Type-Checking Heterogeneous Sequences in a Simple Embeddable Type System

Jim E Newton

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 supported in a wide range of programming languages, both dynamically and statically typed. These sequences often exhibit type patterns such as repetition, alternation, and optionality. The programmer needs a mechanism to *declare* and query adherence to this regularity. 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. For type systems with subtyping, the subtype relation and type vacuity cannot always be computed programmatically. We provide a working, sound solution for constructing finite automata for type-based regular expressions even in cases where the subtype decidability relations is not computable retrospectively, but 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

Our goal is to declaratively describe a set of sequences (rational language) in a programming language based on regularities in the types of the sequence elements, and to efficiently decide membership of these rational languages at run-time.

This paper studies *rational type expressions* (RTEs) and the construction from RTE to symbolic, deterministic, finite automata (σDFA). RTEs are used to specify patterns in the types of the sequence elements, such as $(int \cdot str \cdot evenp)^*$, 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 in a generic enough way to be usable in multiple programming languages, and (2) how to construct σDFA from an RTE despite limitations in the type system. For the first challenge, the system defines 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 (decidable and otherwise) using Maximal Disjoint Type Decomposition (MDTD).

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, Section 2.

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., `(1.0,1.1,1.2)`. In a language whose type system forms a type lattice [14], 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 [33] and Common Lisp [2], support sequences where any element may be an object of any inhabited type whatsoever.

Element types of *heterogeneous sequences* usually follow an implicit, tacit, pattern in the mind of the programmer and hopefully documented in the code 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.

Dynamically typed languages such as Common Lisp, Clojure [10, 11], and Python commonly manipulate heterogeneous sequences. The Common Lisp type system, the Python `mypy` [17, 32] library, and Clojure `spec` [19] allow the annotation of type hints, which can sometimes invoke run-time type checks or help IDEs provide useful development and debug feedback. However, these type systems are not rich enough to express regular patterns in sequences of mixed types.

Scala [28, 29], a statically typed language with limited reflection [6], allows the program to manipulate sequences such as those coming from JSON [31], declared as `Seq[Any]`. Code pattern matches to implement `typecase` logic based on dynamic type meta-data from the JVM. [7].

What is lacking from many dynamic languages (or statically typed languages with sufficient 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 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 [5], 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 theoretical description of D'Antoni and Veanes [5] and Keil and Thiemann [15] to heterogeneously typed sequences in programming languages.

2. Provide an algorithm for the maximal disjoint type decomposition, `MDTD`, which, by construction, avoids the problem of undecidability of subtyping.
3. Demonstrate constructions of σDFA recognizing Regular Type Expressions despite an incomplete subtype predicate.
4. Provide sample implementations in four programming languages: Scala, Clojure, Python, and Common Lisp.

1.3 Previous Work

In [20], Newton used deterministic finite automata (DFA) and RTEs to recognize heterogeneous Common Lisp sequences based on type patterns. The current work, generalizes the RTEs to a wider range of programming languages.

Clojure Spec [19, 11, 10] and `metosin` [9] support some forms of type pattern recognition. After conversations between experts on public forums it is not clear whether Spec is based on finite automata theory at all or rather on NFA (non-deterministic finite automata) work by Might *et al.* [18]. An NFA-based procedure (presumably using backtracking) would have at least polynomial complexity, whereas our approach offers linear complexity.

Christophe Grande authored `seqexp` [1] for Clojure, a regular pattern matching library. According to an interview with Grande, `seqexp` does not use a finite automata approach, suggesting that the size of resulting code would violate the JVM [7] limitation of function size susceptible to optimization.

The Brzozowski derivative and an algorithm to compute it for digital circuits was first presented in 1964 by Janusz Brzozowski [4]. Owens *et al.* [30] presented a *modern* version applied to regular pattern recognition for sequences of characters. Owens noted that a practical obstacle to using this approach is large computation time of generating large finite automata over excessively large alphabets. We ameliorate this problem by considering sets rather than individual values.

D'Antoni and Veanes [5] argue that the generalization retains many of the good properties of their finite-alphabet counterparts. D'Antoni and Veanes discuss a decomposition of types referred to as *Minterms(...)*, *i.e.*, *the set of maximal satisfiable Boolean combinations*. D'Antoni's set is a less optimized version of our MDTD algorithm, Section 5.4.

Grigore [8], Kennedy and Pierce [16], from Microsoft Research, discuss subtype decidability in Java [7], Scala [28, 29], C#, and .NET Intermediate Language. The work curiously lacks citations for C# and .NET Intermediate Language, as if the reader is already intimately familiar with C# and .NET IL.

Hosoya, Vouillon, and Pierce [13] defined regular expression in the XDuce language, allowing static XML types to be defined recursively and hierarchically to describe the structure of XML documents. XDuce programs consume and manipulate XML [3] documents allowing the type checker to assure that the programmatic expressions are type correct according to the XML schema, DTD, XML-Schema *etc.*. RTEs as opposed to the XDuce, add such type checking ability to an existing dynamic type systems.

2 Symbolic Finite Automata

First we give an example of a σDFA. Section 2.1 provides formal definitions. Consider the three RTEs [26], r_1, r_3, and r_2, defined in (1), (2), and (3). The syntax should be intuitive to anyone already familiar with regular expressions.

$$r_1 = (int \cdot str \cdot evenp)^* \tag{1}$$
$$r_2 = (int \cdot str \cdot str^* \cdot evenp)^* \tag{2}$$
$$r_3 = (int \cdot str^* \cdot evenp)^* . \tag{3}$$

RTE, r_1, represents the set of sequences of arbitrary (finite) length, each of which consists of zero or more occurrences: integer, string, even integer. RTE, r_2 allows the string to occur one or more times. Finally, in RTE, r_3, the string is allowed to occur 0 or more times. *E.g.*, the sequence `(11,"a",12,13,"a","b",14)` matches r_3 and r_2, but not r_1; `(11,"a",12,13,"a",14)` matches all of them; and `(11.5,12.6)` matches none of three RTEs.

A finite automaton efficiently decides membership of a rational language. Analogous to classical finite automata theory [12], RTEs correspond to *symbolic* finite automata [5] (σDFA) over a possibly infinite alphabet, Σ. Transitions are labeled *symbolically* as subsets of the alphabet. In our case, Σ is the set of all values representable in a given programming language, including other sequences. RTEs, r_1 and r_2, are implemented in Figure 1 [left] and [right] respectively, r_1 being represented by a deterministic automaton and r_2 non-deterministic. The σDFA for r_3 is shown in Figure 2 [left].

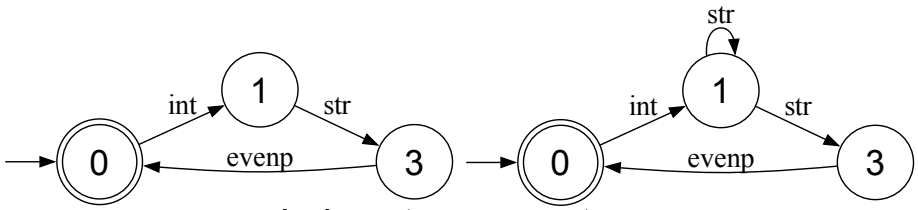
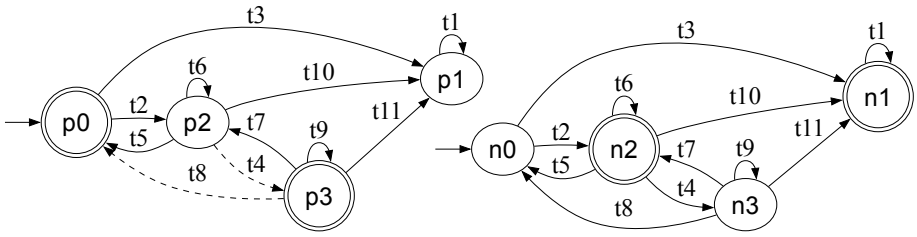


Fig. 1: σDFA for RTEs: [left] $r_1 = (int \cdot str \cdot evenp)^*$ and [right] non-deterministic automaton for $r_2 = (int \cdot str \cdot str^* \cdot evenp)^*$

We have chosen to work with deterministic automata, as opposed to non-deterministic, as they allow operations such as negation and intersection, and vacuity/habitation checks as well as disjoint and subset relations. Even though construction of DFAs can be slow and highly depends the syntactic representation of the regular expression, 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 independence is important because arbitrarily large expression trees may *reduce* to the same,

small RTE. Much of the work in σDFA construction focuses on preventing overlap between transitions, thus enforcing determinism.

2.1 σDFA Formally



$t_1 = \Sigma$

$t_2 = int$

$t_3 = \overline{int}$

$t_4 = str \cap evenp$

$t_5 = \overline{str} \cap evenp$

$t_6 = str \cap \overline{evenp}$

$t_7 = (int \cap \overline{evenp}) \cup (str \cap \overline{evenp})$

$t_8 = \overline{int} \cap \overline{str} \cap evenp$

$t_9 = (int \cap evenp) \cup (str \cap evenp)$

$t_{10} = \overline{str} \cap \overline{evenp}$

$t_{11} = \overline{int} \cap \overline{str} \cap \overline{evenp}$

Fig. 2: σDFAs A_p and A_n: [left] A_p is σDFA for RTE Equation (3), $r_3 = (int \cdot str^* \cdot evenp)^*$; [right] A_n is σDFA for RTE $!r_3$.

The finite automata in Figure 2 are called a *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.

A *symbolic finite automaton* is a structure: $A = (\Sigma, \Upsilon, Q, q_0, F, T)$ where:

1. Σ is a possibly countably infinite set of *objects*.
2. Υ is a set of *symbols*, each designating a subset of Σ.
3. Q is a finite set of *states*.
4. $q_0 \in Q$ is the *initial* state.
5. $F \subseteq Q$ is the set of *accepting* states.
6. $T \subseteq Q \times \Upsilon \times Q$ is a set of transitions.

If $\nu \in \Upsilon$, let $[\![\nu]\!]$ denote the designated subset of Σ; $\nu \in \Upsilon \implies [\![\nu]\!] \subseteq \Sigma$.

A transition, $(q, \nu, r) \in T$, is also denoted $\textcircled{q} \xrightarrow{\nu} \textcircled{r}$. We refer to q as the *origin* of the transition, r as the *target*, and $\nu \in \Upsilon$ as the *label*.

A *partition* of Σ is a set of mutually disjoint, possibly empty subsets of Σ whose union is Σ.

If for each $q \in Q$, the set of labels of transitions having q as origin designates a partition of Σ, then A is called a *complete, deterministic, symbolic, finite automaton* on Σ, or simply a σDFA.

The definition of partition is non-conventional in order to avoid an annoying incoherence between theoretical partition and computed partition. In particular given two labels $\nu_1, \nu_2 \in \Upsilon$, the set $\{\nu_1 \cap \nu_2, \overline{\nu_1} \cap \nu_2, \nu_1 \cap \overline{\nu}_2, \overline{\nu_1} \cap \overline{\nu}_2\}$ is called the *standard partition* ($\overline{\nu_i}$ denotes set complement) and fulfills *our* definition of partition because the sets are mutually disjoint by construction, and their union is Σ, despite the possibility that any of the four intersections be empty.

A transition $\textcircled{q} \xrightarrow{\nu} \textcircled{r}$ is called *satisfiable* if $[\![\nu]\!]$ is inhabited ($[\![\nu]\!] \neq \emptyset$); otherwise it is called *non-satisfiable*. 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 5.1.

3 Rational Expressions

We accept the following definitions, borrowed from classical finite automata theory [12]. A length-n ($n \geq 0$) *sequence* is a function $\{0, 1, \ldots, n-1\} \to \Sigma$. 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. Two sequences, s, t can be concatenated 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 length-n sequence, then the *cons*, $x :: s = (x) \cdot s$, thus extending the sequence to length $n+1$. Finally, the Kleene star of a language, L^*, is the set of all finite concatenations of zero or more sequences from L.

A *rational type expression* or RTE is defined as any expression as defined below which represents (recognizes) a language on Σ. Let f and g be RTEs, recognizing languages $[\![f]\!]$ and $[\![g]\!]$. Let Υ be a set of symbols such that $\{\Sigma, \emptyset\} \subset \Upsilon$: $\nu \in \Upsilon \implies [\![\nu]\!] \subseteq \Sigma$. The following rules recursively define all RTEs.

RTE	**Language**	**RTE**	**Language**	**RTE**	**Language**
$\emptyset$	$\emptyset$	$f \cdot g$	$[\![f]\!] \cdot [\![g]\!]$	f^*	$[\![f]\!]^*$
ε	$\{()\}$	$f + g$	$[\![f]\!] \cup [\![g]\!]$	$\lfloor \nu \rfloor$	$\{(a) \mid a \in [\![\nu]\!]\}$
Σ	$\{(a) \mid a \in \Sigma\}$	$f \,\&\, g$	$[\![f]\!] \cap [\![g]\!]$	$!f$	$\Sigma^* \setminus [\![f]\!]$

The set of all sequences *recognized* by an RTE is called its *language*.

4 Programming Language Supporting RTEs

Even if finite automata theory is more general, 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 finite, non-cyclic 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, Υ_0, of symbols such as `int` and `str` which allow an application program to perform run-time type membership checks. We

suppose that the programming language provides a mechanism for performing type membership and subtype checks in terms of the symbols in Υ_0. For example, in Scala `classOf[Number].isAssignableFrom(classOf[Integer])` and in Python `isinstance(42,int)` both return a Boolean *true*.

4.1 A Simple Embeddable Type System

Different programming languages make different assumptions about types. Newton and Pommellet [27] presented a *Simple Embeddable Type System* (SETS) which we use here. SETS keeps the type system simple enough to implement in a wide range of programming languages, and specifies that types already designatable in the programming language are accepted as atomic types in SETS. This type system suffices for our needs; although some limitations are described in Section 4.4. A formal definition of SETS can be found in [27], but we summarize it here.

A *type* is defined as 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**: $\Sigma, \emptyset \in \Upsilon$ representing the universal and empty types.
3. **Singleton types**: $\forall a \in \Sigma$, $\{a\} \in \Upsilon$, with $[\![\{a\}]\!] = \{a\}$.
4. **Predicates**: for any decidable function $f : \Sigma \to \{true, false\}$, implemented in the host language, $Sat(f) \in \Upsilon$ with $[\![Sat(f)]\!] = \{x \in \Sigma \mid f(x) = true\}$.
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 $[\![\overline{\nu}]\!] = \overline{[\![\nu]\!]} = \{x \in \Sigma \mid x \notin [\![\nu]\!]\}$.

With a clever choice of f, $Sat(f)$ may designate the same set as other composed types in SETS, albeit with less reasoning power; *e.g.*, it is impossible to determine whether $[\![Sat(f)]\!] \subseteq [\![Sat(g)]\!]$ or if $\overline{[\![Sat(f)]\!]} \cap \overline{[\![Sat(g)]\!]}$ is inhabited.

A programming language specific API must implement type designators and $\in$ (Boolean, decidable), $\subseteq$ (semi-Boolean[1]) procedures. *E.g.*, in `scala-rte` [25] and in `python-rte` [24] type designators are implemented as class `SimpleTypeD` and subclasses thereof; while in `clojure-rte` [22] they are implemented as s-expressions. Type designators (type specifiers) are native to Common Lisp. An `inhabited` semi-Boolean predicate can be implemented as $\nu \subseteq \emptyset$, or in a more clever way depending on the programming language.

4.2 The Problematic Subtype Relation

Why is the subtype relation important? Knowing the subtype relation is critical for proving other relations between types. We can prove types ν and μ equivalent

[1] By *semi-Boolean*, we mean function which returns *true*, *false*, or *dont-know*.

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 $\nu \cap \mu \subseteq \emptyset$.

The fact that the subtype relation is not always decidable exacerbates the challenge to guaranteed determinism. For example, the set of even integers, `evenp`, is a subset of the set of integers. However, `evenp` is disjoint from the set of strings. The fact that we cannot *decide* this programmatically, means we compute types such as $t_7 = (\mathit{int} \cap \overline{\mathit{evenp}}) \cup (\mathit{str} \cap \overline{\mathit{evenp}})$ in Figure 2. The component $(\mathit{int} \cap \overline{\mathit{evenp}})$ is reasonable, however the component $(\mathit{str} \cap \overline{\mathit{evenp}})$ is superfluous as all even integers are necessarily not strings.

We would like to represent $t_7 = (\mathit{int} \cap \overline{\mathit{evenp}}) \cup \mathit{str}$, but we cannot. Worse still, in the figure, transitions $(p_2) \xrightarrow{t_{10}} (p_1)$ and $(p_2) \xrightarrow{t_5} (p_3)$ are constructed to guarantee determinism—the two types are disjoint by construction. The type, $t_4 = \mathit{str} \cap \mathit{evenp}$, is empty; consequently the transition is unsatisfiable, and thus state (p_3) is useless. We would like to eliminate (p_3) along with all transitions to and from it; unfortunately, we cannot—lacking an omniscient oracle.

In Figure 2, the transitions leaving each state are labeled with mutually disjoint, but suboptimal types. *E.g.*, state (p_2) has four exiting transitions: t_4, t_5, t_6, and t_{10} which are mutually disjoint and a partition: $t_4 \cup t_5 \cup t_6 \cup t_{10} = \Sigma$.

4.3 Pragmatic Solution to the Subtype Problem

Our solution to the subtype problem is to employ a *semi-Boolean* subtype predicate. SETS (Section 4.1) proposes a type system equipped with a pragmatic subtype ($\subseteq$) semi-Boolean predicate which returns *true*, *false*, or *dont-know*, nevertheless equipped with a true-Boolean membership predicate ($\in$).

One might suppose that the inability to universally determine subtype or disjoint type relations will inevitably lead to non-deterministic transitions, but it does not. For σDFAs, undecidable does not imply non-deterministic. Why? Because given two types μ and ν, it is guaranteed *by construction* that the sets in the standard partition, $\mu \cap \nu$, $\mu \cap \overline{\nu}$, $\overline{\mu} \cap \nu$, and $\overline{\mu} \cap \overline{\nu}$ are mutually disjoint. These are disjoint independent of whether $\mu \subseteq \nu$, or whether μ and ν are disjoint. An example of transitions, disjoint by construction, can be seen with state (p_2) as was explained in Section 4.2.

Indeterminate transitions, leading to useless states, are common, as seen with the *evenp* type in Section 4.2. These indeterminate transitions do not cause misbehavior at run-time. Even if slightly inefficient, at run-time the `evenp` predicate returns `true`, given an even integer, and `false` otherwise.

Undecidable subtype relations, and consequently unsatisfiable transitions, lead to challenges in minimization. Whereas classical finite automata are always uniquely minimizable, σDFAs are not. Case in point, as discussed in Section 4.2, the system cannot eliminate state (p_3), even though the only transition leading to it is unsatisfiable.

It may occur (albeit not in this example) that certain transitions reference equivalent types; *e.g.*. $t_4 = \mathit{str}$ and $t_6 = \mathit{str} \cap \overline{\mathit{evenp}}$. These equivalent types, may prevent some σDFAs from being optimally minimized, in that certain equivalent

states cannot be recognized as such. 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 of 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.

4.4 Limitations of SETS

The type system we describe in Section 4.1 and used throughout our research lacks features some programmers might expect. Readers might wonder whether SETS is an appropriate interface to serve as a robust foundation of RTEs. We believe it is perfectly sufficient for the following reasons.

SETS does not explicitly mention tuple types, but exposes the tuple type built-into the language if available, *e.g.*, in Scala, `SAtomic(Tuple2[Int,Double])`.

SETS does not attempt to robustly express function types such as $\nu \to \mu$ as such would violate the axioms of SETS. SETS demands that the type membership predicate return a Boolean, and that the subtype predicate return a semi-Boolean. Suppose function $f : \alpha_1 \to \beta_1$. Asking whether $f \in \alpha_2 \to \beta_2$, requires a subtype check $(\alpha_1 \to \beta_1) \subset (\alpha_2 \to \beta_2)$, obtaining a semi-Boolean. Thus the function type membership predicate would need to be a semi-Boolean. This limitation is not actually troublesome in practice, for none of the languages we have used in our research (Common Lisp, Clojure, Scala, Python) have a run-time operator to test whether a given function is an element of a type $\nu \to \mu$.

SETS cannot know when additional types are defined at run-time, thus invalidating a memoized result. In Java, such loading is possible via run-time loading of `jar` files. Because of this issue, the Scala implementation of SETS can be configured in `open-world-view` or `closed-world-view`. In `closed-world-view`, we assume that no additional types will be defined. Thus if ν and μ are the only subclasses of τ, then $\tau \setminus (\nu \cup \mu)$ can be reduced to $\emptyset$.

SETS supports types such as $Sat(f)$ which wrap predicates of arbitrary complexity. Thus we cannot reason about the computation complexity of a type membership query such as $x \in Sat(f)$.

5 Brzozowski Derivative Construction of σDFA

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

While Brzozowski/Owens [4, 30] defined $\partial_a r$ (the derivative of regular expression, r, with respect to letter $a \in \Sigma$), we present $\partial_\nu r$ for type $\nu \in \Upsilon$.

Let r be an RTE and $\nu \in \Upsilon$ be a type designator, then the RTE $\partial_\nu r$, the *derivative* of r *with respect to* ν is defined such that $[\![\partial_\nu r]\!] = \{s \in \Sigma^* \mid \exists h \in [\![\nu]\!], h :: s \in [\![r]\!]\}$.

Consider the RTE from Equation (3): $r_3 = (int \cdot str^* \cdot evenp)^*$. Each sequence in $[\![r_3]\!]$ is either () or begins with an *int*. $\partial_{int}\, r_3$ is the set recognized by

$$\partial_{int}\,(int \cdot str^* \cdot evenp)^* = str^* \cdot evenp \cdot (int \cdot str^* \cdot evenp)^*$$

because if we take the subset of $[\![r_3]\!]$ containing sequences starting with an *int*, which is $[\![r_3]\!] \setminus \{()\}$, then strip off the head (an *int*) from each sequence; each of the remaining tails consists of zero or more *str* followed by *evenp*, then followed by zero or more occurrences of $(int \cdot str^* \cdot evenp)$.

We wish to compute a derivative of an RTE by recursively applying reduction rules in Figure 3. We introduce subtype based rules (10), (11) and (12) which generalize equivalence based rules which Owens [30] stated.

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

$$\partial_\nu\,(r^*) = \partial_\nu\, r \cdot r^* \quad (5)$$

$$\partial_\nu\,(r + s) = \partial_\nu\, r \;+\; \partial_\nu\, s \quad (6)$$

$$\partial_\nu\,(r \,\&\, s) = \partial_\nu\, r \;\&\; \partial_\nu\, s \quad (7)$$

$$\partial_\nu !r = !\partial_\nu\, r \quad (8)$$

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

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

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

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

Fig. 3: Brzozowski Derivative Rules. For Equation (8) see Section 5.3. Variables, ν and μ represent type designators, $[\![\nu]\!], [\![\mu]\!] \subseteq \Sigma$; while r and s represent RTEs.

Owens [30] and Keil [15] also give similar recursive rules for computing *nullability*, detecting whether $() \in [\![r]\!]$, as well as $1^{st}(r)$ which is set of symbols which appear as first positions in a regular expression, *i.e.* the set of type designators to which (10) and (11) will be applied. We omit these rules here for lack of space.

5.1 Support for Overlapping Types

The Brzozowski construction for finite alphabets does not encounter the problem of overlapping types, because every letter in the alphabet is distinct. On the contrary, in our case, the *labels* under considerations designate 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 5.4.

The derivative rules, (4) through (12), are similar to those Owens [30] 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 $\partial_\nu \lfloor \mu \rfloor$ we must consider several cases: $\nu \subseteq \mu$, in which case rule (10) applies and $\partial_\nu \lfloor \mu \rfloor$ reduces to ε; or $\nu \cap \mu = \emptyset$, in which case rule (11) applies and $\partial_\nu \lfloor \mu \rfloor$ 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 equivalence and disjoint relation, are sometimes undecidable. One might naively think that if we deconstruct the type designators ν and μ (as discussed in Section 4.3) to the standard partition, $\{\nu \cap \mu, \nu \cap \overline{\mu}, \overline{\nu} \cap \mu, \overline{\nu} \cap \overline{\mu}\}$, 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 clever to subsequently determine all subtype relations necessary. *E.g.*, the subtype procedure does not inherently know *ex post facto* that the types in question were generated by a partitioning algorithm.

We do not attempt to solve the undecidability problem, rather we solve the problem of computing the derivative in our `MDTD` algorithm (Section 5.4), which assures knowledge of subtype and disjointness *by construction. I.e.*, every time we need to compute $\partial_\nu \lfloor \mu \rfloor$, 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.

5.2 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, $\partial_\nu^{[n]} r \ \forall \nu \in \Upsilon$. Brzozowski [4, 30] argues that the set of all such derivatives is finite.

We wish to compute the states and transitions of the σDFA A_p in Figure 2 [left] corresponding to Equation (3), $r_3 = (int \cdot str^* \cdot evenp)^*$. We name the states $p_0 \ldots p_3$ to distinguish the states from A_n [left] which we will call $n_0 \ldots n_3$. Step 1: construct the initial state $\textcircled{p_0}$ corresponding to r_3. Step 2: compute $MDTD(\{int\})$, because int is the only type which may appear as first element of a sequence contained in $[\![r_3]\!]$. `MDTD` returns the partition $\Pi = \{int, \overline{int}\}$. Step 3: compute $\{p_1, p_2\} = \{\partial_\nu r_3 \mid \nu \in \{\overline{int}, int\}\}$ as in (13) and (14), by applying rules (5), (9), (10), and (11):

During the computation of (13) and (14), we encounter the computation of $\partial_{\overline{int}} \lfloor int \rfloor$ and $\partial_{int} \lfloor int \rfloor$. As explained in Section 5.1, $\partial_{int} \lfloor int \rfloor = \varepsilon$, because $[\![int]\!] \subseteq [\![int]\!]$, rule (10); and $\partial_{\overline{int}} \lfloor int \rfloor = \emptyset$, because $[\![\overline{int}]\!] \cap [\![int]\!] = \emptyset$, rule (11).

Input: r : an RTE
Output: Components of a σDFA as in Section 2.1.
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 $\nu \in MDTD(1^{st}(q_1.expression))$ `// 1st() see Owens [30]`
    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 $State(d)$
          $T \leftarrow\ (q_1, \nu, q_2) :: T$
          $W \leftarrow q_2 :: W$
          $Q \leftarrow q_2 :: Q$
  $F \leftarrow \{q \in Q \mid () \in [\![q.expression]\!]\}$ `// see Owens [30]`
  return (Q, q_0, F, T)

Algorithm 1: Compute DFA by Brzozowski derivative

Having computed the RTEs, p_1 and p_2, we create two new states (in the σDFA), labeled (p_1) and (p_2), and add two transitions, one for each derivative: $(p_0) \xrightarrow{int} (p_2)$ and $(p_0) \xrightarrow{\overline{int}} (p_1)$, using the values of ν as the respective labels.

We process the constructed states in any order. Resulting derivative computations are shown in Figure 4. As we encounter RTEs not yet seen, we simply add them to a to-do list. The only types which can be the first element of a sequence in $[\![p_2]\!]$ are *str* and *evenp*, so when we process (p_2), we compute $MDTD(\{str, evenp\})$, using Algorithm 2, to obtain the partition $\Pi = \{str \cap evenp,\ \overline{str} \cap evenp,\ str \cap \overline{evenp},\ \overline{str} \cap \overline{evenp}\}$, and the additional information in Figure 5, which provides the subtype and disjoint relations needed to apply reduction rules (10) and (11), when computing $\{\partial_\nu\, p_2 \mid \nu \in \Pi\}$.

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; $() \in [\![r_3]\!]$, $() \notin [\![q_1]\!]$, $() \notin [\![q_2]\!]$, and $() \in [\![q_3]\!]$. Owens [30], provides a simple algorithm for deciding whether $\varepsilon \subseteq [\![r]\!]$; we omit the algorithm in this article.

5.3 Constructing a σDFA from a Negated RTE

We consider (8) in more detail. Keil and Thiemann [15] address the question: under which conditions $\partial_\nu !r = !\partial_\nu\, r$? They argue that the equivalence holds

$$
\begin{array}{llll}
 & & & \\
p_0 = & r_3 & = (int \cdot str^* \cdot evenp)^* & \\
p_1 = & \partial_{t_3}\, p_0 & = \partial_{\overline{int}}\, p_0 = \emptyset & (13) \\
p_2 = & \partial_{t_2}\, p_0 & = str^* \cdot evenp \cdot (int \cdot str^* \cdot evenp)^* & (14) \\
p_3 = & \partial_{t_4}\, p_2 & = \partial_{str \cap evenp}\, p_2 & \\
 & & = str^* \cdot evenp \cdot (int \cdot str^* \cdot evenp)^* & \\
 & & \quad + (int \cdot str^* \cdot evenp)^* & (15)
\end{array}
\quad\Big\|\quad
\begin{array}{l}
\partial_\nu\, p_1 = \emptyset \quad \forall \nu \\
\partial_{t_6}\, p_2 = p_2 \\
\partial_{t_5}\, p_2 = p_0 \\
\partial_{t_{10}}\, p_2 = p_1 \\
\partial_{t_7}\, p_3 = p_2 \\
\partial_{t_9}\, p_3 = p_3 \\
\partial_{t_8}\, p_3 = p_0 \\
\partial_{t_{11}}\, p_3 = p_1
\end{array}
$$

Fig. 4: Computation of Transitions of σDFA in Figure 2.

Type designator	**Supertypes**	**Disjoint types**
$str \cap evenp$	$\Sigma, str, evenp$	$\emptyset, \overline{str}, \overline{evenp}$
$str \cap \overline{evenp}$	$\Sigma, str, \overline{evenp}$	$\emptyset, \overline{str}, evenp$
$\overline{str} \cap evenp$	$\Sigma, \overline{str}, evenp$	$\emptyset, str, \overline{evenp}$
$\overline{str} \cap \overline{evenp}$	$\Sigma, \overline{str}, \overline{evenp}$	$\emptyset, str, evenp$

Fig. 5: `MDTD` computation for types: $\{str, evenp\}$.

whenever $\bigcup_{a \in [\![\nu]\!]} [\![\partial_{\{a\}}\, r]\!] = \bigcap_{a \in [\![\nu]\!]} [\![\partial_{\{a\}}\, r]\!]$, and that condition is valid whenever ν is selected from a partition of a union of types which appear as a $1^{st}(r)$, which is what our MDTD algorithm (Section 5.4) enforces.

The identity, $\partial_\nu !r = !\partial_\nu\, r$, can also be seen visually in A_p and A_n, Figure 2 [left] and [right] respectively. In order for A_n to accept the language complementary to A_p, the underlying graph structures of the two σDFAs must be isomorphic except that state acceptance is toggled. Consequently, state (n_0) corresponds to the RTE $!p_0$; *i.e.*, $n_0 = !p_0$. For the Brzozowski construction to be valid the RTE associated with (n_2) must be $\partial_{t_2}\, n_0$; *i.e.* $n_2 = \partial_{t_2}\, n_0$.

How do we know that $n_2 = !\partial_\nu\, p_0$? Because the language of state (n_2) is the complement of the language of state (p_2); *i.e.* $[\![n_2]\!] = \overline{[\![p_2]\!]}$.

$$[\![\partial_{t_2}\, !p_0]\!] = [\![\partial_{t_2}\, n_0]\!] = [\![n_2]\!] = \overline{[\![p_2]\!]} = \overline{[\![\partial_{t_2}\, p_0]\!]}\,.$$

5.4 Maximal Disjoint Type Decomposition (MDTD)

Newton [21] presented a streamlined algorithm to compute `MDTD` (Maximal Disjoint Type Decomposition) which creates other artifacts useful in σDFA construction especially in light of undecidability of subtype or disjoint procedures.

Given a set of potentially intersecting type designators, $\mathcal{M} = \{\mu_1, \mu_2, \ldots, \mu_n\}$, Algorithm 2 computes a pair, $(\Pi, \mathcal{S})$, where $\mathcal{S}$ is meta-data described below, and

Input: $\mathcal{M}$: a set of type designators
Output: $(\mathcal{U}, \mathcal{S})$: a pair as described in Section 5.4.
begin
  $\mathcal{S} \leftarrow \{(\top, \{\top\}, \{\bot\})\}$ `// working list of triples`
  for $\mu \in \mathcal{M}$ **do**
    for $(\nu, f, d) \in \mathcal{S}$ **do**
      $\mathcal{S} \leftarrow \mathcal{S} \setminus \{(\nu, f, d)\}$ `// remove triple from` $\mathcal{S}$
      if $\mu \cap \nu = \emptyset$ **then**
        $\mathcal{S} \leftarrow (\nu, f, \mu :: d) :: \mathcal{S}$ `//` ν `and` μ `are disjoint`
      else if $\overline{\mu} \cap \nu = \emptyset$ **then**
        $\mathcal{S} \leftarrow (\nu, \mu :: f, d) :: \mathcal{S}$ `//` $\nu \subseteq \mu$
      else
        $\mathcal{S} \leftarrow (\mu \cap \nu, \mu :: f, d) :: \mathcal{S}$ `//` $\mu \cap \nu \subseteq \mu$
        $\mathcal{S} \leftarrow (\overline{\mu} \cap \nu, f, \mu :: d) :: \mathcal{S}$ `//` $\overline{\mu} \cap \nu$ `and` μ `are disjoint`
  $\mathcal{U} \leftarrow$ set of first elements of each 3-tuple in $\mathcal{S}$
  return $(\mathcal{U}, \mathcal{S})$

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

$\Pi = \{\nu_1, \nu_2, \ldots, \nu_m\}$ is a new set of type designators designating a partition of Σ, such that 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 $\nu_i, \nu_j \in \Pi$ such that $[\![\nu_i]\!] = [\![\nu_j]\!] = \emptyset$, because the subset predicate may return *dont-know*; see Section 4.3.

`MDTD` has worst-case (time and space) complexity $\Omega\left(2^{|\mathcal{M}|}\right)$, if the final `else` 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.

As mentioned above, the return value of Algorithm 2 contains as meta-data a set, $\mathcal{S}$, of triples, (ν, f, d): ν is an element of the partition, Π; 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

One of the goals of our project is to design RTE abstractly enough to be implementable in multiple programming languages. As a proof of concept, we provide open source implements for RTE and its support libraries in Common Lisp [2], Scala [28, 29], Clojure [10, 11], and Python [33]. The original implementation is a Common Lisp library `cl-rte` [23], first presented in [20]. The Clojure (`clojure-rte` [22]) and Scala (`scala-rte` [25]) libraries extend the Clojure and Scala type systems, which are already extensions of the type system of the JVM. The Python library, `python-rte` [24], extends the built-in Python type system.

```
def evenp(x: Any): Boolean =
  x match {
    case y: Int => y % 2 == 0
    case _ => false
  }
val even = SSatisfies(evenp,"even")
val int = Atomic(classOf[Int])
val str = Atomic(classOf[String])
val r0 = (int ++ str ++ even).*
val r1 = int ++ str.* ++ even).*
```

```
val dfa = r1.toDfa()
val s1= Seq(1,"hello",3)
val s2= Seq(1,"hello","world",2,
            3,4,
            5,"hello",6)
val s3 = Seq(1.1, 1.2, 1.3)

r0.contains(s1) // Some(false)
dfa.simulate(s2) // Some(true)
dfa.simulate(s3) // None
```

Fig. 6: Scala code for constructing r_1 and r_3 Equations (1) and (3). `r0.contains(s1)` asks whether `s1` is in the language of the RTE, while `dfa.simulate(s2)` asks whether `s2` is in the language of the σDFA.

In this article we suppress most of the specifics of these implementations. We invite the reader to download the code from the links provided. Figure 6 shows a small example of how RTEs can be used in a Scala program.

In each of these `*-rte` libraries, users may specify RTEs based on fundamental types in the language, or 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 σDFA constructions 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 [34, sec 2.4.2]), vacuity checks, etc.

7 Conclusion and Perspectives

We have presented a foundation sufficient for implementing RTEs in various programming languages using an adaptation of the Brzozowski construction algorithm. Two challenges for implementing RTE in a host language are 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 5.4) as solutions to these challenges, along with sample implementations in Common Lisp, Clojure, Scala, and Python.

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. 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. 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/
4. 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
5. 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/
6. EPFL: Scala Reflection Library 2.12.0 (2016), https://www.scala-lang.org/api/2.12.0/scala-reflect/scala/reflect/runtime/index.html
7. 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)
8. Grigore, R.: Java generics are turing complete. CoRR **abs/1605.05274** (2016), http://arxiv.org/abs/1605.05274
9. Heikkilä, M.: Malli, Metosin (2022), https://github.com/metosin/malli
10. Hickey, R.: The Clojure Programming Language. In: Proceedings of the 2008 symposium on Dynamic languages. p. 1. ACM (2008)
11. 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
12. 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)
13. 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
14. 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
15. Keil, M., Thiemann, P.: Symbolic Solving of Extended Regular Expression Inequalities. In: Raman, V., Suresh, S.P. (eds.) 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014). Leibniz International Proceedings in Informatics (LIPIcs), vol. 29, pp. 175–186. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2014). https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSTTCS.2014.175
16. 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/
17. Lehtosalo, J.: The MyPy Project (2022), http://mypy-lang.org
18. 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 Ma-

chinery, New York, NY, USA (2011). https://doi.org/10.1145/2034773.2034801, https://doi.org/10.1145/2034773.2034801
19. Miller, A.: Spec Guide (2022), https://clojure.org/guides/spec
20. Newton, J.: Representing and Computing with Types in Dynamically Typed Languages. Ph.D. thesis, Sorbonne University (Nov 2018)
21. Newton, J.: An elegant and fast algorithm for paritioning types. In: European Lisp Symposium. Amsterdam, Netherlands (Apr 2023)
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. Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The Scala language specification (2004)
29. 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
30. Owens, S., Reppy, J., Turon, A.: Regular-expression Derivatives Re-examined. J. Funct. Program. **19**(2), 173–190 (Mar 2009)
31. 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
32. Rak-amnouykit, I., McCrevan, D., Milanova, A., Hirzel, M., Dolby, J.: Python 3 types in the wild: a tale of two type systems. In: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages. p. 57–70. DLS 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3426422.3426981, https://doi.org/10.1145/3426422.3426981
33. van Rossum, G., Drake, F.L.: The Python Language Reference Manual. Network Theory Ltd. (2011)
34. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, USA (2009)