Type-Checking Heterogeneous Sequences in a Simple Dynamic Type System

Abstract

Heterogeneously typed sequences are frequently used in dynamically typed programming languages. These sequences often exhibit patterns in terms of types, such as repetition, alternation, and optionality. The programmer needs a mechanism to designate or query adherence to these regular patterns. Regular languages and the theory of finite automata over finite alphabets was conceived for a similar purpose, 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 regular type expressions as a means of designating regular patterns in heterogeneous sequences. We formalize the connection to symbolic finite automata. We present a procedure for constructing a symbolic finite automaton based on the Brzozowski derivative and demonstrate the construction of σ-DFAs in a dynamic programming language equipped with a simple type system.

ACM Reference Format:

. 2026. Type-Checking Heterogeneous Sequences in a Simple Dynamic Type System. In *Proceedings of Dynamic Languages Symposium (DLS'20).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Motivation

In statically typed languages, such as Java or C++, sequences of fixed types exist such as `Array[String]`, e.g., `["ab","cd","xyz"]`, or `List[Double]`, e.g., `List(1.0,1.1,1.2)`, which represent sequences for which all the elements are of type `String` or all the elements are of type `Double`. In a language whose type system forms a type lattice [13], it is possible to declare an `Array[String|Double]` to represent a sequence for which each element is in turn either a `String` or a `Double`, e.g., `["ab",1.1,1.2,"cd","xyz"]`. By contrast, dynamically typed languages, such as Python, support sequences of arbitrary elements, where any element of the sequence may be an object of any type whatsoever.

Even if the language supports these *heterogeneous sequences*, it is rare that the types of elements in a sequence are completely random. It is often the case in a given program, that the types of the elements follow some pattern which is implicit in the mind of the programmer and hopefully documented in the comments of the functions and data structures. In such a case the programmer writes code which assumes the elements to be a certain type, or writes ad-hoc code to check the contents of the sequences at runtime.

Common Lisp [2] allows the programmer the flexibility to build a sequence of arbitrary types of elements, but also allows the programmer to make type declarations which make promises to the compiler about those types. These promises allow the compiler to generate optimized code or efficient memory allocation in many cases.

What is otherwise lacking from many dynamic languages, and which we address in this article, is a mechanism for the programmer to declare the type patterns which are expected, allowing the sequences to be efficiently type-checked at runtime, and thereafter to allow the 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 designating sets of sequences based on so called rational type expressions (RTEs), and constructing symbolic finite automata [7] (σ-DFAs) from them. The σ-DFAs, thus constructed, are used to perform computations on finite sequences whose constituent elements are taken from an infinite set of values supported in a dynamic programming language.

Our contributions in this article are as follows. We

1. present a variant of symbolic finite automata, which we call σ-DFA;
2. formalize the connection of RTEs in programming languages which support simple type systems to σ-DFAs;
3. generalize the Brzozowski derivative technique to construct σ-DFAs in a programming language equipped with a sufficiently powerful simple type system; and
4. adapt previous Common Lisp-based work for use in other dynamic languages, and publish a new sample implementation of RTEs in Clojure.

DLS'20, Chicago, Illinois, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.3 Previous Work

In [16] Newton presented a technique to reason about types of heterogeneous sequences in Common Lisp. In the current work, we formalize the technique and generalize it to such sequences in a wider range of dynamic languages.

In [16], Newton used deterministic finite automata (DFA) to recognize certain sequences based on the types of values therein. In the same work, Newton introduced the concept of RTE as a means of declaratively specifying such sequences. RTEs characterize these sequences of values analogously to how regular expressions from classical finite automata (FA) theory characterize strings of letters in regular languages.

The `clojure-rte` implementation bears some resemblance to the Clojure Spec library, https://clojure.org/guides/spec. We have not found peer reviewed articles explaining the theory behind Spec, although Hickey [11] mentions its existence. Thus far, private conversations between experts have lead us to contradictory conclusions that Spec is not based on FA theory at all, and other claims that it is based on NFA (non-deterministic finite automata) work by Might *et al.* [15]. An NFA-based sequence recognition procedure (presumably using backtracking) would have at least polynomial complexity, whereas our approach offers linear complexity. More research is needed to determine how much overlap exists between these two systems.

1.4 Outline of this Article

This paper is presented as follows. Section 2 generalizes classical FA to present σ-DFAs by means of a formal definition and examples. Section 3 presents the rational sets and rational languages which are associated with σ-DFAs. Section 4 describes the assumptions we make about this programming language in order to use σ-DFA to recognize regularly typed sequences. Section 5 generalizes the Brzozowski derivative [6] to support RTEs. The section states the basic formulas needed to construct a σ-DFA based on a regular type expression, and presents intuitive examples. Section 6 describes the procedure to construct a σ-DFA using the Brzozowski derivative presented in the previous section, and states an explicit algorithm. We take special care to explain how our development differs from classical finite automata theory. Section 7 presents a brief description of two sample implementations (in Common Lisp [2] and Clojure [10, 11]) which are publicly available. We conclude in Section 8 by explaining some of our perspective future work.

2 Symbolic Finite Automata

In this article, we consider FA similar to that shown in Figure 1. These FA are called σ-DFA, which will be formally defined in Definition 2.2, but first we convey an intuition to aid in understanding the formalism.

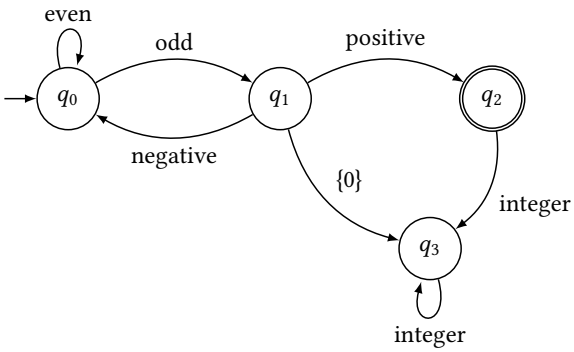


Figure 1. Example of σ-DFA, with the transitions from each state partitioning the set of integers.

2.1 Intuition

Whereas a classical FA has transitions which are labeled with letters from a finite alphabet (elements of a finite set), the transitions of a σ-DFA are labeled with symbols denoting sets of elements of a possibly infinite set. In fact the σ in σ-DFA stands for *symbolic*. An important restriction on the automaton is that if we consider the set of transitions originating at any given state, then every element of the alphabet is applicable to exactly one transition. In other words, the transitions of every state *partition* the alphabet. In Figure 1, if we consider state q_1, any element, i of the integers, then exactly one is true: $i \in \{0\}$, $i \in$ *positive*, or $i \in$ *negative*.

D'Antoni and Veanes [7] argue that the generalization that we present retains many of the good properties of their finite-alphabet counterparts. As the reader might imagine, many intuitions about FA carry over to σ-DFA, but as we will see, some do not. We must modify the classical theory in some subtle ways. To start with, a σ-DFA differs from a classical FA in two significant ways.

1. We do not assume the alphabet to be finite.
2. Transitions are labeled not by individual elements from the alphabet, but rather by symbols representing possibly infinite sets of such elements.

2.2 Deterministic Finite Automata

If A and B are sets, then we denote the power set of A as $\mathfrak{P}(A)$. The power set of the power set is denoted $\mathfrak{P}^2(A)$.

We denote the disjoint set relation as $A \parallel B$; *i.e.*, $A \parallel B$ means $A \cap B = \emptyset$. $A \nparallel B$ means $A \parallel B$ is false. The symbol $\bigsqcup$ indicates the *finite* union of mutually disjoint sets.

We denote the subset relation as $A \subset B$.

We now state precise definitions for core concepts in our treatment of σ-DFA. Readers who are less interested in precise, rigorous definitions may skip directly to Example 2.5 where we clarify the definitions with examples.

Definition 2.1 (disjoined, partition)**.** A finite set of possibly infinite sets, $A = \{A_1, A_2, \ldots A_n\}$, is said to be *disjoined* if its

elements are pairwise disjoint. A is said to be a *partition* of its union, B; *i.e.*, if $B = \bigsqcup_{i=1}^{n} A_i$, ($A_i, A_j$ pairwise disjoint). △

Definition 2.2 (σ-DFA)**.** A *σ-deterministic, complete finite automaton*, or *σ-DFA*, is a structure: $A = (\Sigma, Q, \xi, q_0, F, \delta)$ 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. $F \subset Q$ is the set of *accepting* states.
5. $\widehat{\delta} : Q \times \Sigma \to Q$ is the transition function between states.
6. $\xi : Q \to \mathfrak{P}^2(\Sigma)$ *s.t.* $\forall q \in Q : \bigsqcup \xi(q) = \Sigma$.
7. $\forall\, q \in Q\ \exists\, \delta_q : \xi(q) \to Q$. △

Axioms 6 and 7 of Definition 2.2 may be difficult to understand on casual reading. These axioms describe a partitioning function, ξ, which specifies how the alphabet, Σ, is finitely partitioned for each state q. By axiom 6, we mean that for any state, q, that Σ is finitely partitioned into a set, $\xi(q) \subset \mathfrak{P}(\Sigma)$. By axiom 7, we mean that for each element of such a partition, $v \in \xi(q)$, there is a transition, $q \xrightarrow{v} \delta_q(v)$. Definition 2.3 gives notation associated with transitions.

Definition 2.3 (transition)**.** Let $A = (\Sigma, Q, \xi, q_0, F, \delta)$ be a σ-DFA, and let $q \in Q$, $v \in \xi(q)$, and $r = \delta_q(v)$; then $q \xrightarrow{v} r$ is called the *transition* from *origin* q to *target* r with *label* v. △

Our definition of *transition* for σ-DFAs differs subtly from the corresponding definition in classical FA theory. [12] Classically, a transition from an origin state to a target state is labeled by an *element of* Σ, whereas in our case a transition is labeled with a *subset of* Σ.

Definition 2.4 (sink state)**.** A σ-DFA may have a state, q (not necessarily unique), such that $\xi(q) = \{\Sigma\}$ and $\delta_q(\Sigma) = q$; *i.e.* $q \xrightarrow{\Sigma} q$. Such a state is called a *sink state*. △

By our terminology, every σ-DFA is *complete*. *I.e.*, each state maps the entirety of Σ to the states Q. However, when displayed graphically, we usually omit drawing any sink states. Thus it may appear from the graph that $\bigsqcup \xi(q) \neq \Sigma$. In Figure 1 we may simply omit drawing state q_3 and any transitions having q_3 as target. We may do this because all sink states are indistinguishable.

Example 2.5 (A σ-DFA)**.**

We now clarify some of the definitions and notations introduced thus far by showing a σ-DFA over the set of integers; $\Sigma = \mathit{integer}$. Equations (1), (2), and (3) imply three partitions of *integer*.

$$\begin{aligned} \mathit{integer} &= \mathit{even} \sqcup \mathit{odd} && (1)\\ &= \mathit{positive} \sqcup \{0\} \sqcup \mathit{negative} && (2)\\ &= \mathit{integer} && (3) \end{aligned}$$

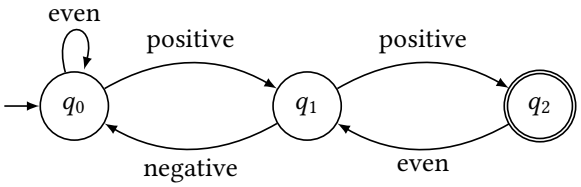
In the σ-DFA shown in Figure 1 of Section 2, $A = (\Sigma, Q, q_0, F, \delta)$, $Q = \{q_0, q_1, q_2, q_3\}$, and $F = \{q_2\}$. Transitions with origin q_0, q_1, and q_2 each correspond to the different partitions.

The values of ξ are: $\xi(q_0) = \{\mathit{even}, \mathit{odd}\}$
$\xi(q_1) = \{\mathit{positive}, \{0\}, \mathit{negative}\}$
$\xi(q_2) = \xi(q_3) = \{\mathit{integer}\}$ △

The σ-DFA is deterministic and complete, because each of $\xi(q_0)$, $\xi(q_1)$, and $\xi(q_2)$ is a partition of Σ.

$$\bigsqcup \xi(q_0) = \mathit{even} \sqcup \mathit{odd} = \Sigma$$
$$\bigsqcup \xi(q_1) = \mathit{positive} \sqcup \{0\} \sqcup \mathit{negative} = \Sigma$$
$$\bigsqcup \xi(q_2) = \bigsqcup \xi(q_3) = \mathit{integer} = \Sigma$$

Example 2.6 (Automaton not deterministic)**.**



This automaton fails to be deterministic because

$$\xi(q_0) = \{\mathit{even}, \mathit{positive}\}$$

which is not disjoined; $\mathit{even} \not\parallel \mathit{positive}$. △

2.3 Alternative interpretation

Another way to think an σ-DFA is by imagining the transitions labeled with letters from Σ as in the classical case. *I.e.*, think of the graph in Figure 1 as having transitions $q_o \xrightarrow{1} q_1$, $q_o \xrightarrow{3} q_1$, $q_o \xrightarrow{5} q_1$, etc, but rather than drawing infinitely many transitions, we instead group the transitions between any two pair of states into skiens [27], and label the skiens with a set such as $q_0 \xrightarrow{\{1,3,5,\ldots\}} q_1$. Since a graph with finitely many vertices, V, has finitely many skiens (at most $|V|^2$), even if it has infinitely many edges.

Using this alternative interpretation, we may conveniently take $q \xrightarrow{\emptyset} r$ to mean that there is no transition from q to r, because the set of transitions q to r is the empty set.

Definition 2.7 (satisfiable)**.** The transition $q \xrightarrow{v} r$ is called *satisfiable* if $v \neq \emptyset$, and is called *non-satisfiable* if $v = \emptyset$.

The challenge of determining programmatically whether a transition is satisfiable is addressed in Section 4.2.

3 Rational Sets and Rational Expressions

In Section 2 the σ-DFA was presented as a graph. In Section 6 we will construct such a graph programmatically. To describe such a construction, we will need a tool called the Brzozowski derivative, the development of which a set theoretical representation needed. We present that set theoretical view in this section. Just as there is a homomorphism between DFA and

rational expressions, there is an analogous homomorphism between σ-DFAs an regular type expressions (RTEs). [16, 19]

3.1 Sequences and Sets of Sequences

Definition 3.1 (sequence)**.** A *sequence*, s, of length $n \geq 0$ on set Σ, is a function which maps the closed integer interval $[1, n]$ into Σ; *i.e.*, $s : [1, n] \rightarrow A$. We may denote the i^{th} element of a sequence either as $s(i)$ or as s_i. We may likewise denote the sequence element-wise as $(s_1\ s_2\ ...\ s_n)$.

In the case that $n = 0$, $s\colon \emptyset \rightarrow \Sigma$ is denoted $()$ and called an *empty sequence*. △

Definition 3.2 (Σ^*)**.** We will take Σ^* to mean the set of all sequences of any finite length on Σ. △

Definition 3.3 (concatenation)**.** If $f, g \in \Sigma^*$, then we define the *concatenation* of the sequences f and g as the sequence on Σ comprising f followed immediately by the sequence of characters comprising g. Such a concatenation of sequences is denoted by $a \cdot b$, or as ab when it is unambiguous. △

Definition 3.4 ($A \cdot B$)**.** If $A, B \subset \Sigma^*$, then we define $A \cdot B$ as follows: $A \cdot B = \{f \cdot g \mid f \in A \text{ and } g \in B\}$. △

Theorem 3.5. *The set concatenation operator is associative. If $A, B, C \subset \Sigma^*$, then $(A \cdot B) \cdot C = A \cdot (B \cdot C)$.*

The proof follows from associativity of sequence concatenation.

Definition 3.6 (A^n)**.** If $A \subset \Sigma^*$, then we define A^n to mean

$$A^n = \begin{cases} \{()\} & \text{if } n = 0 \\ A \cdot A^{n-1} & \text{if } n > 0 \end{cases} . \qquad \triangle$$

If we also think of A^2 intuitively as $A \cdot A$, then we have two definitions for A^2; *i.e.* $A^2 = A \cdot A$ *vs.* $A^2 = A \cdot A^1$. It is not hard to show that these two definitions define the same set. See [16, Corollary 3.13,p 40] for details.

Definition 3.7 (A^*)**.** If $A \subset \Sigma^*$, then A^*, the *Kleene closure of* A [12], denotes the set of sequences, w, such that $w \in A^n$ for some $n \in \mathbb{N}$. △

Definition 3.8 (tail)**.** If a is a sequence of length $n > 1$ on set Σ, then we define $tail(a)$ (the *tail* of a) as sequence of length $n - 1$ such that $(tail(a))(i) = a(i + 1)$ for $1 \leq i < n$. △

Definition 3.9 (cons)**.** If a is a sequence of length n on set Σ, and if $x \in \Sigma$; then we define the *cons* of x with a, denoted $x :: a$ to be the sequence of length $n + 1$ such that

$$(x :: a)(i) = \begin{cases} x & \text{if } i = 1 \\ a(i-1) & \text{if } i \leq n + 1 \end{cases} . \qquad \triangle$$

Definition 3.10 ($\lfloor A \rfloor$, singleton sequence)**.** If $A \subset \Sigma$ we take $\lfloor A \rfloor$ to mean the set of sequences of length 1

$$\lfloor A \rfloor = \{(a) \mid a \in A\} = \{f \in \Sigma^* \mid f_0 \in A \text{ and } tail(f) = ()\}.$$

Each element of $\lfloor A \rfloor$ is called a *singleton sequence*. △

3.2 Clarifying the Notation

The notations A^1 *vs.* $\lfloor A \rfloor$ may unfortunately be confusing. We clarify the respective meanings of $\lfloor A \rfloor^1$ and $\lfloor A^1 \rfloor$, which both reduce to $\lfloor A \rfloor$.

If $A \subset \Sigma^*$ then $A^1 = A$. However, if $A \subset \Sigma$ then $\lfloor A \rfloor \subset \Sigma^*$ is the set of singleton sequences, each a sequence of length 1. Moreover, as $\Sigma^* \subset \Sigma$, (the reason for which is discussed in Section 4) then if $A \subset \Sigma^*$ it is still makes sense to talk about $\lfloor A \rfloor$. If $A \subset \Sigma^*$ then $\lfloor A \rfloor$ is a set of singleton sequences whose elements are themselves sequences.

Since $\lfloor A \rfloor \subset \Sigma^*$ we know that $\lfloor A \rfloor^1 = \lfloor A \rfloor$. As an example let $A = \{(1, 2, 3), (1, 2, 3, 4)\} \subset \Sigma^* \subset \Sigma$ then $\lfloor A \rfloor = \{\big((1, 2, 3)\big), \big((1, 2, 3, 4)\big)\}$.

If $A \subset \Sigma$, then $\lfloor A \rfloor^1 = \lfloor A \rfloor$. As an example let $A = \{1, 2, 3, (4, 5)\} \subset \Sigma$, then $\lfloor A \rfloor = \{(1), (2), (3), ((4, 5))\}$.

3.3 Rational Sets

Definition 3.11 (rational set, $\mathcal{L}_\Sigma$)**.** The set $\mathcal{L}_\Sigma \subset \mathfrak{P}(\Sigma^*)$ (*i.e.* each element of $\mathcal{L}_\Sigma$ is a subset of Σ^*) is defined by the following recursive definition.

$$\emptyset \in \mathcal{L}_\Sigma \tag{4}$$

$$\{()\} \in \mathcal{L}_\Sigma \tag{5}$$

$$\lfloor A \rfloor \in \mathcal{L}_\Sigma \ \forall A \subset \Sigma \tag{6}$$

$$(A \cup B) \in \mathcal{L}_\Sigma \ \forall A, B \in \mathcal{L}_\Sigma \tag{7}$$

$$(A \cdot B) \in \mathcal{L}_\Sigma \ \forall A, B \in \mathcal{L}_\Sigma \tag{8}$$

$$A^* \in \mathcal{L}_\Sigma \ \forall A \in \mathcal{L}_\Sigma \tag{9}$$

$\mathcal{L}_\Sigma$ is the set of *rational sets* on Σ. Each element of $\mathcal{L}_\Sigma$ is called a *rational set* △

The reader may wonder why Axiom (6) is assumed in our treatment, but is unnecessary in classical FA theory. The answer is because the axiom follows from other principles if Σ is finite.

In classical FA theory, where Σ is finite, a weaker axiom, $\{(a)\} \in \mathcal{L}_\Sigma \ \forall a \in \Sigma$ is assumed; *i.e.* a singleton set of containing a length-1 sequence is rational. The property analogous to Axiom (6) is thereafter a logical consequence of Axiom (7) and finite induction. This induction-based argument fails in our case when Σ is infinite.

Theorem 3.12. *If $A, B \in \mathcal{L}_\Sigma$, then $A \cap B \in \mathcal{L}_\Sigma$, and $A \setminus B \in \mathcal{L}_\Sigma$.*

The proof of Theorem 3.12 is beyond the scope of this article but involves arguing that (1) every rational language can be recognized by a σ-DFA, (2) every σ-DFA recognizes a rational language, (3) the synchronized cross product [18, Sec 4] may be used to produce a σ-DFA representing the intersection or relative complement of two rational languages.

Definition 3.13 (regular type expression, RTE)**.** A *regular type expression* or RTE, over Σ is an expression which represents a set of sequences on Σ. We may think of the RTE

as *generating* the possibly infinite set of finite sequences, or any element of the set as *matching* the RTE.

As a matter of notation, we take $S = [\![r]\!]$ to mean that the regular type expression, r, generates the set S.

If $A \subset \Sigma$, and f and g are *RTEs*, then the following are also RTEs.

Definition	Name	Generated Set
$\emptyset$	Empty	$[\![\emptyset]\!] = \emptyset$
ε	Unit	$[\![\varepsilon]\!] = \{()\}$
$\lfloor A \rfloor$	Singletons	$[\![\lfloor A \rfloor]\!] = \lfloor A \rfloor$
$f \cdot g$	Concatenation	$[\![f \cdot g]\!] = [\![f]\!] \cdot [\![g]\!]$
$f \vee g$	Alternation	$[\![f \vee g]\!] = [\![f]\!] \cup [\![g]\!]$
$f \wedge g$	Intersection	$[\![f \wedge g]\!] = [\![f]\!] \cap [\![g]\!]$
$\neg f$	Complement	$[\![\neg f]\!] = \Sigma^* \setminus [\![f]\!]$
f^n	Exponentiation	$[\![f^n]\!] = [\![f]\!]^n$
f^*	Kleene Star	$[\![f^*]\!] = [\![f]\!]^*$

Note that the complement RTE, $\neg f$, generates the set of *finite* sequences which fail to match f, rather than the set of all sequences failing to match f. △

Definition 3.14 (+ and ?)**.** As notation, we take f^+ to mean $f \cdot f^*$, and we take $f^?$ to mean $\varepsilon \vee f$. △

Theorem 3.15. *If $A \subset \Sigma^*$ is generated by some RTE, then $A \in \mathcal{L}_\Sigma$.*

The proof, which we omit for brevity, follows from Theorem 3.12 as well as Definitions 3.11 and 3.13.

4 Programming Language Supporting RTEs

In Section 2 we defined σ-DFAs. In Section 3 we defined rational sets and RTEs. In the sections which follow we describe the construction of σ-DFAs which recognize elements of rational sets designated by RTEs.

This section describes the abstractions we expect in a programming language in order to support the rational sets we intend to recognize. In particular we define a *simple type system* which is a type system sufficiently powerful to express rational sets as types, and also sufficiently flexible allow the construction of σ-DFAs to recognize them.

Suppose Σ is the set of values (sequences and otherwise) expressible in some programming language, and that Σ^* denotes the set of all sequences whose values come from Σ.

There is a notable difference between classical rational expressions and RTEs with regard to the relation of Σ and Σ^*. In classical FA theory, Σ is isomorphic ($\cong$) to a subset of Σ^*: Σ being the set of letters, which is isomorphic to the set of single letter words Σ^1; and Σ^* is the set of all words. This means that $\Sigma \cong \Sigma^1 \subset \Sigma^*$. However, in our theory of rational expressions over infinite alphabets, $\Sigma^* \subset \Sigma$.

4.1 A Simple Type System

Different programming languages make different assumptions about types. We assume a type system which behaves as Definition 4.1.

Definition 4.1 (*type*, *simple type space*, *simple type system*)**.** Let Σ denote the set of all values expressible in a given programming language. A *simple type space*, $\mathfrak{P}(\Sigma)$, is the set of sets of values from Σ. Each subset of Σ is called a *type*.

A *simple type system* is a simple type space, along with a set, Υ of *type designators*. Each $v \in \Upsilon$ designates a type denoted by $[\![v]\!]$.

In addition to some basic data types which differ from language to language, (*e.g.*, `number`, `string`, `bignum`), the set, Υ contains the following type designators.

Type Designator	Type Description
$\top$	$[\![\top]\!] = \Sigma$, the set of all values.
$\bot$	$[\![\bot]\!] = \emptyset$,
$\{a\}$	$[\![\{a\}]\!] = \{a\}$. The singleton type containing only $a \in \Sigma$
$\{a_1, a_2, ...a_n\}$	$[\![\{a_1, a_2, ...a_n\}]\!] = \{a_1, a_2, ...a_n\}$. The set containing precisely the designated elements: $a_1, a_2, ...a_n$.
sequence	$[\![sequence]\!] = \Sigma^*$.
$v_1 \cup v_2$	$[\![v_1 \cup v_2]\!] = [\![v_1]\!] \cup [\![v_2]\!]$
$v_1 \cap v_2$	$[\![v_1 \cap v_2]\!] = [\![v_1]\!] \cap [\![v_2]\!]$
$\overline{v}$	$[\![\overline{v}]\!] = \Sigma \setminus [\![v]\!]$

The type system provides two additional relations:

1. The operator $\in$ defines a relation between values and type designators. In particular if $v \in \Upsilon$, and $a \in [\![v]\!]$, then the relation $a \in v$ holds, otherwise $a \in \overline{v}$ holds.
2. There is a subset relation, $\subset$, between some types. However, the subset relation may not be decidable (see Section 4.2). In particular if $v_1, v_2 \in \Upsilon$, then $v_1 \subset v_2$ only if $[\![v_1]\!] \subset [\![v_2]\!]$ and $v_2 \subset v_1$ only if $[\![v_2]\!] \subset [\![v_1]\!]$. See Section 4.2 for more details. △

We do not specify how the programming language represents type designators: possibly as part of the language specification [26, Chapter 4] as in Common Lisp, or as an Algebraic Data Type (ADT) implemented as an add–on library. In `clojure-rte`, Section 7, we have represented type specifiers (called type designators) as s-expressions defined by a DSL (domain specific language).

The following examples suppose the programming language in question supports the types and subtype relations illustrated in Figure 2. Furthermore, we use notation such as $\mathit{integer} \cap (\overline{\mathit{integer}} \cup \mathit{fixnum}) \cap \overline{\mathit{fixnum}}$ to refer to logical combinations of those types.

4.2 Determining the Subtype Relation

We often need to decide whether two specified sets have a subset relation. In cases where it fails to be decidable, but recognized as such, the resulting automata will contain null-transitions. These redundant transitions may result in needless computation at compile-time and at run-time. Despite these null-transitions, the automata remain deterministic.

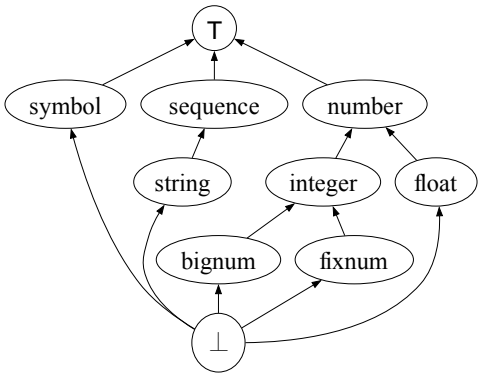


Figure 2. Built-in type hierarchy. Arrows point from subtype to supertype.

Grigore [9] addresses subtype decidability in Java [8]: deciding whether one type is a subtype of another is equivalent to the halting problem. Kennedy and Pierce [14] identify several restrictions which when placed on such a type system, make the question decidable. They investigate the effect of restricting these capabilities in different ways in Java [8], Scala [21, 22], C#, and .NET Intermediate Language. Curiously, [14] from Microsoft Research gives citations for Java and Scala, but not for C# and .NET Intermediate Language. Perhaps authors believe the reader is more familiar with C# and .NET IL than with other languages.

In Scala, the method <:< (see Section 8) attempts to compute the subtype relation, returning *true* if the subtype relation can be proven, and *false* otherwise. Thus, the Scala <:< conflates *cannot compute* with *computed to be false*.

Bonnaire [4] presents *typed Clojure*, but does not rigorously define a clojure *type*, and does not implement anything as ambitious as Common Lisp's `subtypep`. The clojure language offers an `isa?` predicate, which is simple and decidable and is ultimately based on `isAssignableFrom` which is provided by the JVM [8].

In Common Lisp [2] `SUBTYPEP` can be compute intensive in general. Even in cases where the subtype relation is decidable, a Common Lisp implementation is allowed to return *don't know* in cases deemed to be *too compute intensive*. [3, 28]

5 The Brzozowski Derivative

In Section 6 we will construct σ-DFAs based on RTEs. In this section we present the Brzozowski derivative which is the principle tool needed for such a construction.

There are several algorithms for constructing an FA (over a finite alphabet) from a given regular expression. One such algorithm was inspired by Ken Thompson [25, 30], has a notable limitation—it does not support regular expression intersection. The algorithm also has a practical limitation; it produces DFA which are not deterministic, thus requiring an additional step of determinizing. While determinizing σ-DFA has indeed been addressed in [29], we chose an approach which avoids the determinization step.

We have chosen an algorithm based on regular expression *derivatives*, which produces a deterministic DFA by construction. This algorithm was first presented in 1964 by Janusz Brzozowski [6]. While Brzozowski's result was applied to digital circuits, Scott Owens *et al.* [23] presented a *modern* version of the algorithm applied to regular pattern recognition for sequences of characters. We have generalized the Owens interpretation to be compatible with RTEs. Owens also noted that one of the practical obstacles of 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 of values rather than individual values. In this article, we present this generalization, making note of deviations from the Brzozowski/Owens algorithm.

5.1 Definition and Examples

In this section we define the *derivative* of a rational expression and give two examples. To understand the following definitions, the reader is reminded of Definitions 3.8 and 3.9, which define the *tail* and :: operators.

Definition 5.1 (primitive, $\frac{r}{A}$)**.** The *primitive* of an RTE, r with respect to a given type, A, denoted $\frac{r}{A}$, is defined as:

$$\frac{r}{A} = r \wedge (\lfloor A \rfloor \cdot \Sigma^*) \qquad \triangle$$

Definition 5.2 (tail, $\lceil S \rceil$)**.** If r is an RTE generating the set S, then $\lceil S \rceil$, called the *tail set of* S, is defined as: $\lceil S \rceil = \{t \in \Sigma^* \mid \exists h \in \Sigma \text{ such that } h :: t \in S\}$. We also say: $\lceil r \rceil$ is the RTE which generates the set $\lceil S \rceil$. $\triangle$

Definition 5.3 (derivative, $\partial_A r$)**.** If, r is a rational expression generating set S, and A is a type, the *derivative* of r *with respect to* A is defined as:

$$\partial_A r = \lceil \frac{r}{A} \rceil \tag{10}$$

The *derivative* of a set, S, of sequences with respect to a given type, A, denoted $\partial_A S$, is defined as the set of tails of sequences which start with a value from A.

$$\partial_A S = \{x \in \Sigma^* \mid h :: x \in S \text{ for some } h \in A\}.$$

The RTE, $\partial_A r$, is said to generate the set $\partial_A S$. $\triangle$

We may think of an RTE in terms of the set of sequences it *generates*. To give an intuition of what the RTE derivative is and how to calculate it, we offer Examples 5.4 and 5.5.

Example 5.4 (Example of RTE)**.** Let

$$S = \big\{(12\ 12),\ (13\ 12),\ (\text{"}hello\text{"}\ 12),\ (12\ 13\ 14\ 15),\ (\text{"}hello\text{"}\ 12\ 13\ 7.22),\ (13\ 12\ 11\ -1.13),\ (123456\ 12\ 13\ 7.22)\ \big\}.$$

$\frac{S}{integer}$ is the largest subset of S of sequences each of which begins with an integer:

$$\frac{S}{integer} = \{(12\ 12),\ (13\ 12),\ (12\ 13\ 14\ 15),\ (13\ 12\ 11\ -1.13),\ (123456\ 12\ 13\ 7.22)\}.$$

And finally, taking the tail of each sequence:

$$\partial_{integer} S = \lceil \frac{S}{integer} \rceil = \{(12),\ (13\ 14\ 15),\ (12\ 11\ -1.13),\ (12\ 13\ 7.22)\}.$$

△

Example 5.5 (Example of RTE)**.** We generalize Example 5.4. Consider the rational expression

$$r = (integer \vee string) \cdot fixnum^* \cdot number\,.$$

Let S denote the set of sequences generated by this RTE. Here, S is a superset of the S in Example 5.4. Each of these sequences has finite length, and each begins with exactly one object of type *integer* or exactly one object of type *string*, ends with exactly one object of type *number*, and between contains zero or more occurrences of objects of type *fixnum*.

Thus $\partial_{integer} S$ is the set generated by

$$\partial_{integer} r = fixnum^* \cdot number\,,$$

because $\frac{S}{integer}$ is the subset of S generated by

$$\frac{r}{integer} = integer \cdot fixnum^* \cdot number\,.$$

This set contains those sequences (elements of S) which start with an integer as opposed to a string.

Stripping off the head from each sequence in $\frac{S}{integer}$ results in the set $\lceil \frac{S}{integer} \rceil$, which is precisely the set generated by $fixnum^* \cdot number$. △

5.2 Nullability

We will address the derivative more precisely in Section 5.3, but we find it useful to first define nullability. As will be seen, to calculate the derivative of some rational expressions, we must calculate whether the rational expression is nullable.

Definition 5.6 (*nullable*)**.** A set of sequences is said to be *nullable* if it contains the empty sequence. If such a set is generated by an RTE, we say that the RTE itself is *nullable*.

The function ν (the Greek letter nu) calculates nullability. ν may be applied to any RTE, and evaluates to either $\emptyset$ or ε, according to the recursive rules in Figure 3. If and only if $\nu(r) = \varepsilon$, then r is said to be *nullable*. △

It may be worth reëmphasizing here that the symbol ε in Definition 5.6 and in Figure 3 is the rational expression ε not the empty sequence. As is common practice, Owens [23] used the notation, ε, to represent both the empty word and also the set consisting precisely of the empty word. We avoid this ambiguity by referring to the empty sequence as ().

$$\nu(\emptyset) = \emptyset \tag{11}$$
$$\nu(\varepsilon) = \varepsilon \tag{12}$$
$$\nu(r \vee s) = \nu(r) \ \vee\ \nu(s) \tag{13}$$
$$\nu(r \cdot s) = \nu(r) \ \wedge\ \nu(s) \tag{14}$$
$$\nu(r \wedge s) = \nu(r) \ \wedge\ \nu(s) \tag{15}$$
$$\nu(r^*) = \varepsilon \tag{16}$$
$$\nu(r^+) = \nu(r) \tag{17}$$
$$\nu(r^?) = \varepsilon \tag{18}$$
$$\nu(\neg r) = \begin{cases} \varepsilon & \text{if } \nu(r) = \emptyset \\ \emptyset & \text{if } \nu(r) = \varepsilon \end{cases} \tag{19}$$
$$\nu(a) = \emptyset \qquad \text{otherwise} \tag{20}$$

Figure 3. Recursive rules defining the nullability function ν

5.3 Computing the Brzozowski Derivative

Given an RTE, we would like to calculate the RTE representing its derivative. To do so, the reduction rules shown in Figure 4 can be recursively applied.

$$\partial_A \emptyset = \emptyset \tag{21}$$
$$\partial_A \varepsilon = \emptyset \tag{22}$$
$$\partial_A(r \ \vee\ s) = \partial_A r \vee \partial_A s \tag{23}$$
$$\partial_A(r \cdot s) = \begin{cases} \partial_A r \cdot s, & \text{if } \nu(r) = \emptyset \quad (24) \\ \partial_A r \cdot s \vee \partial_A s, & \text{if } \nu(r) = \varepsilon \quad (25) \\ \partial_A r \cdot s \vee \nu(r) \cdot \partial_A s, & \text{in either case} \quad (26) \end{cases}$$
$$\partial_A(r \ \wedge\ s) = \partial_A r \ \wedge\ \partial_A s \tag{27}$$
$$\partial_A(r^*) = \partial_A r \cdot r^* \tag{28}$$
$$\partial_A(r^+) = \partial_A(r^*) \tag{29}$$
$$\partial_A(r^?) = \partial_A(r) \tag{30}$$
$$\partial_\varepsilon r = r \tag{31}$$
$$\partial_A(\neg r) = \neg \partial_A r \tag{32}$$
$$\partial_A \lfloor B \rfloor = \varepsilon \quad \text{if } [\![A]\!] \subset [\![B]\!] \tag{33}$$
$$\partial_A \lfloor B \rfloor = \emptyset \quad \text{if } [\![A]\!] \parallel [\![B]\!] \tag{34}$$
$$\partial_A \lfloor B \rfloor \text{ is undefined otherwise.} \tag{35}$$

Figure 4. Recursive rules for Brzozowski derivatives extended to RTEs. The variables, A and B represent type designators, $[\![A]\!], [\![B]\!] \subset \Sigma$. And r and s represent RTEs, $r, s \in \mathcal{L}_\Sigma$.

Note that (26) is useful for theoretical and hand calculation but is problematic for algorithmic calculation. In the case that $\nu(r) = \emptyset$, (26) is equivalent to (24), but special attention is required because a naïve implementation of $\partial_A s$ may result

in an infinite recursion. To avoid such an error, (24) and (25) should be used instead of (26).

The rules in Figures 3 and 4 are similar to those Owens [23] mentions. We have omitted a rule from the treatment from Owens, which was $\partial_A b = \varepsilon$ whenever $a \neq b$. This rule does not hold when regular expressions are generalized to RTEs; we show a counter example in Example 6.3. An important insight we have added to the presentation of Owens is the introduction of Equations (33) and (34) which are proven in Section 6.3 as Theorems 6.5 and 6.7 respectively. Moreover, we have made Equation (35) explicit.

Although they are not completely necessary, we have added Equations (17), (18), (29), and (30) to Figures 3 and 4 as they often aid in computation, such as in Examples 6.1 and 6.2. These equations follow from Definition 3.14, and the other equations in Figures 3 and 4. The derivations are omitted here but may be found in [16, Sec 3.6].

6 Constructing the σ-DFA from an RTE

In the previous sections we have presented the tools necessary to construct an σ-DFA based on an RTE. In this section we present the construction algorithm which exploits the Brzozowski derivative as presented in Section 5.

Some approaches to construction of FA, such as [12, 25], involve constructing a DFA and thereafter determinizing it. We use the technique presented by Brzozowski [6] and clarified by Owens [23]. The algorithm uses a technique called the derivative, to construct a DFA, and thereby obviating the determinization step. In [16, 19], Newton *et al.* explain how the Brzozowski derivative can be extended to accommodate Common Lisp types, in particular rather than calculating the Brzozowski derivative (as Owens suggests) with respect to each letter of the alphabet, instead one must calculate the derivative with respect each type appearing in the maximal disjoint type decomposition as explained in [20].

6.1 Constructing States and Transitions

In order to determine whether a given sequence matches a particular RTE, we execute a σ-DFA with the sequence as input. Thus we must convert the RTE to a σ-DFA. This need only be done once and can often be done at compile time, depending the meta-programming capability of your programming language [17]. The flow involves our extension of the Brzozowski derivative algorithm, explained in Section 5.

The algorithm can be summarized as follows. Each state in the σ-DFA represents all the possible futures which are accepting. Moreover, there is a (not necessarily unique) RTE which designates that set of futures. An elaborate example starts in Example 6.1 and continues in Example 6.2.

There is a delicate matter when mapping a sequence of objects to a sequence of type designators: the mapping is not unique, and may lead to an NFA. We would like to avoid non-determinism, because the algebra associated with σ-DFAs is simpler, especially when involving intersection and complementation. We ignore this issue for the moment, but return to it Section 6.2.

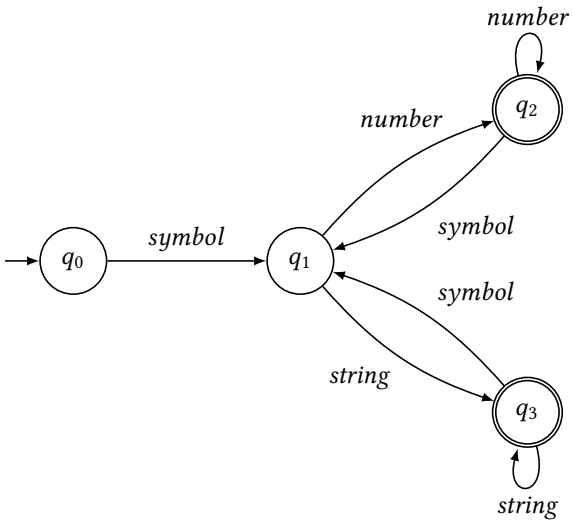


Figure 5. Example σ-DFA implementing the RTE: $(symbol \cdot (number^+ \vee string^+))^+$

Example 6.1 (Calculating the states and transitions σ-DFA for $(symbol \cdot (number^+ \vee string^+))^+$)**.**
Consider the RTE

$$q_0 = (symbol \cdot (number^+ \vee string^+))^+ . \tag{36}$$

We wish to construct a σ-DFA which recognizes sequences matching this pattern. Such a σ-DFA is shown in Figure 5. As the first step in this construction, we create a state q_0 corresponding to the given RTE, (36).

Next, we proceed to calculate the derivative with respect to each type designator mentioned in q_0. As will be seen, it suffices to differentiate with respect to the type designators which are permissible as the first element of the sequence. For example, the first element of the sequence is neither allowed to be a *string* nor a *number*. This is equivalent to saying that the corresponding derivatives are $\emptyset$: $\partial_{string} q_0 = \partial_{number} q_0 = \emptyset$ This set of types of first elements is precisely the *first types* from Definition 6.8.

Thus, as far as q_0 is concerned, we need only calculate one derivative: $\partial_{symbol} q_0$. To help understand this calculation, we show the steps explicitly.

To compress and simplify the computation, we abbreviate the types in questions: $S = symbol, N = number, R = string$.

$$\begin{aligned}\partial_{symbol} q_0 &= \partial_{symbol}\big((symbol \cdot (number^+ \vee string^+))^+\big)\\ &= \partial_S\big((S \cdot (N^+ \vee R^+))^+\big)\\ &= \partial_S\Big(S \cdot (N^+ \vee R^+)\Big) \cdot \Big(S \cdot (N^+ \vee R^+)\Big)^*\\ &= \big((\partial_S S) \cdot (N^+ \vee R^+) \ \vee \ \nu(S) \cdot \partial_S(N^+ \vee R^+)\big)\\ &\quad \cdot \Big(S \cdot (N^+ \vee R^+)\Big)^*\\ &= \big(\varepsilon \cdot (N^+ \vee R^+) \ \vee \ \emptyset \cdot \partial_S(N^+ \vee R^+)\big)\\ &\quad \cdot \ (S \cdot (N^+ \vee R^+))^*\\ q_1 &= (N^+ \vee R^+) \ \cdot \ \big(S \cdot (N^+ \vee R^+)\big)^*\end{aligned}$$

Since there is not yet a state in the automaton labeled with this expression, we create one named q_1. We also create a transition $q_0 \xrightarrow{symbol} q_1$, as shown in Figure 5. This transition is labeled with *symbol* because $\partial_{symbol} q_0 = q_1$. We now proceed to calculate the derivatives of q_1.

$$\begin{aligned}q_2 = \partial_{number} q_1 &= \partial_N\big((N^+ \vee R^+) \ \cdot \ \big(S \cdot (N^+ \vee R^+)\big)^*\big)\\ &= N^* \cdot (S \cdot (N^+ \vee R^+))^*\end{aligned}$$

We add q_2 to the automaton with transition $q_1 \xrightarrow{number} q_2$.

$$q_3 = \partial_{string} q_1 = R^* \cdot (S \cdot (N^+ \vee R^+))^*$$

We add q_3 to the automaton with transition $q_1 \xrightarrow{string} q_3$.

If we continue calculating the derivatives, we find that we again obtain expression we have already obtained in earlier steps. $\partial_{symbol} q_2 = q_1$ $\partial_{string} q_3 = q_3$ $\partial_{number} q_2 = q_2$ $\partial_{symbol} q_3 = q_1$ $\partial_{string} q_2 = q_1$ From these derivatives we create the following transitions thus completing the transitions in the state machine (Figure 5): $q_2 \xrightarrow{symbol} q_1$, $q_2 \xrightarrow{number} q_2$, $q_2 \xrightarrow{string} q_1$, $q_3 \xrightarrow{string} q_3$, and $q_3 \xrightarrow{symbol} q_1$. △

In Example 6.1, we have created all the states in the σ-DFA, and we have labeled all the transitions between states with the type designator which was used in the derivative calculation between those states. We have ignored transitions from any state to the $\emptyset$ state.

Finally, we label the *accepting* states. To determine which states are accepting states, we must determine which of the rational expressions are nullable. Example 6.2 extends Example 6.1 by calculating the accepting states.

Example 6.2 (Calculating the accepting states of σ-DFA for $(symbol \cdot (number^+ \vee string^+))^+$)**.**
The accepting states are q_2 and q_3, because only those two states are nullable. We know this intuitively, because q_2 and q_3 can match the empty sequence.

$$() \in [\![q_2]\!] = [\![N^* \cdot (S \cdot (N^+ \ \vee \ R^+))^*]\!]$$
$$() \in [\![q_3]\!] = [\![R^* \cdot (S \cdot (N^+ \ \vee \ R^+))^*]\!]$$

Despite our intuition, we may use ν to compute the nullability of q_0, q_1, q_2, and q_3.

$$\begin{aligned}\nu(q_0) &= \nu((S \cdot (N^+ \vee R^+))^+) = \nu(S \cdot (N^+ \vee R^+))\\ &= \emptyset \wedge \nu(N^+ \vee R^+) = \emptyset\end{aligned}$$

So q_0 is not nullable.

$$\begin{aligned}\nu(q_1) &= \nu\big((N^+ \vee R^+) \ \cdot \ (S \cdot (N^+ \vee R^+))^*\big)\\ &= \nu(N^+ \vee R^+) \ \wedge \ \nu((S \cdot (N^+ \vee R^+))^*)\\ &= \nu(N^+ \vee R^+) \wedge \varepsilon\\ &= \nu(N^+ \vee R^+) \quad ; \text{For } \nu(r) \in \{\emptyset, \varepsilon\}\\ &= \nu(N^+) \vee \nu(R^+)\\ &= \nu(N) \vee \nu(R) \quad = \emptyset \vee \emptyset = \emptyset\end{aligned}$$

So q_1 is not nullable. In similar manner, we find that none other of the expressions is nullable.

$$\begin{aligned}\nu(q_2) &= \nu\big(N^* \cdot (S \cdot (N^\vee + R^+))^*\big)\\ &= \nu(N^*) \wedge \nu\big((S \cdot (N^+ \vee R^+))^*\big) = \varepsilon \wedge \varepsilon = \varepsilon\end{aligned}$$

So q_2 is nullable.

$$\begin{aligned}\nu(q_3) &= \nu\big(R^* \cdot (S \cdot (N^+ \vee R^+))^*\big)\\ &= \nu(R^*) \wedge \nu\big((S \cdot (N^+ \vee R^+))^*\big) = \varepsilon \wedge \varepsilon = \varepsilon\end{aligned}$$

So q_3 is nullable. △

6.2 Dealing with Overlapping Types

In the examples in Section 6.1, all the types considered exiting each state were disjoint. If the same method is naïvely used with types which are intersecting, the resulting σ-DFA will not be a valid representation of the rational expression, as shown in Example 6.3.

Example 6.3 (Counterexample disproving $a \neq b \implies \partial_A b = \varepsilon$.)**.** Consider the rational expression involving the intersecting types *integer* and *number*:

$$q_0 = \big((number \cdot integer) \ \vee \ (integer \cdot number)\big) .$$

The sequences which match this expression are sequences of two numbers, at least one of which is an integer. Unfortunately, when we calculate $\partial_{number} q_0$ and $\partial_{integer} q_0$ we arrive at a different result. To compress and simplify the computation, we abbreviate the types in questions: $Z = integer, N = number$.

$$\begin{aligned}\partial_N q_0 &= \partial_N\big((N \cdot Z) \vee (Z \cdot N)\big) = \partial_N(N \cdot Z) \vee \partial_N(Z \cdot N)\\ &= (\partial_N N) \cdot Z \vee (\partial_N Z) \cdot N = \varepsilon \cdot Z \vee \emptyset \cdot N = Z \vee \emptyset = Z\end{aligned}$$

Without continuing to calculate all the derivatives, it is already clear that this result is wrong. If we start with the set of sequences of two numbers one of which is an integer, and out of that find the subset of sequences starting with a number, we get back the entire set. The set of tails of this set is *not* the set of singleton sequences of integer as this naïve calculation of $\partial_{number} q_0$ predicts. △

The problem demonstrated in Example 6.3 is that if an RTE is treated blindly as an ordinary rational expression, then $number \neq integer$ end of story. But if we wish to create a σ-DFA which will allow validation of rationally typed sequences of objects, we must extend the theory slightly to accommodate intersecting types.

To address this problem, we have augmented the algorithm of Brzozowski with an additional step. Rather than calculating the derivative at each state with respect to each type mentioned in the regular type expression, some of which might be overlapping, instead we calculate a disjoint set of types. Recall from Equation (35) from Figure 4 that the derivative, $\partial_A B$, is only defined for types A and B, when $A \parallel B$ or $A \subset B$. More specifically, given a set $\mathcal{A}$ of potentially overlapping types, we calculate a set $\mathcal{B}$ which has the properties: Each element of $\mathcal{B}$ is a subtype of some element of $\mathcal{A}$, any two elements $\mathcal{B}$ are disjoint from each other, and $\cup\mathcal{A} = \cup\mathcal{B}$. This deconstruction as the Maximal Disjoint Type Decomposition (MDTD) and is discussed in depth in [16].

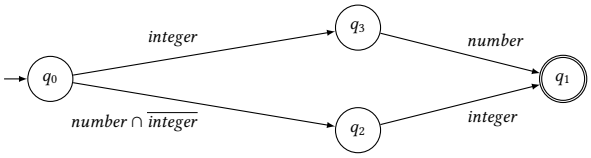


Figure 6. Example σ-DFA with disjoint types

Figure 6 illustrates a σ-DFA whose transitions are based on such a disjoint union. The set of overlapping types $\mathcal{A} = \{number, integer\}$ has been replaced with the set of disjoint types $\mathcal{B} = \{number \cap \overline{integer}, integer\}$.

This extra step has two positive effects on the algorithm. 1) it ensures that the constructed automaton is deterministic, *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.

To use these differentiation rules on expressions containing partially overlapping types we must only differentiate a given rational expression with respect to disjoint types. Figure 6 shows an σ-DFA expressing the rational expression $q_0 = ((number \cdot integer) \wedge (integer \cdot number))$ but only using types for which the derivative is defined. $q_3 = \partial_{integer} q_0$ and $q_1 = \partial_{number \cap \overline{integer}} q_0$. Figure 6 does show transitions from $q_3 \xrightarrow{number} q_2$ and $q_1 \xrightarrow{integer} q_2$ using intersecting types. This is not, however, a violation of the rules in Figure 4 because q_3 and q_1 are different states.

6.3 Proofs of Derivatives with Set Relations

Lemma 6.4 simplifies the proofs of Theorems 6.5 and 6.7, which in turn justify the claims in Equations (33) and (34).

Lemma 6.4. $\partial_A \lfloor B \rfloor = \lceil \lfloor B \rfloor \wedge \lfloor A \rfloor \rceil$

Proof. $\partial_A \lfloor B \rfloor = \lceil \frac{\lfloor B \rfloor}{A} \rceil = \lceil \lfloor B \rfloor \wedge (\lfloor A \rfloor \cdot \Sigma^*) \rceil$, by Definitions 5.3 and 5.1.

$\lfloor B \rfloor \wedge (\lfloor A \rfloor \cdot \Sigma^*) = \lfloor B \rfloor \wedge \lfloor A \rfloor$, because the elements of $\lfloor B \rfloor \wedge (\lfloor A \rfloor \cdot \Sigma^*)$ are those sequences which are simultaneously in $\lfloor B \rfloor$, *i.e.* sequences (h_1) where $h_1 \in B$ and also in $\lfloor A \rfloor \cdot \Sigma^*$, *i.e.* sequences of the form $h_2 :: s$ where $h_2 \in A$. To be in both sets, a sequence must be a singleton (h) with $h \in A$ and also $h \in B$, which is precisely $\lfloor B \rfloor \wedge \lfloor A \rfloor$. □

Theorem 6.5. *If $A \neq \emptyset$ and $A \subset B$ then $\partial_A \lfloor B \rfloor = \varepsilon$.*

Proof. By Lemma 6.4, $\partial_A \lfloor B \rfloor = \lceil \lfloor B \rfloor \wedge \lfloor A \rfloor \rceil$. Since $A \subset B$ we have $\lfloor A \rfloor \subset \lfloor B \rfloor$, so $\lfloor B \rfloor \wedge \lfloor A \rfloor = \lfloor A \rfloor$. Since $A \neq \emptyset$, $\lfloor A \rfloor$ is a nonempty set of singleton sets; the tail of each being $()$. Therefore $\lceil \lfloor A \rfloor \rceil = \{()\} = \varepsilon$ □

Corollary 6.6. $\partial_A \lfloor A \rfloor = \varepsilon$.

Proof. $A \subset A$, so let $B = A$, and apply Theorem 6.5. □

Theorem 6.7. *If $A \parallel B$, then $\partial_A \lfloor B \rfloor = \emptyset$.*

Proof. $\lfloor B \rfloor \wedge \lfloor A \rfloor = \emptyset$, because if we suppose $x \in \lfloor B \rfloor \wedge \lfloor A \rfloor$, then x is of the form $h :: ()$ where $x \in B$ and $x \in A$, which is impossible since $A \parallel B$. Thus $\emptyset = \lceil \emptyset \rceil = \lceil \lfloor B \rfloor \wedge \lfloor A \rfloor \rceil = \partial_A \lfloor B \rfloor$, by Lemma 6.4. □

6.4 Practical Optimizations

There are a couple of useful optimization steps.

If the derivative is $\emptyset$, there is no reason to add the state to the σ-DFA, lest we clutter the graphical representation with arrows leading to this sink state.

It is not necessary that there will be 1:1 correspondence between the non-trivial derivatives and the states. The problem is that reducing the rational expressions to a canonical form is a hard problem, since many rational expressions may generate the same rational language. Even so, one would expect that there might be one *canonical reduced* expression which could be arrived at, given a finite set of identities such as $\emptyset \vee L = L = L \vee \emptyset$, $\varepsilon \cdot L = L = L \cdot \varepsilon$, $(L^*)^* = L^*$, $L \vee K = K \vee L$, *etc.* In fact, there is no finite set of identities which permits to deduce all identities between rational expressions [24].

Owens *et al.* [23] noted that the problem of determining whether two regular expressions are equivalent is expensive. In fact, deciding language equality for regular expressions with intersection and complement operators is of non-elementary complexity [1]. According to Owens, it suffices to allow the same derivative in different algebraic forms to be represented by multiple states as long as it is a reasonable number. Owens suggests a weak form of regular expression equivalence, based on Brzozowski's work. In summary, we perform a reduction step in the derivative calculation to limit the number of forms expressed, but the reduction need not fully reduce every expression to a canonical form.

6.5 Algorithm for Brzozowski Applied to RTEs

To compute the automaton corresponding to a rational expression [23], use Algorithm 1. The algorithm assumes the existence of a constructor function `State` (which creates a state given a rational expression), and requires the computation of $1^{st}()$ of a rational expression according to Definition 6.8. Brzozowski argued that this procedure terminates because there is only a finite number of derivatives possible, modulo multiple equivalent algebraic forms. Eventually, all the expressions encountered will be algebraically equivalent to the derivative of some other expression in the set.

Definition 6.8 ($1^{st}(f)$)**.** The set of *first types* of an RTE, f, denoted $1^{st}(f)$, is calculated as follows:

$$1^{st}(\emptyset) = 1^{st}(\varepsilon) = \emptyset$$
$$1^{st}(\lfloor A \rfloor) = \{A\}$$
$$1^{st}(r \vee s) = 1^{st}(r \wedge s) = 1^{st}(r) \cup 1^{st}(s)$$
$$1^{st}(r \cdot s) = \begin{cases} 1^{st}(r) \cup 1^{st}(s) & \text{if } \nu(r) = \varepsilon \\ 1^{st}(r) & \text{if } \nu(r) = \emptyset \end{cases}$$
$$1^{st}(\neg r) = 1^{st}(r)$$
$$1^{st}(r^*) = 1^{st}(r^+) = 1^{st}(r^?) = 1^{st}(r)$$

△

7 Sample Implementations

In the previous sections we have presented σ-DFAs (Section 2), RTEs (Section 3), and an algorithm (Section 6) to construct a σ-DFA from an RTE. In this section we briefly describe two open source implementations of RTE, one in Common Lisp [2] and one in Clojure [10, 11].

The reference implementation is a Common Lisp library, `rte`, available from [address withheld for anonymity] and available from Quicklisp quicklisp.org. The second, and more recent, `clojure-rte`, is an implementation in Clojure, clojure.org, and is available at [address withheld because of anonymity] The `rte` implementation in Common Lisp is the subject of several existing publications, including [WITHHELD] [16, 17, 19].

The `clojure-rte` implementation is a result of generalizing the Common Lisp work and uses the technique presented in the current article. Clojure runs atop the JVM, and the Clojure type system is based on the type system of Java. We have extended Clojure's type system, via a user library, sufficiently to implement Algorithm 1 and consequently to compute first types, Definition 6.8, as well as nullability from Figure 3 and the Brzozowski derivative from Figure 4.

The Common Lisp version, builds the σ-DFA at compile time and serializes it to low level machine instructions. It is unclear whether similar low-level optimizations can be realized using the meta-programming APIs of other dynamic languages, particularly those implemented on the JVM.

Algorithm 1: Compute DFA by Brzozowski derivative

Input: r : a rational type expression
Output: Components of a σ-DFA as in Definition 2.2.

begin
  $q_0 \leftarrow State(r)$ `// one initial state`
  $\delta \leftarrow ()$ `// empty list of triples`
  $Q \leftarrow \{q_0\}$ `// set of states`
  $W \leftarrow \{q_0\}$ `// working list states`
  while $W \neq \emptyset$ **do**
    $q_1 \leftarrow$ any element from W
    $W \leftarrow W \setminus \{q_1\}$
    for $a \in MDTD(\top :: 1^{st}(q_1.expression))$ **do**
      $r \leftarrow q_1.expression$
      $d \leftarrow \partial_a r$
      Reduce d to canonical form
      if $d \neq \emptyset$ **then**
        if $\exists\, q_2 \in Q$ *such that* $q_2.expression = d$ **then**
          $\delta \leftarrow (q_1, a, q_2) :: \delta$ `// transition`
        else
          $q_2 \leftarrow State(d)$
          $\delta \leftarrow (q_1, a, q_2) :: \delta$
          $W \leftarrow q_2 :: W$
          $Q \leftarrow q_2 :: Q$
  $F \leftarrow \{q \in Q \mid \nu(q.expression) = \varepsilon\}$
  return (Q, q_0, F, δ)

8 Conclusion and Perspectives

In this article we have presented a theoretical foundation for implementing RTEs in dynamically typed programming languages. We have presented an algorithm for computing the construction of σ-DFAs for recognizing such, given a sufficiently expressive type system. Reference implementations in Common Lisp and Clojure are available at [WITHHELD] and [WITHHELD] .

While details of the construction for the Common Lisp implementation may be found in [16, 19], we have not explained the technique used to encapsulate/disguise the Clojure type system into a simple type system as described in Section 4.1. We consider this a matter for future publication, in particular computation of subtype-ness, disjointness, and MDTD. Prior to such a future publication, interested readers are invited to peruse source code of `clojure-rte`.

Scala is not generally considered a dynamic language, but it does exhibit some dynamic features: REPL, runtime reflection via the `scala.reflect.runtime.universe` library or even the lower-level Java reflection. A Scala program may inquire at run-time about the types of object and invoke run-time logic as a function of subtype-ness via the methods `<:<` and `isAssignableFrom`. However, it is a matter of

active, ongoing research as to the extent to which Scala provides sufficient reflective capacity for implementing RTE, especially in light of type-erasure [5]. Furthermore, even if such implementation is possible it is also a matter of ongoing research whether such implementation could be useful or culturally acceptable in a statically typed language.

We have not addressed in this article how to *interpret* a σ-DFA at run-time to determine whether a given sequence matches the regular type expression.

Our σ-DFA construction is the first step of a more complete flow which includes combining and merging σ-DFAs to calculate Boolean combinations (union, intersection, etc) of rational typed languages. Once constructed, the σ-DFA may not be in minimal form. Another topic of research is minimization techniques of such σ-DFAs.

As mentioned in Section 3, the proof of Theorem 3.12 has been omitted from this article. We would like to present that proof at a later time in conjunction with the presentation of Boolean combinations of σ-DFAs.

References

[1] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974. ISBN 0201000296.

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

[3] Henry G. Baker. A Decision Procedure for Common Lisp's SUBTYPEP Predicate. *Lisp and Symbolic Computation*, 5(3):157–190, 1992. URL http://dblp.uni-trier.de/db/journals/lisp/lisp5.html#Baker92a.

[4] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure, 2018.

[5] Gilad Bracha. Generics in the java programming language. July 5, 2004. URL http://www.cs.rice.edu/~cork/312/Readings/GenericsTutorial.pdf.

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

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

[8] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390069X, 9780133900699.

[9] Radu Grigore. Java generics are turing complete. *CoRR*, abs/1605.05274, 2016. URL http://arxiv.org/abs/1605.05274.

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

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

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

[13] Lionel Parreaux Jim Newton, Sébastien Doeraene. Union types in scala 3, feb 2020. URL https://contributors.scala-lang.org/t/union-types-in-scala-3/4046.

[14] Andrew Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, January 2007. URL https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/.

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

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

[17] Jim Newton and Didier Verna. Recognizing hetergeneous sequences by rational type expression. In *Proceedings of the Meta'18: Workshop on Meta-Programming Techniques and Reflection*, Boston, MA USA, November 2018.

[18] Jim Newton and Didier Verna. Finite automata theory based optimization of conditional variable binding. In *European Lisp Symposium*, Genova, Italy, April 2019.

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

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

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

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

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

[24] Jean-Éric Pin. Mathematical Foundations of Automata Theory. 2015. URL http://www.liafa.jussieu.fr/~jep/PDF/MPRI/MPRI.pdf.

[25] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, USA, 2009. ISBN 0521844258.

[26] Guy L Steele Jr, Scott E Fahlman, Richard P Gabriel, David A Moon, Daniel L Weinreb, Daniel G Bobrow, Linda G DeMichiel, Sonya E Keene, Gregor Kiczales, Crispin Perdue, et al. *Common LISP: The Language (2nd Ed.)*. Digital Press, Bedford, Massachusetts, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[27] R.J. Trudeau. *Introduction to Graph Theory*. Dover Books on Mathematics. Dover Publications, 2013. ISBN 9780486318660. URL https://books.google.fr/books?id=eRLEAgAAQBAJ.

[28] Léo Valais, Jim Newton, and Didier Verna. Implementing baker's SUBTYPEP decision procedure. In *12th European Lisp Symposium*, Genova, Italy, April 2019.

[29] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. Symbolic automata constraint solving. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 640–654, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16242-8.

[30] Guangming Xing. Minimized Thompson NFA. *Int. J. Comput. Math.*, 81(9):1097–1106, 2004. URL http://dblp.uni-trier.de/db/journals/ijcm/ijcm81.html#Xing04.