Type-Checking of Heterogeneous Sequences in Common Lisp

Jim E Newton, Akim E Demaille, Didier E Verna

Type-Checking of Heterogeneous Sequences in Common Lisp

Jim E. Newton
jnewton@lrde.epita.fr

Akim Demaille
akim@lrde.epita.fr

Didier Verna
didier@lrde.epita.fr

EPITA/LRDE
14-16 rue Voltaire
F-94270 Le Kremlin-Bicêtre
France

ABSTRACT

We introduce the abstract concept of *rational type expression* and show its relationship to rational language theory. We further present a concrete syntax, *regular type expression*, and a Common Lisp implementation thereof which allows the programmer to declaratively express the types of heterogeneous sequences in a way which is natural in the Common Lisp language. The implementation uses techniques well known and well founded in rational language theory, in particular the use of the Brzozowski derivative and deterministic automata to reach a solution which can match a sequence in linear time. We illustrate the concept with several motivating examples, and finally explain many details of its implementation.

CCS Concepts

•**Theory of computation → Regular languages;** *Automata extensions;* •**Software and its engineering → Data types and structures;** *Source code generation;*

Keywords

Rational languages, Type checking, Finite automata

1. INTRODUCTION

In Common Lisp [15] a *type* is identically a set of (potential) values at a particular point in time during the execution of a program [15, Section 4.1]. Information about types provides clues for the compiler to make optimizations such as for performance, space (image size), safety or debuggability [10, Section 4.3] [15, Section 3.3]. Application programmers may as well make explicit use of types within their programs, such as with `typecase`, `typep`, `the` etc.

While the programmer can specify a homogeneous type for all the elements of a vector [15, Section 15.1.2.2], or the type for a particular element of a list, [15, System Class CONS], two notable limitations, which we address in this article, are 1) that there is no standard way to specify heterogeneous types for different elements of a vector, 2) neither is there a standard way to declare types (whether heterogeneous or homogeneous) for all the elements of a list.

We introduce the concept of *rational type expression* for abstractly describing patterns of types within sequences. The concept is envisioned to be intuitive to the programmer in that it is analogous to patterns described by regular expressions, which we assume the reader is already familiar with.

Just as the characters of a `string` may be described by a rational expression such as $(a \cdot b^* \cdot c)$, which intends to match strings such as `"ac"`, `"abc"`, and `"abbbbc"`, the rational type expression $(\textit{string} \cdot \textit{number}^* \cdot \textit{symbol})$ is intended to match vectors like `#("hello" 1 2 3 world)` and lists like `("hello" world)`. Rational expressions match character constituents of strings according to character equality. Rational type expressions match elements of sequences by element type.

We further introduce an s-expression based syntax, called *regular type expression* to encode a *rational type expression.* This syntax replaces the infix and post-fix operators in the rational type expression with prefix notation based s-expressions. The regular type expression `(:1 string (:* number) symbol)` corresponds to the rational type expression $(\textit{string} \cdot \textit{number}^* \cdot \textit{symbol})$. In addition, we provide a parameterized type named **`rte`**, whose argument is a regular type expression. The members of such a type are all sequences matching the given regular type expression. Section 2 describes the syntax.

While we avoid making claims about the potential utility of declarations of such a type from the compiler's perspective [1], we do suggest that a declarative system to describe patterns of types within sequences has great utility for program logic, code readability, and type safety.

A discussion of the theory of rational languages in which our research is grounded, may be found in [9, Chapters 3,4]. This article avoids many details of the theory, and instead emphasizes examples of problems this approach helps to solve and explains a high level view of its implementation. A more in-depth report of the research including the source code is provided in [11].

2. THE REGULAR TYPE EXPRESSION

We have implemented a parameterized type named **`rte`** (regular type expression), via `deftype`. The argument of

`:*`	match zero or more times.
`:+`	match one or more times.
`:?`	match zero or one time.
`:1`	match exactly once.
`:or`	match any of the regular type expressions.
`:and`	match all of the regular type expressions.

Table 1: Regular type expression keywords

rte is a *regular type expression.*

A *regular type expression* is defined as either a Common Lisp type specifier, such as `number`, `(cons number)`, `(eql 12)`, or `(and integer (satisfies oddp))`, or a list whose first element is one of a limited set of keywords shown in Table 1, and whose trailing elements are other regular type expressions.

As a counter example, `(rte (:1 (number number)))` is invalid because `(number number)` is neither a valid Common Lisp type specifier, nor does it begin with a keyword from Table 1. Here are some valid examples.

`(rte (:1 number number number))`
: corresponds to the rational type expression $(number \cdot number \cdot number)$ and matches a sequence of exactly three numbers.

`(rte (:or number (:1 number number number)))`
: corresponds to $(number + (number \cdot number \cdot number))$ and matches a sequence of either one or three numbers.

`(rte (:1 number (:?  number number)))`
: corresponds to $(number \cdot (number \cdot number)^?)$ and matches a sequence of one mandatory number followed by exactly zero or two numbers. This happens to be equivalent to the previous example.

`(rte (:* cons number))`
: corresponds to $(cons \cdot number)^*$ and matches a sequence of a `cons` followed by a `number` repeated zero or more times, *i.e.*, a sequence of even length.

The **rte** type can be used anywhere Common Lisp expects a type specifier as the following examples illustrate. The `point` slot of the class `C` expects a sequence of two numbers, *e.g.*, `(1 2.0)` or `#(1 2.0)`.

```
(defclass C ()
  ((point :type (and list (rte (:1 number number))))
   ...))
```

The Common Lisp type specified by `(cons number)` is the set of non-empty lists headed by a number, as specified in [15, System Class CONS]. The `Y` argument of the function `F` must be a possibly empty sequence of such objects, because it is declared as type `(rte (:* (cons number)))`. *E.g.*, `#((1.0) (2 :x) (0 :y "zero"))`.

```
(defun F (X Y)
  (declare
    (type (rte (:* (cons number)))
          Y))
  ...)
```

3. APPLICATION USE CASES

The following subsections illustrate some motivating examples of regular type expressions.

```
lambda-list :=
  (var*
   [&optional
     {var | (var [init-form [supplied-p-parameter]])}*]
   [&rest var]
   [&key {var | ({var | (keyword-name var)}
                 [init-form [supplied-p-parameter]]) }*
         [&allow-other-keys]]
   [&aux {var | (var [init-form])}*]
   )
```

Figure 1: CL specification syntax for the ordinary lambda list

3.1 RTE based string regular expressions

Since a string in Common Lisp is a sequence, the **rte** type may be used to perform simple string regular expression matching. Our tests have shown that the **rte** based string regular expression matching is about 35% faster than CL-PPCRE [16] when restricted to features strictly supported by the theory of rational languages, thus ignoring CL-PPCRE features such as character encoding, capture buffers, recursive patterns, etc.

The call to the function `remove-if-not`, below, filters a given list of strings, retaining only the ones that match an implicit regular expression `"a*Z*b*"`. The function, `regexp-to-rte` converts a string regular expression to a regular type expression.

```
(regexp-to-rte "(ab)*Z*(ab)*")
==>
  (:1 (:* (member #\a #\b))
      (:* (eql #\Z))
      (:* (member #\a #\b)))

(remove-if-not
  (lambda (str)
    (typep str
      `(rte ,(regexp-to-rte "(ab)*Z*(ab)*"))))
  '("baZab"
    "ZaZabZbb"
    "aaZbbbb"
    "aaZZZZbbbb"))
==>
("baZab"
 "aaZbbbb"
 "aaZZZZbbbb")
```

The `regexp-to-rte` function does not attempt the daunting task of fully implementing Perl compatible regular expressions as provided in CL-PPCRE. Instead `regexp-to-rte` implements a small but powerful subset of CL-PPCRE whose grammar is provided by [4]. Starting with this context free grammar, we use `CL-Yacc` [5] to parse a string regular expression and convert it to a regular type expression.

3.2 DWIM lambda lists

As a complex yet realistic example we use a regular type expression to test the validity of a Common Lisp lambda list, which are sequences which indeed are described by a pattern.

Common Lisp specifies several different kinds of lambda lists, used for different purposes in the language. *E.g.*., the *ordinary lambda list* is used to define lambda functions, the *macro lambda list* is for defining macros, and the *destructuring lambda list* is for use with `destructuring-bind`. Each of these lambda lists has its own syntax, the simplest of which is the *ordinary lambda list* (Figure 1). The following code shows examples of ordinary lambda lists which obey the specification but may not mean what you think.

```
(defun F1 (a b &key x &rest other-args)
  ...)

(defun F2 (a b &key ((Z U) nil u-used-p))
  ...)
```

The function `F1`, according to careful reading of the Common Lisp specification, is a function with three keyword arguments, `x`, `&rest`, and `other-args`, which can be referenced at the call site with a bizarre function calling syntax such as `(F1 1 2 :x 3 :&rest 4 :other-args 5)`. What the programmer probably meant was one keyword argument named `x` and an `&rest` argument named `other-args`. According to the Common Lisp specification [15, Section 3.4.1], in order for `&rest` to have its normal *rest-args* semantics in conjunction with `&key`, it must appear before not after the `&key` lambda list keyword. The specification makes no prevision for `&rest` following `&key` other than that one name a function parameter and the other have special semantics. This issue is subtle. In fact, SBCL considers this such a bizarre situation that it diverges from the specification and flags a `SB-INT:SIMPLE-PROGRAM-ERROR` during compilation: `misplaced &REST in lambda list: (A B &KEY X &REST OTHER-ARGS)`

The function `F2` is defined with an *unconventional* `&key` parameter which is not a symbol in the `keyword` package but rather in the current package. Thus the parameter `U` must be referenced at the call-site as `(F2 1 2 'Z 3)` rather than `(F2 1 2 :Z 3)`.

These situations are potentially confusing, so we define what we call the *dwim ordinary lambda list.* Figure 2 shows an implementation of the type `dwim-ordinary-lambda-list`. A Common Lisp programmer might want to use this type as part of a code-walker based checker. Elements of this type are lists which are indeed valid lambda lists for `defun`, even though the Common Lisp specification allows a more relaxed syntax.

The *dwim ordinary lambda list* differs from the *ordinary lambda list*, in the aspects described above and also it ignores semantics the particular lisp implement may have given to additional lambda list keywords. It only supports semantics for: `&optional`, `&rest`, `&key`, `&allow-other-keys`, and `&aux`.

3.3 destructuring-case

```
(defun F3 (obj)
  (typecase obj
    ((rte (:1 symbol (:+ (eql :count) integer)))
     (destructuring-bind (name &key (count 0)) obj
        ...))
    ((rte (:1 symbol list (:* string)))
     (destructuring-bind (name data
                          &rest strings) obj
        ...))))
```

Notice in the code above that each `rte` clause of the `typecase` includes a call to `destructuring-bind` which is related and hand coded for consistency. The function `F3` is implemented such that the object being destructured is certain to be of the format expected by the corresponding destructuring lambda list.

We provide a macro `destructuring-case` which combines the capability of `destructuring-bind` and `typecase`. Moreover, `destructuring-case` constructs the `rte` type specifiers in an intelligent way, taking into account not only the structure of the destructuring lambda list but also any given type declarations.

```
(deftype var ()
  `(and symbol
        (not (or keyword
                 (member t nil)
                 (member ,@lambda-list-keywords)))))

(deftype dwim-ordinary-lambda-list ()
  (let* ((optional-var
          '(:or var
                (:and list
                  (rte
                    (:1 var
                        (:? t
                            (:? var)))))))
         (optional
          `(:1 (eql &optional)
               (:* ,optional-var)))
         (rest '(:1 (eql &rest) var))
         (key-var
          '(:or var
                (:and list
                   (rte (:1
                          (:or var
                             (cons keyword
                                (cons var
                                      null)))
                          (:? t
                              (:? var)))))))
         (key
          `(:1 (eql &key)
               (:* ,key-var)
               (:?
                 (eql &allow-other-keys))))
         (aux-var
          '(:or var
                 (:and list
                    (rte (:1 var (:? t))))))
         (aux `(:1 (eql &aux)
                   (:* ,aux-var))))
    `(rte
      (:1 (:* var)
          (:? ,optional)
          (:? ,rest)
          (:? ,key)
          (:? ,aux)))))
```

Figure 2: The `dwim-ordinary-lambda-list` type

```
(defun F4 (obj)
  (destructuring-case obj
    ((name &key count)
     (declare (type symbol name)
              (type integer count))
     ...)
    ((name data &rest strings)
     (declare (type name symbol)
              (type data list)
              (type strings
                    (rte (:* string))))
     ...)))
```

This macro is able to parse any valid destructuring lambda list and convert it to a regular type expression. Supported syntax includes `&whole`, `&optional`, `&key`, `&allow-other-keys`, `&aux`, and recursive lambda lists such as:

```
(&whole llist a (b c)
 &key x ((:y (c d)) '(1 2))
 &allow-other-keys)
```

A feature of `destructuring-case` is that it can construct regular type expressions much more complicated than would be practical by hand. Consider the following example which includes two destructuring lambda lists, whose computed regular type expressions pretty-print to about 20 lines each. An example of the regular type expression matching `Case-1` is shown in Figure 3.

```
(:1 (:1 fixnum (:and list (rte (:1 fixnum fixnum))))
    (:and
      (:* keyword t)
      (:or
        (:1 (:? (eql :x) symbol (:* (not (member :y :z)) t))
            (:? (eql :y) string (:* (not (eql    :z))    t))
            (:? (eql :z) list   (:* t t)))
        (:1 (:? (eql :y) string (:* (not (member :x :z)) t))
            (:? (eql :x) symbol (:* (not (eql    :z))    t))
            (:? (eql :z) list   (:* t t)))
        (:1 (:? (eql :x) symbol (:* (not (member :y :z)) t))
            (:? (eql :z) list   (:* (not (eql    :y))    t))
            (:? (eql :y) string (:* t t)))
        (:1 (:? (eql :z) list   (:* (not (member :x :y)) t))
            (:? (eql :x) symbol (:* (not (eql    :y))    t))
            (:? (eql :y) string (:* t t)))
        (:1 (:? (eql :y) string (:* (not (member :x :z)) t))
            (:? (eql :z) list   (:* (not (eql    :x))    t))
            (:? (eql :x) symbol (:* t t)))
        (:1 (:? (eql :z) list   (:* (not (member :x :y)) t))
            (:? (eql :y) string (:* (not (eql    :x))    t))
            (:? (eql :x) symbol (:* t t))))))
```

Figure 3: Regular type expression matching destructuring lambda list Case-1

```
(destructuring-case data

  ;; Case-1
  ((&whole llist
     a (b c)
     &rest keys
     &key x y z
     &allow-other-keys)
   (declare (type fixnum a b c)
            (type symbol x)
            (type string y)
            (type list z))
   ...)

  ;; Case-2
  ((a (b c)
   &rest keys
   &key x y z)
   (declare (type fixnum a b c)
            (type symbol x)
            (type string y)
            (type list z))
   ...))
```

4. IMPLEMENTATION OVERVIEW

Using an **rte** involves several steps. The following subsections describe these steps.

1. Convert a parameterized **rte** type into code that will perform run-time type checking.
2. Convert the regular type expression to DFA (deterministic finite automaton, sometimes called a *finite state machine*).
3. Decompose a list of type specifiers into disjoint types.
4. Convert the DFA into code which will perform run-time execution of the DFA.

4.1 Type definition

The **rte** type is defined by `deftype`.

```
(deftype rte (pattern)
  `(and sequence
      (satisfies ,(compute-match-function
                     pattern))))
```

As in this definition, when the `satisfies` type specifier is used, it must be given a `symbol` naming a globally callable unary function. In our case `compute-match-function` accepts a regular type expression, such as `(:1 number (:* string))`, and computes a named unary predicate. The predicate can thereafter be called with a sequence and will return `true` or `false` indicating whether the sequence matches the pattern. Notice that the pattern is usually provided at *compile*-time, while the sequence is provided at *run*-time. Furthermore, `compute-match-function` ensures that given two patterns which are `EQUAL`, the same function name will be returned, but will only be created and compiled once. An example will make it clearer.

```
(deftype 3-d-point ()
  `(rte (:1 number number number)))
```

The type `3-d-point` invokes the **rte** parameterized type definition with argument `(:1 number number number)`. The `deftype` of **rte** assures that a function is defined as follows. The function name, `|(:1 number number number)|` even if somewhat unusual, is so chosen to improve the error message and back-trace that occurs in some situations.

```
(defun rte::|(:1 number number number)|
    (input-sequence)
  (match-sequence input-sequence
           '(:1 number number number)))
```

The following back-trace occurs when attempting to evaluate a failing assertion.

```
CL-USER> (the 3-d-point (list 1 2))

The value (1 2)
is not of type
  (OR (AND #1=(SATISFIES |(:1 NUMBER NUMBER NUMBER)|)
           CONS)
      (AND #1# NULL) (AND #1# VECTOR)
      (AND #1# SB-KERNEL:EXTENDED-SEQUENCE)).
   [Condition of type TYPE-ERROR]
```

It is also assured that the DFA corresponding to `(:1 number number number)` is built and cached, to avoid unnecessary re-creation at run-time. Finally, the type specifier `(rte (:1 number number number))` expands to the following.

```
(and sequence
     (satisfies |(:1 number number number)|))
```

A caveat of using **rte** is that the usage must obey a restriction posed by the Common Lisp specification [15, Section DEFTYPE]. A self-referential type definition is not valid. Common Lisp specification states: *Recursive expansion of the type specifier returned as the expansion must terminate, including the expansion of type specifiers which are nested within the expansion.*

As an example of this limitation, here is a failed attempt to implement a type which matches a unary tree, *i.e.* a type whose elements are `1`, `(1)`, `((1))`, `(((1)))`, etc.

```
CL-USER> (deftype unary-tree ()
           `(or (eql 1)
                (rte unary-tree)))
UNARY-TREE
RTE> (typep '(1) 'unary-tree)
Control stack exhausted (no more space for function call
frames).  This is probably due to heavily nested or
infinitely recursive function calls, or a tail call that
SBCL cannot or has not optimized away.

PROCEED WITH CAUTION.
 [Condition of type SB-KERNEL::CONTROL-STACK-EXHAUSTED]
```

$$
\begin{aligned}
\partial_a \emptyset &= \emptyset \\
\partial_a \varepsilon &= \emptyset \\
\partial_a a &= \varepsilon \\
\partial_a b &= \emptyset \text{ for } b \neq a \\
\partial_a (r \cup s) &= \partial_a r \ \cup \ \partial_a s \\
\partial_a (r \cdot s) &= \begin{cases} (\partial_a r) \cdot s, & \text{if } r \text{ is not nullable} \\ (\partial_a r) \cdot s \ \cup \ \partial_a s, & \text{if } r \text{ is nullable} \end{cases} \\
\partial_a (r \cap s) &= \partial_a r \ \cap \ \partial_a s \\
\partial_a (r^*) &= (\partial_a r) \cdot r^* \\
\partial_a (r^+) &= (\partial_a r) \cdot r^*
\end{aligned}
$$

Figure 4: Rules for the Brzozowski derivative

4.2 Constructing a DFA

In order to determine whether a given sequence matches a particular regular type expression, we conceptually execute a DFA with the sequence as input. Thus we must convert the regular type expression to a DFA. This need only be done once and can often be done at compile time.

4.2.1 Rational derivative

In 1964, Janusz Brzozowski [3] introduced the concept of the *Rational Language Derivative*, and provided a theory for converting a regular expression to a DFA. Additional work was done by Scott Owens *et al.* [12] which presented the algorithm in easy to follow steps.

To understand what the rational expression derivative is and how to calculate it, first think of a rational expression in terms of its language, *i.e.* the set of sequences the expression *generates*. For example, the language of $((a|b) \cdot c^* \cdot d)$ is the set of words (finite sequences of letters) which begin with exactly one letter a or exactly one letter b, end with exactly one letter `d` and between contain zero or more occurrences of the letter c.

The *derivative of the language* with respect to a given letter is the set of suffixes of words which have the given letter as prefix. Analogously the *derivative of the rational expression* is the rational expression which generates that language. *E.g.*, $\partial_a((a|b) \cdot c^* \cdot d) = (c^* \cdot d)$.

The Owens [12] paper explains a systematic algorithm for symbolically calculating such derivatives. The formulas listed in Figure 4 detail the calculations which must be recursively applied to calculate the derivative.

4.2.2 DFA for regular expressions

Another commonly used algorithm for constructing a DFA was inspired by Ken Thompson [18, 17] and involves decomposing a rational expression into a small number of cases such as base variable, concatenation, disjunction, and Kleene star, then following a graphical template substitution for each case. While this algorithm is easy to implement, it has a serious limitation. It is not able to easily express automata resulting from the intersection or complemention of rational expressions. We rejected this approach as we would like to support regular type expressions containing the keywords `:and` and `:not`, such as in `(:and (:* t integer) (:not (:* float t)))`.

We chose the algorithm based on Brzozowski derivatives.

Initial state Create the single initial state, and label it with the original rational expression. Seed a *to-do* list with this initial state. Seed a *visited* list to $\emptyset$.

States While the *to-do* list is non empty, operate on the first element as follows:

1. Move the state from the *to-do* list to the *visited* list.
2. Get the expression associated with the state.
3. Calculate the derivative of this expression with respect to each letter of the necessarily finite alphabet.
4. Reduce each derivative to a canonical form.
5. For each canonical form that does not correspond to a state in the *to-do* nor *visited* list, create a new state corresponding to this expression, and add it to the *to-do* list.

Transitions Construct transitions between states as follows: If S_1 is the expression associated with state P_1 and S_2 is the expression associated with state P_2 and $\partial_a S_1 = S_2$, then construct a transition from state P_1 to state P_2 labeled a.

Final states If the rational expression labeling a state is *nullable*, *i.e.* if it matches the empty word, label the state a final state.

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.

4.2.3 DFA for regular type expressions

The set of sequences of Common Lisp objects is not a rational language, because for one reason, the perspective alphabet (the set of all possible Common Lisp objects) is not a finite set. Even though the set of sequences of objects is infinite, the set of sequences of type specifiers is a rational language, if we only consider as the alphabet, the set of type specifiers explicitly referenced in a regular type expression. With this choice of alphabet, sequences of Common Lisp type specifiers conform to the definition of *words* in a *rational language*.

There is a delicate matter when the mapping of sequence of objects to sequence of type specifiers: the mapping is not unique. This issue is ignored for the moment, but is discussed in Section 4.4.

Consider the extended rational type expression $P_0 = (symbol \cdot (number^+ \cup string^+))^+$. We wish to construct a DFA which recognizes sequences matching this pattern. Such a DFA is shown in Figure 5.

First, we create a state P_0 corresponding to the given rational type expression.

Next we proceed, to calculate the derivative with respect to each type specifier mentioned in P_0 and construct states P_1, P_2, and P_3 as those are the unique derivative forms which are obtained by the calculation. We discard the $\emptyset$ value.

$$
\begin{aligned}
\partial_{number} P_0 &= \emptyset \\
\partial_{string} P_0 &= \emptyset
\end{aligned}
$$

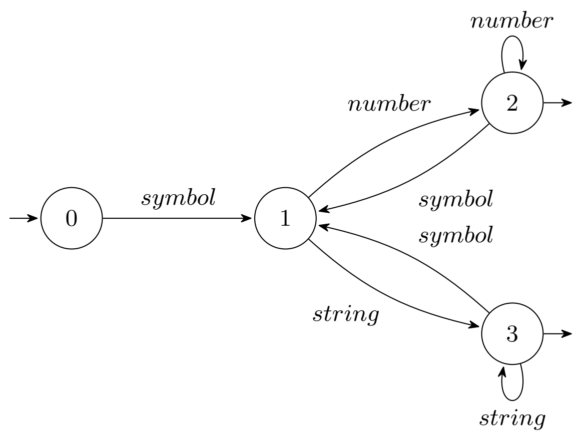

Figure 5: Example DFA

$$
\begin{aligned}
\partial_{symbol} P_0 &= (number^+ \cup string^+) \\
&\quad \cdot (symbol \cdot (number^+ \cup string^+))^* \\
&= P_1 \\
\partial_{number} P_1 &= number^* \\
&\quad \cdot (symbol \cdot (number^+ \cup string^+))^* \\
&= P_2 \\
\partial_{string} P_1 &= string^* \cdot (symbol \cdot (number^+ \cup string^+))^* \\
&= P_3 \\
\partial_{symbol} P_1 &= \emptyset \\
\partial_{number} P_2 &= P_2 \\
\partial_{string} P_2 &= \emptyset \\
\partial_{symbol} P_2 &= P_1 \\
\partial_{number} P_3 &= \emptyset \\
\partial_{string} P_3 &= P_3 \\
\partial_{symbol} P_3 &= P_1
\end{aligned}
$$

Next, we label the transitions between states with the type specifier which was used in the derivative calculation between those states. We ignore transitions from any state to the $\emptyset$ state.

Finally, we label the *final* states. They are P_2 and P_3 because only those two states are nullable. *I.e.* $(number^* \cdot (symbol \cdot (number^+ \cup string^+))^*)$ can match the empty sequence, and so can $(string^* \cdot (symbol \cdot (number^+ \cup string^+))^*)$

4.3 Optimized code generation

The mechanism we chose for implementing the execution (as opposed to the generation) of the DFA was to generate specialized code based on `typecase`, `block`, and `go`. As an example, consider the DFA shown in Figure 5. The code in Figure 6 was generated given this DFA as input.

The code is organized according to a regular pattern. The `typecase`, commented as `OUTER-TYPECASE` switches on the type of the sequence itself. Whether the sequence, `seq`, matches one of the carefully ordered types `list`, `simple-vector`, `vector`, or `sequence`, determines which functions are used to access the successive elements of the sequence: `svref`, `incf`, `pop`, etc.

The final case, `sequence`, is especially useful for applications which wish to exploit the SBCL feature of *Extensible sequences* [10, Section 7.6] [13]. One of our `rte` based applications uses extensible sequences to view vertical and horizontal *slices* of 2D arrays as sequences in order to match

```
(lambda (seq)
  (declare
    (optimize (speed 3) (debug 0) (safety 0)))
  (block check
    (typecase seq  ; OUTER-TYPECASE
      (list
       (tagbody
        0
          (when (null seq)
            (return-from check nil)) ; rejecting
          (typecase (pop seq) ; INNER-TYPECASE
            (symbol (go 1))
            (t (return-from check nil)))
        1
          (when (null seq)
            (return-from check nil)) ; rejecting
          (typecase (pop seq) ; INNER-TYPECASE
            (number (go 2))
            (string (go 3))
            (t (return-from check nil)))
        2
          (when (null seq)
            (return-from check t)) ; accepting
          (typecase (pop seq) ; INNER-TYPECASE
            (number (go 2))
            (symbol (go 1))
            (t (return-from check nil)))
        3
          (when (null seq)
            (return-from check t)) ; accepting
          (typecase (pop seq) ; INNER-TYPECASE
            (string (go 3))
            (symbol (go 1))
            (t (return-from check nil)))))
      (simple-vector
       ...)
      (vector
       ...)
      (sequence
       ...)
      (t nil))))
```

Figure 6: Generated code recognizing an RTE

certain patterns within row vectors and column vectors.

While the code is iterating through the sequence, if it encounters an unexpected end of sequence, or an unexpected type, the function returns `nil`. These cases are commented as `rejecting`. Otherwise, the function will eventually encounter the end of the sequence and return `t`. These cases are commented `accepting` in the figure.

Within the inner section of the code, there is one label per state in the state machine. In the example, the labels P_0, P_1, P_2, and P_3 are used, corresponding to the states in the DFA in Figure 5. At each step of the iteration, a check is made for end-of-sequence. Depending on the state either `t` or `nil` is returned depending on whether that state is a final state of the DFA or not.

The next element of the sequence is examined by the

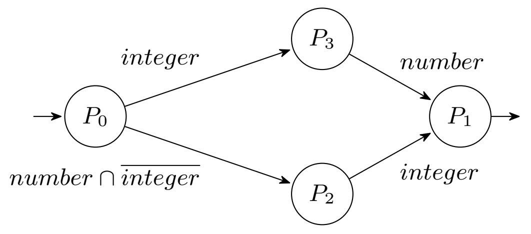

Figure 7: Example DFA with disjoint types

INNER-TYPECASE, and depending of the type of the object encountered, control is transferred (via go) to a label corresponding to the next state.

One thing to note about the complexity of this function is that the number of states encountered when the function is applied to a sequence is equal or less than the number of elements in the sequence. Thus the time complexity is linear in the number of elements of the sequence and is independent of the number of states in the DFA.

In some cases the same type may be specified with either the **rte** syntax or with the Common Lisp native cons type specifier. For example, a list of three numbers can be expressed either as (cons number (cons number (cons number null))) or as (rte (:1 number number number)).

Should the **rte** system internally exploit the cons specifier when possible, thus avoiding the creation of finite state machines? We began investigating this possibility, but abandoned the investigation on discovering that it lead to significant performance degradation for long lists. We measured roughly a 5% penalty for lists of length 5. The penalty grew for longer lists: 25% with a list length of 10, 40% with a list length of 20.

4.4 The overlapping types problem

In the example in Section 4.2.3, all the types considered (symbol, string, and number) were disjoint. If the same method is naively used with types which are intersecting, the resulting DFA will not be a valid representation of the rational expression. Consider the rational expression involving the intersecting types *integer* and *number*: $P_0 = ((number \cdot integer) \cup (integer \cdot number))$. The sequences which match this expression are sequences of two numbers, at least one of which is an integer. Unfortunately, when we calculate $\partial_{number} P_0$ and $\partial_{integer} P_0$ we arrive at a different result.

$$\begin{aligned}
\partial_{number} P_0 &= \partial_{number}(\ (number \cdot integer) \\
&\qquad\qquad \cup (integer \cdot number)) \\
&= \partial_{number}(number \cdot integer) \\
&\quad \cup \partial_{number}(integer \cdot number) \\
&= (\partial_{number} number) \cdot integer \\
&\quad \cup (\partial_{number} integer) \cdot number \\
&= \varepsilon \cdot integer \ \cup\ \emptyset \cdot number \\
&= integer \cup \emptyset \\
&= integer
\end{aligned}$$

Without continuing to calculate all the derivatives, it is already clear that this result is wrong. If you 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 suffixes of this set is **not** the set of singleton sequences of integer.

To address this problem, we augment 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. More specifically, given a set $\mathcal{A}$ of 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 $\cup\mathcal{A} = \cup\mathcal{B}$.

Figure 7 illustrates 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 assures that the constructed automaton is deterministic, *i.e.*, we assure 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 the Brzozowski/Owens algorithm.

The algorithm for decomposing a set of types into a set of disjoint types is an interesting research topic in its own right. While this topic is still under investigation, we have several algorithms which work very well for a small number of types (*i.e.* lists of up to 15 types). At the inescapable core of each algorithm is Common Lisp function subtypep [2]. This function is crucial not only in type specifier simplification, needed to test equivalence of symbolically calculated Brzozowski derivatives, but also in deciding whether two given types are disjoint. For example, we know that string and number are disjoint because (and string number) is a subtype of nil.

We explicitly omit further discussion of that algorithm in this article. We will consider it for future publication. For a complete exposition of our ongoing research into this topic, see the project report on the LRDE website [11].

5. RELATED WORK

Attempts to implement destructuring-case are numerous. We mention three here. R7RS Scheme provides case-lambda [14, Section 4.2.9] which appears to be syntactically similar construct, allowing argument lists of various fixed lengths. However, according to the specification nothing similar to Common Lisp style destructuring is allowed.

The implementation of destructuring-case provided in [6] does not have the feature of selecting the clause to be executed according to the format of the list being destructured. Rather it uses the first element of the given list as a case-like key. This key determines which pattern to use to destructure the remainder of the list.

The implementation provided in [7], named destructure-case, provides similar behavior to that which we have developed. It destructures the given list according to which of the given patterns matches the list. However, it does not handle destructuring within the optional and keyword arguments.

```
(destructuring-case '(3 :x (4 5))
  ((a &key ((:x (b c))))
   (list 0 a b c)) ;; this clause should be taken
  ((a &key x)
   (list 2 a x)))  ;; not this clause
```

In none of the above cases does the clause selection consider the types of the objects within the list being destructured. Clause selection also based on type of object is a distinguishing feature of the **rte** based implementation of destructuring-case.

The **rte** type along with destructuring-bind and typecase as mentioned in Section 3.3 enables something similar to pattern matching in the XDuce language [8]. The XDuce language allows the programmer to define a set of functions with various lambda lists, each of which serves as a pattern available to match particular target structure within an XML document. Which function gets executed depends on which lambda list matches the data found in the XML data structure.

XDuce introduces a concept called *regular expression types* which indeed seems very similar to *regular type expressions*.

In [8] Hosoya *et al.* introduce a *semantic type* approach to describe a system which enables their compiler to guarantee that an XML document conform to the intended type. The paper deals heavily with assuring that the regular expression types are well defined when defined recursively, and that decisions about subtype relationships can be calculated and exploited.

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

The concept of regular trees, is more general that what **rte** supports, posing interesting questions regarding apparent shortcomings of our approach. The semantic typing concept described in [8] indeed seems to have many parallels with the Common Lisp type system in that types are defined by a set of objects, and sub-types correspond to subsets thereof. These parallels would suggest further research opportunities related to **rte** and Common Lisp. However, the limitation that **rte** cannot be used to express trees of arbitrary depth as discussed in Section 4.1 seems to be a significant limitation of the Common Lisp type system. Furthermore, the use of `satisfies` in the **rte** type definition, seriously limits the `subtypep` function's ability to reason about the type. Consequently, programs cannot always use `subtypep` to decide whether two **rte** types are disjoint or equivalent, or even if a particular **rte** type is empty. Neither can the compiler dependably use `subtypep` to make similar decisions to avoid redundant assertions in function declarations.

It is not clear whether Common Lisp could provide a way for a type definition in an application program to extend the behavior of `subtypep`. Having such a capability would allow such an extension for **rte**. Rational language theory does provide a well defined algorithm for deciding such questions given the relevant rational expressions [9, Sections 4.1.1, 4.2.1]. It seems from the specification that a Common Lisp implementation is forbidden from allowing self-referential types, even in cases where it would be possible to do so.

6. CONCLUSIONS

In this paper we presented a Common Lisp type definition, **rte**, which implements a declarative pattern based approach for declaring types of heterogeneous sequences illustrating it with several motivating examples. We further discussed the implementation of this type definition and its inspiration based in rational language theory. While the total computation needed for such type checking may be large, our approach allows most of the computation to be done at compile time, leaving only an $\mathcal{O}(n)$ complexity calculation remaining for run-time computation.

Our contributions are

1. recognizing the possibility to use principles from rational theory to address the problem dynamic type checking of sequences in Common Lisp,

2. adapting the Brzozowski derivative algorithm to sequences of lisp types by providing an algorithm to symbolically decompose a set of lisp types into an equivalent set of disjoint types,

3. implementing an efficient $\mathcal{O}(n)$ algorithm to pattern match an arbitrary lisp sequence, and

4. implementing concrete **rte** based algorithms for recognizing certain commonly occurring sequence patterns.

For future extensions to this research we would like to experiment with extending the `subtypep` implementation to allow application level extensions, and therewith examine run-time performance when using **rte** based declarations within function definitions.

Another topic we would like to research is whether the core of this algorithm can be implemented in other dynamic languages, and to understand more precisely which features such a language would need to have to support such implementation.

7. REFERENCES

[1] Declaring the elements of a list, discussion on comp.lang.lisp, 2015.

[2] H. G. Baker. A decision procedure for Common Lisp's SUBTYPEP predicate. *Lisp and Symbolic Computation*, 5(3):157–190, 1992.

[3] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[4] R. D. Cameron. Perl style regular expressions in Prolog, CMPT 384 lecture notes, 1999.

[5] J. Chroboczek. CL-Yacc, a LALR(1) parser generator for Common Lisp, 2009.

[6] P. Domain. Alexandria implementation of destructuring-case.

[7] N. Funato. Public domain implementation of destructuring-bind, 2013.

[8] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, Jan. 2005.

[9] J. D. U. Johh E. Hopcroft, Rajeev Motwani. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2001.

[10] W. H. Newman. Steel Bank Common Lisp user manual, 2015.

[11] J. Newton. Report: Efficient dynamic type checking of heterogeneous sequences. Technical report, EPITA/LRDE, 2016.

[12] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, Mar. 2009.

[13] C. Rhodes. User-extensible sequences in Common Lisp. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 13:1–13:14, New York, NY, USA, 2009. ACM.

[14] A. Shinn, J. Cowan, and A. A. Gleckler. Revised 7 report on the algorithmic language scheme. Technical report, 2013.

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

[16] E. Weitz. *Common Lisp Recipes: A Problem-solution Approach.* Apress, 2015.

[17] G. Xing. Minimized Thompson NFA. *Int. J. Comput. Math.*, 81(9):1097–1106, 2004.

[18] F. Yvon and A. Demaille. *Théorie des Langages Rationnels.* 2014.