ∅ then
1.9  |  |  |  |  S ← (ν, μ :: f, d) :: S                              // ν ⊆ μ
1.10 |  |  |  else
1.11 |  |  |  |  S ← (μ ∩ ν, μ :: f, d) :: S                        // μ ∩ ν ⊆ μ
1.12 |  |  |  |  S ← (¬μ ∩ ν, f, μ :: d) :: S        // ¬μ∩ν and μ are disjoint
1.13 |  return S
```

Algorithm 1: Compute `MDTD` of given $\mathcal{M}$.

of space), and (2) it provides metadata which permits σDFA construction even in light of undecidability of subtype or disjoint procedures.

Given a set of potentially intersecting types, $\mathcal{M} = \{\mu_1, \mu_2, \ldots, \mu_n\}$, Algorithm 1 computes a sequence, $\mathcal{S}$, of m triples,

$$\mathcal{S} = ((\nu_1, \mathcal{F}_1, \mathcal{D}_1),\ (\nu_2, \mathcal{F}_2, \mathcal{D}_2),\ \ldots,\ (\nu_m, \mathcal{F}_m, \mathcal{D}_m))\,. \tag{13}$$

The sequence $\mathcal{U} = (\nu_1, \nu_2, \ldots, \nu_m)$ has the following properties:

1. **Union invariance**: $\bigcup_{i \leq m} \nu_i = \Sigma$
2. **Disjointness**: $\nu_i \cap \nu_j = \emptyset$ for all $i \neq j$.
3. **Refinement**: For $i \leq m$, and $j \leq n$, either $\nu_i \subseteq \mu_j$ or $\nu_i \cap \mu_j = \emptyset$

A feature of our MDTD implementation, which distinguishes it from previously published algorithms, is that it computes the values of $\mathcal{F}_i$ and $\mathcal{D}_i$, where $\forall i \in [1 \ldots m], \forall j \in [1 \ldots n]$ the following hold

1. $(\mathcal{F}_i, \mathcal{D}_i)$ is a partition of $\mathcal{M}$
2. $\nu_i \subseteq \mu_j \iff \mu_j \in \mathcal{F}_i$
3. $\mu_i \cap \nu_j = \emptyset \iff \mu_i \in \mathcal{D}_j$.

The values of $\mathcal{F}_i$ and $\mathcal{D}_i$ allow the Brzozowski derivative computation to distinguish between rules (10) and (11) and assure that rule (12) never occurs. When deciding between rules (10) and (11), *i.e.* when computing $\partial_\nu \mu$, then ν corresponds to some element of $\mathcal{U}$ with index i, in which case we are sure that either $\mu \in \mathcal{F}_i$ or $\mu \in \mathcal{D}_i$, thus we know whether $\nu_i \subseteq \mu$ or $\nu_i \cap \mu = \emptyset$.

Algorithm 1 has a limitation that there may be ν_i, ν_j such that $[\![\nu_i]\!] = [\![\nu_j]\!] = \emptyset$. This limitation is due to the subset relation being sometimes undecidable; *i.e.* that the `subtype` procedure returns *dont-know*. The `subtype` procedure referred to in Algorithm 1 is the procedure provided by SETS (Section 4.1) which always returns `true`, `false`, or *dont-know*, and does not loop forever.

We believe that this undecidability issue means that any alternative `MDTD` algorithm will have the same problem. For example, consider the most primitive implementation, which simply returns a conjunction of all the possible $2^{|\mathcal{M}|}$ minterms. In such a case, most of the terms would designate the empty type.

`MDTD` has worst-case (time and space) complexity $\Omega\left(2^{|\mathcal{M}|}\right)$, if line 1.10 is taken every time through the inner loop. Any alternate algorithm must also have worst-case, exponential complexity, because in the worst case, a set of size $2^{|\mathcal{M}|}$ must be computed. We say Ω, because we are ignoring the complexity of the vacuity/disjoint checks, which only worsen the complexity. However, the complexity is quadratic if $\mathcal{M}$ is already a partition of its union.

The return value of Algorithm 1 contains a set, $\mathcal{S}$, of triples, (ν, f, d): ν is an element of the partition; f is a set of factors, each of which is a guaranteed supertype of ν, even if the `subtype` predicate is not able to detect it; and d is a set of type designators, each of which is guaranteed disjoint with ν, even if (especially if) the `disjoint` predicate is not capable of detecting the fact.

6 Sample Implementations

```
1   def isEven(x: Any): Boolean =
2     x match {
3       case y: Int => y % 2 == 0
4       case _ => false
5     }
6   val even = SSatisfies(isEven, "even")
7   val r_0 = Star(Cat(classOf[Int], classOf[String], even))
8   val r_1 = Star(Cat(classOf[Int], Star(classOf[String]), even))
9   val dfa_0 = r_0.toDfa()
10  val dfa_1 = r_1.toDfa()
11  val v0 = dfa_0.simulate(Seq(1, "hello", 2)) // returns Some(true)
12  val v1 = dfa_1.simulate(Seq(1, "hello", "world", 2, 3, 4,
13                              5, "hello", 6)) // returns Some(true)
14  val v2 = dfa_1.simulate(Seq(1.1, 1.2, 1.3)) // returns None
```

Fig. 3: Scala code for constructing r_0 and r_1 Equations (1) and (2).

As stated earlier, one of the goals of our project is to describe RTE in such a way so that it is implementable in many different programming languages. As a proof of concept, we provide open source implementations for RTE and its support libraries in Common Lisp, Scala, Clojure, and Python. The reference implementation is a Common Lisp library, `rte` [23, 21] which is available from Quicklisp [4]. The Clojure and Scala implementations, `clojure-rte` [22] and `cl-robdd-scala` [25], extend the Clojure and Scala type systems, which are already extensions of the type system of the JVM. The Python implementation,

`python-rte` [24] extends the built-in Python type system. In this article we do not delve into the details of any of these implementations. However, we invite the curious reader to download the code from the links provided.

In each of these four applications, users may specify RTEs based on fundamental types in the language, and also user defined classes. The language-level types are interfaced to RTE via SETS serving as a wrapper recognizable by the RTE implementation code.

The implementations provide Brzozowski and Thompson constructions (The Common Lisp implementation is missing the Thompson construction) to convert the RTE (expression tree) into a σDFA, including APIs for manipulating RTEs, such as inversion, intersection, union, determinization, minimization, extraction (extracting an RTE from a σDFA [32, sec 2.4.2]), vacuity checks, etc.

Figure 3 shows a small example of how RTEs can be used in a Scala program.

7 Conclusion and Perspectives

In this article we have presented a foundation sufficient for implementing RTEs in various programming languages. We have presented two algorithms (Thompson and Brzozowski) for computing the construction of σDFAs for recognizing such. Two challenges for implementing RTE in a host language, is representing and computing with types in the host language, and converting a set of overlapping types to a partition of the value space. We have proposed SETS (Section 4.1) and `MDTD` (Section 5.5) as solutions to these challenges, along with sample implementations in Common Lisp, Clojure, Scala, and Python.

Scala is not generally considered a dynamic language, but it does exhibit some dynamic features: REPL, run-time reflection via its reflection [10] library or even the lower-level Java reflection. The RTE library adds sort of a philosophical flavor of dynamic typing into what is generally viewed as a statically typed language. We say *generally viewed* because, a Scala program is indeed allowed to inquire at run-time about the type of an object and invoke run-time logic as a function of subtypeness via `isAssignableFrom`. However, it is a matter of ongoing debate whether the RTE implementation in Scala is useful or culturally acceptable given tendency among Scala programmers to avoid run-time type checking. An important strength of our type system is that many questions are answered with three-way logic. For types ν and μ, we distinguish $\nu \subseteq \mu$ (ν is proven to be a subtype of μ), $\nu \not\subseteq \mu$ (ν is proven NOT to be a subtype of μ), and `dont-know` (we were unable to prove or disprove $\nu \subseteq \mu$). This three-way logic extends into the question of habitation of rational languages. For example, given a σDFA it might be that every computation path from q_0 to a final state passes through at least one indeterminate transition–not provably satisfiable and not provably non-satisfiable. In this case we cannot determine whether the language of the σDFA is inhabited.

References

1. Seqexp: regular expressions for sequences (2014), https://github.com/cgrand/seqexp
2. Ansi: American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999) (1994)
3. Baker, H.G.: A Decision Procedure for Common Lisp's SUBTYPEP Predicate. Lisp and Symbolic Computation **5**(3), 157–190 (1992), http://dblp.uni-trier.de/db/journals/lisp/lisp5.html#Baker92a
4. Beane, Z.: Quicklisp beta (2015), https://www.quicklisp.org/beta
5. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for Clojure (2018)
6. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (xml) 1.0 (fifth edition). W3C Recommendation (2008), available at http://www.w3.org/TR/REC-xml/
7. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **35**, 677–691 (August 1986)
8. Brzozowski, J.A.: Derivatives of Regular Expressions. J. ACM **11**(4), 481–494 (Oct 1964). https://doi.org/10.1145/321239.321249, http://doi.acm.org/10.1145/321239.321249
9. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Computer Aided Verification, 29th International Conference (CAV'17). Springer (July 2017), https://www.microsoft.com/en-us/research/publication/power-symbolic-automata-transducers-invited-tutorial/
10. EPFL: Scala Reflection Library 2.12.0 (2016), https://www.scala-lang.org/api/2.12.0/scala-reflect/scala/reflect/runtime/index.html
11. Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edn. (2014)
12. Grigore, R.: Java generics are turing complete. CoRR **abs/1605.05274** (2016), http://arxiv.org/abs/1605.05274
13. Heikkilä, M.: Malli, Metosin (2022), https://github.com/metosin/malli
14. Hickey, R.: The Clojure Programming Language. In: Proceedings of the 2008 symposium on Dynamic languages. p. 1. ACM (2008)
15. Hickey, R.: A History of Clojure. Proc. ACM Program. Lang. **4**(HOPL) (Jun 2020). https://doi.org/10.1145/3386321, https://doi.org/10.1145/3386321
16. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
17. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular Expression Types for XML. ACM Trans. Program. Lang. Syst. **27**(1), 46–90 (Jan 2005). https://doi.org/10.1145/1053468.1053470
18. Keil, M., Thiemann, P.: Symbolic solving of extended regular expression inequalities (2014). https://doi.org/10.48550/ARXIV.1410.3227, https://arxiv.org/abs/1410.3227
19. Kennedy, A., Pierce, B.C.: On decidability of nominal subtyping with variance. In: International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD) (January 2007), https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/
20. Miller, A.: Spec Guide (2022), https://clojure.org/guides/spec

21. Newton, J.: Representing and Computing with Types in Dynamically Typed Languages. Ph.D. thesis, Sorbonne University (Nov 2018)
22. Newton, J.: Regular Type Expressions for Clojure (2024), github.com/jimka2001/clojure-rte
23. Newton, J.: Regular Type Expressions for Common Lisp (2024), github.com/jimka2001/cl-rte
24. Newton, J.: Regular Type Expressions for Python (2024), github.com/jimka2001/python-rte
25. Newton, J.: Regular Type Expressions for Scala (2024), github.com/jimka2001/scala-rte
26. Newton, J., Demaille, A., Verna, D.: Type-Checking of Heterogeneous Sequences in Common Lisp. In: European Lisp Symposium. Kraków, Poland (May 2016)
27. Newton, J., Pommellet, A.: A portable, simple, embeddable type system. In: European Lisp Symposium. Online, Everywhere (May 2021)
28. Newton, J., Verna, D.: Strategies for typecase optimization. In: European Lisp Symposium. Marbella, Spain (Apr 2018)
29. Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The Scala language specification (2004)
30. Odersky, M., Zenger, M.: Scalable component abstractions. In: Sigplan Notices - SIGPLAN. vol. 40, pp. 41–57 (10 2005). https://doi.org/10.1145/1103845.1094815
31. Owens, S., Reppy, J., Turon, A.: Regular-expression Derivatives Re-examined. J. Funct. Program. **19**(2), 173–190 (Mar 2009)
32. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, USA (2009)
33. Valais, L., Newton, J., Verna, D.: Implementing baker's `SUBTYPEP` decision procedure. In: 12th European Lisp Symposium. Genova, Italy (Apr 2019)

1 Correctness of $\partial_\nu(\neg r) = \neg\partial_\nu r$

To avoid confusion we consider (5) in more detail. Keil and Thiemann [18] divide the derivative into two operators, a *positive* and a *negative* derivative, whereas Owens [31] does not. There is a subtle issue pointed out by Keil and Thiemann [18]: How do we know $\partial_\nu(\neg r) = \neg\partial_\nu r$?

$$\begin{aligned}
\partial_\nu r &= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \in [\![r]\!]\} \\
\partial_\nu(\neg r) &= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \in [\![\neg r]\!]\} \\
&= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \notin [\![r]\!]\} \qquad (14) \\
\neg\partial_\nu r &= \Sigma \setminus \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \in [\![r]\!]\} \\
&= \{s \in \Sigma^* \mid \forall h \in [\![\nu]\!], h :: s \notin [\![r]\!]\} \qquad (15)
\end{aligned}$$

We partition ν into $\{\nu_1, \nu_2\}$ with $[\![\nu_1]\!] = \{g \in \nu \mid g :: s \in [\![r]\!]\}$ and $[\![\nu_2]\!] = \{g \in \nu \mid g :: s \notin [\![r]\!]\}$. If $[\![\nu_1]\!] \neq \emptyset$, then it follows from (14) and (15) that $\neg\partial_\nu r \subseteq \partial_\nu(\neg r)$. However, is it true that $\partial_\nu(\neg r) \subseteq \neg\partial_\nu r$?

To answer that question, suppose $s \in [\![\partial_\nu(\neg r)]\!]$ but $s \notin [\![\neg\partial_\nu r]\!]$; equivalently $s \in [\![\partial_\nu(\neg r)]\!]$ and $s \in [\![\partial_\nu r]\!]$. This means

$$\begin{aligned}
s \in [\![\partial_\nu(\neg r)]\!] &\implies \exists h_1 \in \nu, h_1 :: s \notin [\![r]\!] \\
&\implies h_1 \in [\![\nu_1]\!] \\
&\implies [\![\nu_1]\!] \neq \emptyset \\
s \in [\![\partial_\nu r]\!] &\implies \exists h_2 \in \nu, h_2 :: s \in [\![r]\!] \\
&\implies h_2 \in [\![\nu_2]\!] \\
&\implies [\![\nu_2]\!] \neq \emptyset
\end{aligned}$$

To assure that $\partial_\nu(\neg r) \subseteq \neg\partial_\nu r$ and thus $\partial_\nu(\neg r) = \neg\partial_\nu r$, it suffices to assure by construction that ν_2 and ν_2 cannot both be inhabited types. *I.e.*, we must always decompose types mentioned in r into types which are disjoint from each other, and never partially overlapping types mentioned in r. The MDTD procedure (Section 5.5) assures this kind of straddling is avoided.

2 Algorithm of σDFA Construction

This algorithm illustrates the verbal explanation found in Section 5.2.

Input: r : an RTE
Output: σDFA

```
2.1  begin
2.2  |  q_0 ← new State(r), T ← (), Q ← {q_0}
2.3  |  for q_1 ∈ Q                                  // iterate over changing set
2.4  |   do
2.5  |  |  r ← q_1.rte
2.6  |  |  for (ν, f, d) ∈ MDTD(1^st(r)) do
2.7  |  |  |  r' ← ∂_ν r                          // derivative and canonicalize
2.8  |  |  |  if r' = ∅ then
2.9  |  |  |  |  continue                  // avoid unsatisfiable transition
2.10 |  |  |  else if ∃ q_2 ∈ Q such that q_2.rte = r' then
2.11 |  |  |  |  T ← (q_1, ν, q_2) :: T    // transition to pre-existing state
2.12 |  |  |  else
2.13 |  |  |  |  q_2 ← new State(d)
2.14 |  |  |  |  T ← (q_1, ν, q_2) :: T          // transition to a new state
2.15 |  |  |  |  Q ← q_2 :: Q
2.16 |  F ← {q ∈ Q | () ∈ [[q.rte]]}    // compute final states, cf Owens [31]
2.17 |  return (Q, q_0, F, T)
```

Algorithm 2: Compute σDFA by Brzozowski derivative