Type-Checking 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 over finite alphabets was conceived for characterizing patterns in so-called regular languages, but does not exactly meet this challenge, because the set of potential elements of the sequences 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 and manipulating symbolic finite automata, a generalization of classical finite automata, using a portable, simple, 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, the system defines types using "a Simple Embeddable Type System" from Newton and Pommellet [28]. For the second challenge, the we use Brzozowski style derivative-based construction, and we solve the challenge of overlapping types using Maximal Disjoint Type Decomposition (MDTD). The paper also briefly discusses another construction algorithm Thompson.

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 Motivation

Statically typed languages, such as Java or C++, support sequences of fixed types such as `Array[String]`, e.g., `["ab","cd","xyz"]`, or `List[Double]`, e.g., `List(1.0,1.1,1.2)`. In a language whose type system forms a type lattice [17], it is possible to declare an `Array[String|Double]` designating a sequence such as `["ab",1.1,1.2,"cd","xyz"]`, with each element either `String` or `Double`. More general still, languages such as Python [34] and Common Lisp [1], support sequences where any element may be an object of any inhabited type whatsoever.

It is rare that *heterogeneous sequences* contain values completely random in type. Rather, in a given program, the element types generally follow an implicit pattern which is firm in the mind of the programmer and documented in the comments. The programmer writes code assuming elements to be a certain type, or writes ad-hoc code to check the contents of the sequences at run-time.

Scala [30, 31], a JVM [10] (Java Virtual Machine) based language, allows the program to manipulate sequences of arbitrary types such as those coming from JSON [33]. The code normally uses pattern matching to implement `typecase` logic based on dynamic type meta-data managed by the JVM. Clojure [13, 14], also a JVM language, likewise supports such sequences of arbitrary type. In Clojure the programmer usually uses a tool called `spec` [21] to implement program logic to detect type patterns, or resorts to a set of type predicates provided by the base language such as `int?`, `number?`, `string?`, etc.

What is otherwise lacking from many dynamic languages (or statically typed langauges with sufficiently flexible reflection), and which we address in this article, is a mechanism for the programmer to declare the expected type patterns, allowing the sequences to be efficiently type-checked at run-time, and thereafter to allow application code the safety of making simplifying assumptions about the data in question.

1.2 Illustrative Examples

For example,

$$r_0 = (\mathit{int} \cdot \mathit{string} \cdot \mathit{even?})^* \tag{1}$$

$$r_1 = (\mathit{int} \cdot \mathit{string}^* \cdot \mathit{even?})^* \tag{2}$$

$$r_2 = (\mathit{int} \cdot \mathit{string} \cdot \mathit{string}^* \cdot \mathit{even?})^* \,. \tag{3}$$

Such expressions are called a *rational type expressions* [27]. The syntax should be intuitive to anyone already familiar with regular expressions. To avoid limiting our application to a single programming language, we define *type* simply and portably [28]. The RTE, r_0, 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. The second, r_1, is similar to r_0, but the string is allowed to occur 0 or more times. Finally, r_2, requires that the string occur at least once. *E.g.*, the sequence (11, `"a"`, 12, 13, `"a"`, `"b"`, 14)

matches r_1 and r_2, but not r_0; $(11, \texttt{"a"}, 12, \quad 13, \texttt{"a"}, 14)$ matches all of them; and $(11.5, 12.6)$ matches none.

A finite automaton is used to efficiently decide membership of a rational language. Analogous to classical finite automata theory [15], RTEs correspond to *symbolic* finite automata [8] (σDFA) over a possibly infinite alphabet—labeled by subsets of the alphabet, denoted Σ. In our case, Σ is the set of all values representable in a given 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. In general construction of the finite automata

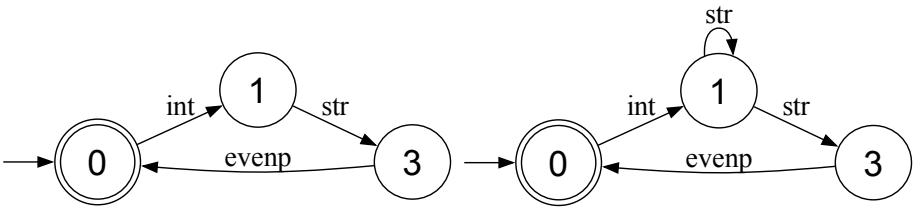


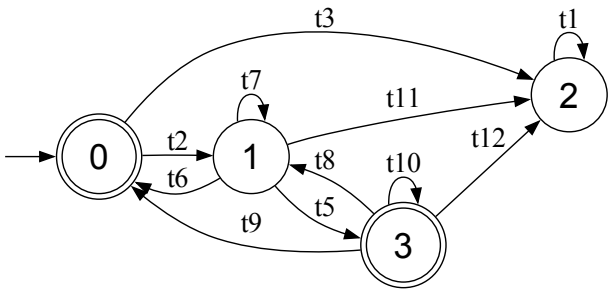
Fig. 1: σDFA for RTEs: $r_0 = (\mathit{int} \cdot \mathit{string} \cdot \mathit{even}?)^*$ [(1) left] and non-deterministic automaton for $r_2 = (\mathit{int} \cdot \mathit{string} \cdot \mathit{string}^* \cdot \mathit{even}?)^*$ [(3) right]

(classical or symbolic) is expensive to compute. This expense can be mitigated by pre-computing at compile-time, if the programming language allows application code to be run at compile-time (such as is the case with Common Lisp [1]), or at least by memoizing the constructed automata so that they can be re-used. 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 representation of the RTE itself. This is important because there exist arbitrarily large expression trees which *reduce* to the same, *small* RTE.

Since we wish to perform operations such as, negation and intersection, we chose to work with deterministic automata. Deterministic finite automata (symbolic and otherwise) not only allow us to decide membership in a rational language, but also to perform reasoning on the languages themselves: checks such as disjoint, subset, vacuity/habitation, etc.

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. If ν and μ are types, then we can prove ν and μ equivalent by proving $(\nu \subseteq \mu) \wedge (\mu \subseteq \nu)$. ν is provably vacuous, if we can prove $\nu \subseteq \emptyset$. ν and μ are provably disjoint, if we can prove $\nu \cap \mu \subseteq \emptyset$.



$t_1 = \Sigma$
$t_2 = int$
$t_3 = \overline{int}$
$t_4 = string$
$t_5 = string \wedge even?$
$t_6 = \overline{string} \wedge even?$
$t_7 = string \wedge \overline{even?}$
$t_8 = (int \wedge \overline{even?}) \vee (string \wedge \overline{even?})$
$t_9 = \overline{int} \wedge \overline{string} \wedge even?$
$t_{10} = (int \wedge even?) \vee (string \wedge even?)$
$t_{11} = \overline{string} \wedge \overline{even?}$
$t_{12} = \overline{int} \wedge \overline{string} \wedge \overline{even?}$

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

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

As an example of a type which causes difficulty for the subtype decision procedure consider the type *even*?. The type, *even*?, is a so-called *satisfies* type, which is constructed using a predicate. The predicate 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 transitions leaving 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: $t_5 \cup t_6 \cup t_7 \cup t_{11} = \Sigma$.

Undecidable subtype relations, and consequently unsatisfiable transitions, 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 transitions to and from ③ because its label, t_5 is not satisfiable. However, we cannot determine programmatically that this elimination is possible, without a sufficiently intelligent oracle.

It may occur (albeit not in this example) that certain transitions reference equivalent types; *e.g.*. $t_4 = string$ and $t_7 = string \wedge \overline{even?}$. These equivalent types, may prevent some σDFAs from being optimally minimized. However, we

are guaranteed by construction that two such transitions never appear leaving the same state, else the automaton would fail to be deterministic.

Minimization on σDFAs is an open question which requires more research. We will address this topic no further in the current article. We only mention it here because the impossibility of minimization can lead to sub-optimal σDFA, as well as compile-time and run-time inefficiencies.

Unsatisfiable but indeterminate transitions are common; *e.g.*. $t_9 = \overline{int} \wedge \overline{string} \wedge even? = \emptyset$, because $\overline{int}$ and $even?$ are disjoint. Such transitions are inevitable, ultimately because the subtype relation is not decidable in many cases. In this case, the fact that $even? \subset int$ is not known.

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. 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.

1.3 Our Contribution

We present a technique for recognizing certain heterogeneous sequences based on the types of their constituent elements. The technique involves declaratively describing such sequences using so-called RTEs. The RTEs are used to construct symbolic finite automata [8], which are then used to validate and reason about finite sequences whose elements are taken from an infinite set of values supported in the programming language.

Our contributions in this article are as follows. We

1. Adapt the theoretical description of D'Antoni and Veanes [8] to practical programming languages: Python, Scala, and Clojure, providing sample implementations in Common Lisp, Scala, Clojure, and Python. (Section 6)
2. Contrast two σDFA constructions: Brzozowski and Thompson, summarizing important implementation details, emphasizing algorithmic differences with respect to classical finite automata on finite alphabets.
3. Provide a simple and comprehensible algorithm for the maximal disjoint type decomposition, `MDTD`, which avoids, by construction, the problem of undecidability of subtyping.

1.4 Previous Work

In [22], Newton used deterministic finite automata (DFA) to recognize certain heterogeneous sequences based on the types of values therein. The work introduced the concept of RTE as a means of declaratively specifying such sequences.

[22] addressed only Common Lisp sequences. The current work, generalizes the RTEs to a wider range of programming languages.

The `clojure-rte` implementation is reminiscent of the Clojure Spec library [21] and `metosin` [12]. Although Hickey [14, 13] mentions Spec's existence, we have otherwise found no 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.* [20]. An NFA-based procedure (presumably using backtracking) would have at least polynomial complexity, whereas our approach offers linear complexity.

Christophe Grande has authored a regular pattern matching library called `seqexp`. The implementation is in Clojure [14, 13]. We have not found any peer reviewed articles referring to this library and thus refer the reader to the implementation at `https://github.com/cgrand/seqexp`. According to an interview with Grande, the `seqexp` does not use a finite automata approach, suggesting that the size of resulting code would violate the JVM [10] limitation of function size susceptible to optimization.

The Brzozowski derivative and an algorithm to compute it was first presented in 1964 by Janusz Brzozowski [7]. While Brzozowski's result was applied to digital circuits, Scott Owens *et al.* [32] presented a *modern* version applied to regular pattern recognition for sequences of characters. 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 [8] argue that the generalization retains many of the good properties of their finite-alphabet counterparts. The reader may imagine, many intuitions about finite automata apply to σDFA, but some do not. In the same work D'Antoni and Veanes discuss a decomposition of types which he refers to as *Minterms(...)* which they describe as *the set of maximal satisfiable 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 [11] addresses subtype decidability in Java [10]: equating it to the halting problem. Kennedy and Pierce [19] identify several type system restrictions which make the question decidable. They investigate the effect of restricting these capabilities in different ways in Java [10], Scala [30, 31], C#, and .NET Intermediate Language. Curiously, [19] from Microsoft Research gives citations for Java and Scala, but not for C# and .NET Intermediate Language. Perhaps they believe the reader is already intimately familiar with C# and .NET IL.

In Scala and in Clojure, the method `isAssignableFrom` (which is provided by the JVM [10]) attempts to compute the subtype relation, returning *true* if the subtype relation can be proven, and *false* otherwise. Thus `isAssignableFrom` conflates *cannot compute* with *computed to be false*. The Clojure language offers an `isa?` predicate based on `isAssignableFrom`.

In Common Lisp [1], `SUBTYPEP` can be compute intensive in general. Even in cases where the subtype relation is decidable, a Common Lisp implementation is allowed to return *dont-know* when estimated *too compute intensive*. [2, 36] Common Lisp is also 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 sometimes return *dont-know*, but the type-membership check, `typep`, must return `true` or `false`, which would be impossible in certain membership checks for functions. *E.g.*, if $f : A \to B$ and if $A \to B \subseteq X \to Y$ returns *dont-know*, then $f \in X \to Y$ cannot return `true` or `false`. SETS (Section 4.1) circumvents this problem by avoiding explicit support for function types, other than that already supported in the host language.

Bonnaire [4] presents *typed Clojure*, without rigorously defining a Clojure *type*, nor anything as ambitious as Common Lisp's `SUBTYPEP`.

Hosoya, Vouillon and Pierce [16] defined what they called regular expression types and developed a programming language, XDuce, around them. XDuce allows static XML types to be defined recursively and hierarchically to describe the structure of the XML document in question. The goal, as they explain, is allow the programmer to write programs which consume and manipulate XML [5] documents allowing the type checker to assure that the programmatic expressions are type correct according to the XML schema, DTD, XML-Schema *etc.*.

A notable distinction of the RTE implementation as opposed to the XDuce language is that our implementation illustrates adding such type checking ability to an existing type system and suggests that such extensions are feasible in other existing dynamic or reflective languages.

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,origin,target,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* q *as the* origin *of the transition,* r *as the* target*, and* ν *as the label.* $\triangle$

Definition 2 (σDFA, complete, deterministic, partition). *Let* $A = (\Sigma, Q, q_0, F, T)$ *be a symbolic finite automaton. If for each* $q \in Q$*, the*

set labels of transitions exiting q is a partition of Σ, then A is called a complete, deterministic, symbolic, finite automaton*, or simply a σDFA.*
A partition *is set of mutually disjoint, possibly empty subsets of Σ covering Σ.*

A transition $q \xrightarrow{\nu} r$ is called *satisfiable* if ν is inhabited ($\nu \neq \emptyset$). It is called *non-satisfiable* if ν is empty ($\nu = \emptyset$). It is called *indeterminate* if it cannot be determined whether $\nu = \emptyset$. The challenge of determining programmatically whether a transition is satisfiable is addressed in Section 1.4.

3 Rational Expressions

We accept the following definitions which are taken directly from classical finite automata theory. A *sequence* of length $n \geq 0$ is a function mapping $\{0, 1, \ldots, n-1\} \to \Sigma$. A *language* is any set of finite-length sequences. The set of all finite-length sequences is called Σ^*. The symbol () represents the empty sequence (sequence of zero-length). The language, ε, contains only (); $\varepsilon = \{()\}$. Two sequences, s, t can be concatenated in the intuitive fashion to form 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*, $x :: s = (x) \cdot s$, thus extending the sequence to length $n + 1$. Finally, we take the normal definition of the Kleene star of a language, L^*, the set of all possible finite concatenations of zero or more sequences from L.

Given a σDFA, there is a unique subset of Σ^* which is *recognized* by the σDFA. This set is called the *language* of the σDFA. As in the classical finite automata case, 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 3 (rational type expression, RTE). *A* rational type expression *or RTE is an expression (tree) which represents (generates) a language on Σ. Let f and g be RTEs, generating languages F and G, and $\nu \subseteq \Sigma$, then the following recursively defines all RTEs.*

RTE	***Generated Language***	***RTE***	***Generated Language***
$\emptyset$	$\emptyset$	$f \cdot g$	$F \cdot G$
ε	$\{()\}$	$f \vee g$	$F \cup G$
		$f \wedge g$	$F \cap G$
ν	$\{(a) \mid a \in \nu\}$	$\neg f$	$\Sigma^* \setminus F$
Σ	$\{(a) \mid a \in \Sigma\}$	f^*	F^*

If r is an RTE, we take $[\![r]\!]$ to denote the language generated by r. △

4 Programming Language Supporting RTEs

Even if finite automata theory is more general, allowing us to reason about arbitrary, free monoids, we will restrict our discussion to objects and sequences

representable in a given programming language. Σ will be the set of values expressible in some programming language, and Σ^* denotes the set of all sequences whose values come from Σ. Suppose further that the programming language supports a set of built-in types and user-defined types, and a set of symbols such as `int` and `string` which allow an application program to perform run-time type membership checks. We call the set of all such symbols Υ_0, and 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

Different programming languages make different assumptions about types. Newton and Pommellet [28] presented a *Simple Embeddable Type System* (SETS) which we use here. The principle is that we keep the type system simple enough to implement in a wide range of programming languages, and that types already expressible in the particular language are accepted as atomic types in SETS. A formal definition of SETS can be found in [28], but we summarize it here.

Definition 4 (type). *A type is any set of values, i.e. any subset of Σ.* △

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 by, Υ, and is recursively defined as follows.

1. **Hosted types**: $\Upsilon_0 \subseteq \Upsilon$.
2. **Terminal types**: $\Sigma, \emptyset \in \Upsilon$ with $[\![\Sigma]\!] = \Sigma$, and $[\![\emptyset]\!] = \emptyset$.
3. **Singleton types**: $\forall a \in \Sigma$, $\{a\} \in \Upsilon$, with $[\![\{a\}]\!] = \{a\}$.
4. **Predicates**: for any decidable function $f : \Sigma \to \{true, false\}$, there is a symbol $\mathfrak{Sat}(f) \in \Upsilon$ with $[\![\mathfrak{Sat}(f)]\!] = \{x \in \Sigma \mid f(x) = true\}$.
5. **Union**: if $\nu_1, \nu_2 \in \Upsilon$, then $\nu_1 \cup \nu_2 \in \Upsilon$ with $[\![\nu_1 \cup \nu_2]\!] = [\![\nu_1]\!] \cup [\![\nu_2]\!]$.
6. **Intersection**: if $\nu_1, \nu_2 \in \Upsilon$, then $\nu_1 \cap \nu_2 \in \Upsilon$ with $[\![\nu_1 \cap \nu_2]\!] = [\![\nu_1]\!] \cap [\![\nu_2]\!]$.
7. **Complement**: if $\nu \in \Upsilon$, then $\overline{\nu} \in \Upsilon$ with $[\![\overline{\nu}]\!] = \{x \in \Sigma \mid x \notin [\![\nu]\!]\}$.

We distinguish a singleton type, $\{a\}$, from a singleton sequence (a). Given $a \in \Sigma$, the singleton type $\{a\}$ is the set containing only a, while the singleton sequence (a) is a sequence of length 1 whose first and only element is a. Finally, if $\nu \in \Upsilon$, then the RTE ν is the set of all singleton sequences (a) where $a \in \Sigma$.

With a clever choice of f, $\mathfrak{Sat}(f)$ may designate the same set by other composed types in SETS, albeit with less reasoning power. For example, given two predicates f and g it is not possible in general to determine whether $\mathfrak{Sat}(f) \subseteq \mathfrak{Sat}(g)$ or whether $\overline{\mathfrak{Sat}(f)} \cap \overline{\mathfrak{Sat}(g)}$ is inhabited.

A programming language specific API must implement type designators and $\in$, $\subseteq$ procedures. The membership predicate, $\in$, must be decidable and return always `true`, `false`. An `inhabited` predicate can be implemented as $\nu \subseteq \emptyset$. However, because the subtype relation is sometimes undecidable (or may be

excessively compute intensive) $\subseteq$ and consequently, the `inhabited` predicate must always return, or *dont-know* in a programming language specific way. To reiterate, we require that the $\subseteq$ predicate in SETS never loop forever—either it returns `true`, `false` indicating that the subtype relation has been proven or disproven, or it gives up and returns *dont-know*.

5 Construction of σDFA

In this section we describe σDFA construction. The Brzozowski algorithm creates a deterministic automaton by construction. Some other construction algorithms, such as Thompson (Section 5.6), create a non-deterministic automaton which must thereafter be determinized.

5.1 Brzozowski Derivative

In this section we present a generalization of the Brzozowski derivative which is the principal tool needed for such the Brzozowski σDFA construction.

While Brzozowski/Owens [7, 32] 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 5 (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): $r_1 = (\mathit{int} \cdot \mathit{string}^* \cdot \mathit{even}?)^*$, and let S be the language of r_1. Each sequence is either () or begins with an *int*.

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

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

because if we take the subset of S containing sequences starting with an *int*, which is $S \setminus \{()\}$, then strip off the head (a *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 $(\mathit{int} \cdot \mathit{string}^* \cdot \mathit{even}?)$.

We wish to 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 [32] succinctly stated.

$$\partial_\nu\, \emptyset = \partial_\nu\, \varepsilon = \emptyset \tag{4}$$

$$\partial_\nu\, (\neg r) = \neg \partial_\nu\, r \tag{5}$$

$$\partial_\nu\, (r^*) = \partial_\nu\, r \cdot r^* \tag{6}$$

$$\partial_\nu\, (r \vee s) = \partial_\nu\, r \ \vee\ \partial_\nu\, s \tag{7}$$

$$\partial_\nu\, (r \wedge s) = \partial_\nu\, r \ \wedge\ \partial_\nu\, s \tag{8}$$

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

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

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

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

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 [32] does not. There is a subtle issue pointed out by Keil and Thiemann [18]: How do we know $\partial_\nu (\neg r) = \neg \partial_\nu r$?

$$\begin{aligned}
\partial_\nu r &= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \in [\![r]\!]\} \\
\partial_\nu (\neg r) &= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \in [\![\neg r]\!]\} \\
&= \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \notin [\![r]\!]\} \qquad (13) \\
\neg \partial_\nu r &= \Sigma \setminus \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \in [\![r]\!]\} \\
&= \{s \in \Sigma^* \mid \forall h \in [\![\nu]\!], h :: s \notin [\![r]\!]\} \qquad (14)
\end{aligned}$$

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 (13) and (14) 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.

5.2 Dealing with Overlapping Types

The Brzozowski construction for finite alphabets does not encounter the problem of overlapping types, because every letter in the alphabet is distinct from all others. On the contrary, in our case, the *letters* under considerations are types which correspond to subsets of Σ. Two types might be related by a subtype relation, or a disjoint relation. We have extended the Brzozowski method to accommodate intersecting types. 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 [32] mentions. We have replaced a rule from the treatment from Owens, which was

$\partial_a\, b = \varepsilon$ whenever $a \neq b$ with a generalization. This rule as Owens states, does not hold when regular expressions are generalized to RTEs. Our generalization is the introduction of Equations (10), (11), and (12): 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 determine whether $\nu \subseteq \mu$ or whether $\nu \cap \mu = \emptyset$. This poses a challenge: the subtype relation, and thus the disjoint relation, is sometimes undecidable. One might naively think that if we deconstruct 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, there is no guarantee that the subtype procedure in the host language nor the subtype procedure in SETS (Section 4.1) is sufficiently complete to determine all subtype relations necessary.

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 that either $\nu \subseteq \mu$ or $\nu \cap \mu = \emptyset$; moreover, we know which of the two holds.

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.

5.3 Constructing States and Transitions

Algorithm 1 specifies the construction of a σDFA from an RTE. The algorithm can be summarized as follows. The initial state, q_0 represents the given RTE, r. Each subsequent state represents some n^{th} derivative, $\partial_\nu^{[n]} r$. Brzozowski [7, 32] argues that the set of all such derivatives is finite.

We wish to compute the σDFA in Figure 2 corresponding to $r_1 = (\mathit{int} \cdot \mathit{string}^* \cdot \mathit{even}?)^*$. Step 1: start with the initial state $q_0 = \textcircled{0}$ corresponding to r_1. Step 2: compute `MDTD` given the set of types which may be the type of the first element of a sequence in the language of r_1. The only such type is *int*; `MDTD` returns the partition $\Sigma = \mathit{int} \cup \overline{\mathit{int}}$. Step 3: compute $\{q_1, q_2\} = \{\partial_\nu\, r_1 \mid \nu \in \{\mathit{int}, \overline{\mathit{int}}\}$ by applying rules (6), (9), (10), and (11):

$$q_1 = \partial_{\mathit{int}}\, r_1 = \mathit{string}^* \cdot \mathit{even}? \cdot (\mathit{int} \cdot \mathit{string}^* \cdot \mathit{even}?)^* \tag{15}$$

$$q_2 = \partial_{\overline{\mathit{int}}}\ r_1 = \emptyset\,. \tag{16}$$

During the computation of q_1 and q_2, we encounter the computation of $\partial_{\mathit{int}}\, \mathit{int}$ and $\partial_{\overline{\mathit{int}}}\, \mathit{int}$. As explained in Section 5.2, $\partial_{\mathit{int}}\, \mathit{int} = \varepsilon$, because $\mathit{int} \subseteq \mathit{int}$, rule (10); and $\partial_{\overline{\mathit{int}}}\, \mathit{int} = \emptyset$, because $\overline{\mathit{int}} \subseteq \mathit{int} = \emptyset$, rule (11).

Input: r : an RTE
Output: Components of a σDFA as in Definition 2.

begin
  $q_0 \leftarrow$ new $State(r)$; $T \leftarrow ()$; $Q \leftarrow \{q_0\}$
  $W \leftarrow \{q_0\}$ `// working list states`
  while $W \neq \emptyset$ **do**
    $q_1 \leftarrow$ any element from W
    $W \leftarrow W \setminus \{q_1\}$
    for $\nu \in MDTD(1^{st}(q_1.expression))$ **do**
      $d \leftarrow \partial_\nu\,(q_1.expression)$ `// reduced to canonical form`
      if $d \neq \emptyset$ `// if not the empty language`
      then
        if $\exists\ q_2 \in Q$ *such that* $q_2.expression = d$ **then**
          $T \leftarrow\ (q_1, \nu, q_2) :: T$ `// transition`
        else
          $q_2 \leftarrow$ new $State(d)$
          $T \leftarrow\ (q_1, \nu, q_2) :: T$
          $W \leftarrow q_2 :: W$
          $Q \leftarrow q_2 :: Q$
  $F \leftarrow \{q \in Q \mid () \in [\![q.expression]\!]\}$ `// Explained in Owens [32]`
  return (Q, q_0, F, T)

Algorithm 1: Compute DFA by Brzozowski derivative

Having computed q_1 and q_2, we create two new states, ① and ②, and add two transitions, one for each derivative: ⓪ $\xrightarrow{int}$ ① and ⓪ $\xrightarrow{\overline{int}}$ ②. The two values of ν correspond respectively to the labels of the two transitions.

Type designator	**Supertypes**	**Disjoint types**
$string \wedge even?$	$\Sigma, string, even?$	$\emptyset, \overline{string}, \overline{even?}$
$string \wedge \overline{even?}$	$\Sigma, string, \overline{even?}$	$\emptyset, \overline{string}, even?$
$\overline{string} \wedge even?$	$\Sigma, \overline{string}, even?$	$\emptyset, string, \overline{even?}$
$\overline{string} \wedge \overline{even?}$	$\Sigma, \overline{string}, \overline{even?}$	$\emptyset, string, even?$

Fig. 3: `MDTD` computation for types: $\{string, even?\}$.

We now compute the derivatives of q_1 and later q_2. All of the derivatives of $\partial_\nu\, q_1$ are $\emptyset$ except those for which ν designates a first element of some sequence in the language of q_1: *i.e.* $\nu \in \{string, even?\}$. If we use the `MDTD` function in Algorithm 2, we obtain the partition $\Pi = \{string \wedge even?,\ \overline{string} \wedge even?,\ string \wedge \overline{even?},\ \overline{string} \wedge \overline{even?}\}$, and the additional information in Figure 3, which provides the subtype and disjoint relations needed to apply reduction rules (10) and (11), when computing $\{\partial_\nu\, q_1 \mid \nu \in \Pi\}$.

When $\nu = t_5 = \mathit{string} \wedge \mathit{even?}$, we obtain a new RTE and state, which we call $q_3 = ③$:

$q_3 = \partial_{t_5}\, q_1 = \mathit{string}^* \cdot \mathit{even?} \cdot (\mathit{int} \cdot \mathit{string}^* \cdot \mathit{even?})^* \vee (\mathit{int} \cdot \mathit{string}^* \cdot \mathit{even?})^*$.

Computing the remaining derivatives of q_1 we find duplicates of derivatives computed earlier. Thus we can construct the four transitions from state q_1: ① $\xrightarrow{t_5}$ ③, ① $\xrightarrow{t_6}$ ⓪, ① $\xrightarrow{t_7}$ ①, and ① $\xrightarrow{t_{11}}$ ②; because $\partial_{t_5}\, q_1 = q_3$, $\partial_{t_6}\, q_1 = r_1$, $\partial_{t_7}\, q_1 = q_1$, and $\partial_{t_{11}}\, q_1 = q_2$. In like manner we compute all the derivatives of q_2 and q_3, but in doing so we find no new RTEs: $\partial_{t_1}\, q_2 = q_2$, $\partial_{t_8}\, q_3 = q_1$, $\partial_{t_9}\, q_3 = r_1$, $\partial_{t_{10}}\, q_3 = q_3$, and $\partial_{t_{13}}\, q_3 = q_2$. These derivatives complete the transitions in Figure 2: ② $\xrightarrow{t_1}$ ②, ③ $\xrightarrow{t_8}$ ①, ③ $\xrightarrow{t_9}$ ⓪, ③ $\xrightarrow{t_{10}}$ ③, and ③ $\xrightarrow{t_{13}}$ ②.

Finally, we decide which states are *accepting*. A state associated with RTE r is accepting if $() \in [\![r]\!]$. Thus the accepting states are q_0 and q_3; $() \in [\![r_1]\!]$, $() \notin [\![q_1]\!]$, $() \notin [\![q_2]\!]$, and $() \in [\![q_3]\!]$. Owens [32], provides a simple algorithm for deciding whether $\varepsilon \subseteq [\![r]\!]$; we omit the algorithm in this article.

5.4 Avoiding Redundant Run-time Type Checking

It should be noted that cases such as state ① in Figure 2 whose transitions were computed in Section 5.3 are a common occurrence. The labels of the transitions exiting state ① have redundant leaf-level type designators. *E.g.*, $t_5 = \mathit{string} \wedge \mathit{even?}$, $t_6 = \overline{\mathit{string}} \wedge \mathit{even?}$, $t_7 = \mathit{string} \wedge \overline{\mathit{even?}}$, and $t_{11} = \overline{\mathit{string}} \wedge \overline{\mathit{even?}}$ all involve both *string* and *even?*. Additionally, some labels such as $t_8 = (\mathit{int} \wedge \overline{\mathit{even?}}) \vee (\mathit{string} \wedge \overline{\mathit{even?}})$ 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 [6, 29], 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 [22] 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, 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 2 computes a pair, $(\mathcal{U}, \mathcal{S})$, where $\mathcal{S}$ is meta data described below, and $\mathcal{U}$ a new set of types $\mathcal{U} = \{\nu_1, \nu_2, \ldots, \nu_m\}$, which has the following properties.

Input: $\mathcal{M}$: a set of type designators
Output: $(\mathcal{U}, \mathcal{S})$: a pair as described in Section 5.5.

```
2.1  begin
2.2      S ← {(⊤, {⊤}, {⊥})}                          // working list of triples
2.3      for μ ∈ M do
2.4          for (ν, f, d) ∈ S do
2.5              S ← S \ {(ν, f, d)}                   // remove triple from S
2.6              if μ ∩ ν = ∅ then
2.7                  S ← (ν, f, μ :: d) :: S           // ν and μ are disjoint
2.8              else if ¬μ ∩ ν = ∅ then
2.9                  S ← (ν, μ :: f, d) :: S                          // ν ⊆ μ
2.10             else
2.11                 S ← (μ ∩ ν, μ :: f, d) :: S                  // μ ∩ ν ⊆ μ
2.12                 S ← (¬μ ∩ ν, f, μ :: d) :: S        // ¬μ∩ν and μ are disjoint
2.13     U ← set of first elements of each 3-tuple in S
2.14     return (U, S)
```

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

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

Algorithm 2 has a limitation that there may be $\nu_i, \nu_j \in \mathcal{U}$ 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 2 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 $2^{|\mathcal{M}|}$ min-terms. 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 2.10 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.

When implementing Algorithm 2, there are several things to take into account. As explained in Section 4.1, given a type, we may ask whether it is inhabited, so tests such as $\mu \cap \nu = \emptyset$ (lines 2.6 and 2.8) may be implemented as $(\mu \cap \nu).\mathit{inhabited} == \mathit{false}$, or as $(\mu.\mathit{disjoint}(\nu) == \mathit{true})$, or possibly as both as separate cases depending on the robustness of the code. For readability, in Algorithm 2, we use the notation :: for sets. If $\mathcal{U}$ is a set then $x :: \mathcal{U} = \{x\} \cup \mathcal{U}$.

The return value of Algorithm 2 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.

5.6 Thompson Construction of σDFA

Section 5.3 described the Brzozowski construction of a σDFA. In this section we summarize in far less detail, the Thompson [35, 38] construction. The Thompson construction for σDFA does not differ significantly from the classical case except in the necessary determinization step [37].

In the determinized σDFA, the set of outgoing transitions for each state, q, partition Σ into a maximal disjoint type decomposition given the labels originating in the non-deterministic automaton. To compute a state and its transitions in the determinized σDFA, the input is taken from two or more states of the non-deterministic automaton. Given n different input states, which each have m (or fewer) transitions, we have n potentially different partitions of Σ. A naive approach to computing the partition in the determinized automaton would be simply to compute the m^n intersections. All such intersections are indeed guaranteed to be disjoint, but after these type designators are simplified and canonicalized, it is thereafter difficult (sometimes impossible) to decide which types are subtypes of which of the given types. This is an algorithmic problem which can of course be solved, but turns out to be unnecessary, thanks to the `MDTD` algorithm (Section 5.5) which we already used in the Brzozowski construction. Given the set of n partitions of size m, we have $m \times n$ type designators which can be used as input for `MDTD`, each time producing a partition, and a mapping of which computed type is a subtype of which of the original types.

6 Sample Implementations

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 implements for RTE and its support libraries in Common Lisp, Scala, Clojure, and Python. The reference implementation is a Common Lisp library, `rte` [24, 22] which is available from Quicklisp [3]. The Clojure and Scala implementations, `clojure-rte` [23] and `cl-robdd-scala` [26], extend the Clojure and Scala type systems, which are already extensions of the type system of the JVM. The Python implementation, `python-rte` [25] extends the built-in Python type system.

In this article we do not delve into the details of the implementation of either 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.

```scala
def isEven(x: Any): Boolean =
  x match {
    case y: Int => y % 2 == 0
    case _ => false
  }
val even = SSatisfies(isEven, "even")
val r_0 = Star(Cat(classOf[Int], classOf[String], even))
val r_1 = Star(Cat(classOf[Int], Star(classOf[String]), even))

val dfa_0 = r_0.toDfa()
val dfa_1 = r_1.toDfa()

val v0 = dfa_0.simulate(Seq(1,"hello",2)) // returns Some(true)
val v1 = dfa_1.simulate(Seq(1,"hello","world",2,
                            3,4,
                            5,"hello",6)) // returns Some(true)
val v2 = dfa_1.simulate(Seq(1.1, 1.2, 1.3)) // returns None
```

Fig. 4: Scala code for constructing r_0 and r_1 Equations (1) and (2).

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 [35, sec 2.4.2]), vacuity checks, etc.

Figure 4 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 [9] 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 and object and invoke run-time logic as a function of subtype-ness 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 an 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. Ansi: American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999) (1994)
2. 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
3. Beane, Z.: Quicklisp beta (2015), https://www.quicklisp.org/beta
4. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for Clojure (2018)
5. 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/
6. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **35**, 677–691 (August 1986)
7. 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
8. 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/
9. EPFL: Scala Reflection Library 2.12.0 (2016), https://www.scala-lang.org/api/2.12.0/scala-reflect/scala/reflect/runtime/index.html
10. 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)
11. Grigore, R.: Java generics are turing complete. CoRR **abs/1605.05274** (2016), http://arxiv.org/abs/1605.05274
12. Heikkilä, M.: Malli, Metosin (2022), https://github.com/metosin/malli
13. Hickey, R.: The Clojure Programming Language. In: Proceedings of the 2008 symposium on Dynamic languages. p. 1. ACM (2008)
14. 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
15. 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)
16. 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

17. Jim Newton, Sébastien Doeraene, L.P.: Union types in scala 3 (feb 2020), https://contributors.scala-lang.org/t/union-types-in-scala-3/4046
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. Might, M., Darais, D., Spiewak, D.: Parsing with derivatives: A functional pearl. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. p. 189–195. ICFP '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2034773.2034801, https://doi.org/10.1145/2034773.2034801
21. Miller, A.: Spec Guide (2022), https://clojure.org/guides/spec
22. Newton, J.: Representing and Computing with Types in Dynamically Typed Languages. Ph.D. thesis, Sorbonne University (Nov 2018)
23. Newton, J.: Regular Type Expressions for Clojure (2024), github.com/jimka2001/clojure-rte
24. Newton, J.: Regular Type Expressions for Common Lisp (2024), github.com/jimka2001/cl-rte
25. Newton, J.: Regular Type Expressions for Python (2024), github.com/jimka2001/python-rte
26. Newton, J.: Regular Type Expressions for Scala (2024), github.com/jimka2001/scala-rte
27. Newton, J., Demaille, A., Verna, D.: Type-Checking of Heterogeneous Sequences in Common Lisp. In: European Lisp Symposium. Kraków, Poland (May 2016)
28. Newton, J., Pommellet, A.: A portable, simple, embeddable type system. In: European Lisp Symposium. Online, Everywhere (May 2021)
29. Newton, J., Verna, D.: Strategies for typecase optimization. In: European Lisp Symposium. Marbella, Spain (Apr 2018)
30. Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The Scala language specification (2004)
31. 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
32. Owens, S., Reppy, J., Turon, A.: Regular-expression Derivatives Re-examined. J. Funct. Program. **19**(2), 173–190 (Mar 2009)
33. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D.: Foundations of json schema. In: Proceedings of the 25th International Conference on World Wide Web. p. 263–273. WWW '16, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE (2016). https://doi.org/10.1145/2872427.2883029, https://doi.org/10.1145/2872427.2883029
34. van Rossum, G., Drake, F.L.: The Python Language Reference Manual. Network Theory Ltd. (2011)
35. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, USA (2009)
36. Valais, L., Newton, J., Verna, D.: Implementing baker's `SUBTYPEP` decision procedure. In: 12th European Lisp Symposium. Genova, Italy (Apr 2019)

37. Veanes, M., Bjørner, N., de Moura, L.: Symbolic automata constraint solving. In: Fermüller, C.G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 640–654. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
38. Xing, G.: Minimized Thompson NFA. Int. J. Comput. Math. **81**(9), 1097–1106 (2004), http://dblp.uni-trier.de/db/journals/ijcm/ijcm81.html#Xing04