Project Report: Efficient dynamic type checking of heterogeneous sequences

Jim Newton

May 12, 2016

Abstract

This report provides detailed background of our development of the *rational type expression*, concrete syntax, *regular type expression*, and a Common Lisp implementation which allows the programmer to declarative express the types of heterogeneous sequences in a way which is natural in the Common Lisp language. We present a brief theoretical background in rational language theory, which facilitates the development of rational type expressions, in particular the use of the Brzozowski derivative and deterministic automata to arrive at 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.

1 Introduction

In Common Lisp a type is a set of (potential) values at a particular point of time during the execution of a program[23]. Information about types, whether declared in the code (which is an option available to the programmer) or inferred by the compiler, provides clues for the compiler to exploit in making certain optimizations such as for performance, space (image size), safety or debugability.[15] In addition, application programmers may make explicit use of types within the logic of their programs, such as with `typecase`, `typep`, `the` etc.

One observed weakness of the Common Lisp type system relates to sequences. The Common Lisp specification indeed allows the programmer great flexibility indicate a homogeneous type for all the elements of a vector[23]. The following small code example shows the Common Lisp notation for allocating a one dimensional array of 10 elements, each of which are integer, and a one dimensional array of 256 elements each of which may be either a string or a number.

```
(make-array '(256) :element-type '(or string number))
```

Two notable limitations are that there is no standardized way to specify heterogeneous types for different elements of the vector, neither is there a standardized way to declare types for all the elements of a list. See section 9.1 for a vain attempt.

We avoid making claims about the potential effectiveness of such type declarations from the compiler's perspective. Opinions differ as to what advantage compilers could make of this information.[1] There would be numerous and obvious challenges posed by such an attempt. At run-time, cons cells may be freely altered because Common Lisp does not provide read-only cons cells. There is a large number of Common Lisp functions which are allowed to modify cons cells, thus violating the proposed type constraints if left unchecked. Additionally, if declarations were made about certain lists, and thereafter other lists are created (or modified) to share those tails, it is not clear which information about the tails should be maintained.

Nevertheless, we do suggest that a declarative system to describe patterns of types within sequences (vectors and lists) would have great utility for program logic and code readability.

We introduce the *rational type expression* as an abstract concept for describing such patterns within sequences. The concept is envisioned to be intuitive to the user in that it is analogous to patterns described by regular expressions.

Just as the characters of a `string` might 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 $(string \cdot number^* \cdot symbol)$ is intended to match the vector `#("hello" 1 2 3 world)` and the list `("hello" world)`. *I.e.*, while rational expressions are intended to match character constituents of strings according to character identity, rational type expressions are intended to match elements of sequences by element type.

To this end we have implemented a lisp friendly syntax for denoting rational type expressions. We call the lisp implementation a *regular type expression.* The syntax of the regular type expression replaces the infix and post-fix operators in the rational type expression with prefix notation based s-expressions. The regular type expression `(:cat string (:0-* number) symbol)` corresponds to the rational type expression $(string \cdot number^* \cdot symbol)$. In addition, we have implemented a Common Lisp parameterized type named `rte`, whose arguments are rational type expressions. The members of such a type are all sequences (lists or vectors) matching the given regular type expression. See Section 3.2 for more details about the syntax of regular type expressions.

As the lisp programmer would expect, the `rte` type may be used anywhere within a lisp program that a type specifier is expected. For example:

```
(assert (typep my-list '(rte (:cat mytype number))))

(deftype plist ()
  `(rte (:0-* symbol t)))

(defun F (obj plist list-of-int)
  (declare (type plist plist)
           (type (and list (rte (:0-* integer))) list-of-int))
  (typecase obj
    ((rte symbol (:0-* number))
     (destructuring-bind (name &rest numbers) obj
        ...))
    ((rte symbol list (:0-* string))
     (destructuring-bind (name data &rest strings) obj
        ...))))
```

See section 7.4 for more details of list destructuring.

In this article we summarize the theory of rational languages, including an algorithm to construct a finite state machine which recognizes words in a given rational language. We extend the theory to accommodate rational type expressions. We present the Common Lisp implementation of regular type expressions including some analysis of their performance against other reasonable approaches.

2 Theory of Rational Languages

An *alphabet* is defined as any finite set, the elements of which are defined as *letters.* We generally denote the letters of an alphabet by Latin letter symbols. *E.g.*, $\Sigma = \{a, b, c\}$.

Given an alphabet, Σ, a *word* of length $n \in \mathbb{N}$ is a sequence of n characters from the alphabet. This can be denoted as a function mapping the set $\{1, 2, 3...n\}$ to Σ. We denote the sequence in an intuitive way, simply as a juxtaposed sequence of characters. For example $aabc$ denotes the following function $aabc : \{1, 2, 3, 4\} \to \Sigma$

$$aabc(n) = \begin{cases} a & \text{if } n = 1 \\ a & \text{if } n = 2 \\ b & \text{if } n = 3 \\ c & \text{if } n = 4 \end{cases}$$

Note that there is a word of zero length, called the *empty word.* It is denoted ε. The empty word is indeed a function $\varepsilon : \emptyset \subset \mathbb{N} \to \Sigma$.

A *language* is defined as a set of words; more specifically a *language in* Σ is a set of words each of whose letters are in Σ. The set of all words of length one whose letters come from Σ is denoted

Σ^1. The set of all possible words of finite length made up exclusively of letters from Σ is denoted Σ^*. Also note that $\varepsilon \in \Sigma^*$ and $\Sigma^1 \subset \Sigma^*$. Some languages have finite cardinally, while others have countably infinite cardinally.

Examples of languages of $\Sigma = \{a, b\}$ are $\emptyset$, Σ, $\{a, aa, aaa, aaaa\}$, and $\{\varepsilon, ab, aaba, ababbb, aaaababababbbb\}$.

If L is a language and $u, v \in L$, then we define the *concatenation* of u and v as the sequence of letters comprising u followed immediately by the sequence of characters comprising v. The concatenation of words is denoted either by a juxtaposition of symbols or using the $\cdot$ operator: i.e., uv or equivalently $u \cdot v$.

Precisely, if $u : \{1, 2, ..., n_1\} \to \Sigma$ and $v : \{1, 2, ..., n_2\} \to \Sigma$, then $u \cdot v : \{1, 2, ..., (n_1 + n_2)\} \to \Sigma$, such that:

$$(u \cdot v)(n) = \begin{cases} u(n) & \text{if } 1 \leq n \leq n_1 \\ v(n - n_1) & \text{if } n_1 + 1 \leq n \leq n_1 + n_2 \end{cases}$$

If A and B are languages, then $A \cdot B = \{u \cdot v \mid u \in A \text{ and } v \in B\}$. As a special case, if $A = B$, we denote $A \cdot A = A^2$. Similarly $A^n = A \cdot A^{n-1}$. When it is unambiguous, we sometimes denote $A \cdot B$ simply as AB.

If A is a language, then A^*, the *Kleene closure of* A[11], denotes the set of words w such that $w \in A^n$ for some $n \in \mathbb{N}$.

A *rational language* is defined by a recursive definition: The two sets $\emptyset$ and $\{\varepsilon\}$ are rational languages. For each letter of the alphabet, the singleton set containing the corresponding one letter word is rational language. In addition to these base definitions, any set which is the union or concatenation of a two rational languages is a rational language. The Kleene closure of a rational language is a rational language.

Let $\mathcal{L}_\Sigma$ denote the set of all rational languages.

Otherwise stated:

1. $\emptyset \in \mathcal{L}_\Sigma$.
2. $\{\varepsilon\} \in \mathcal{L}_\Sigma$.
3. $\{a\} \in \mathcal{L}_\Sigma \ \forall \ a \in \Sigma^1$.
4. $(A \cup B) \in \mathcal{L}_\Sigma \ \forall \ A, B \in \mathcal{L}_\Sigma$.
5. $(A \cdot B) \in \mathcal{L}_\Sigma \ \forall \ A, B \in \mathcal{L}_\Sigma$.
6. $A^* \in \mathcal{L}_\Sigma \ \forall \ A \in \mathcal{L}_\Sigma$.

While not part of the definition as such, it can be proven that if $A, B \in \mathcal{L}_\Sigma$ then $A \cap B \in \mathcal{L}_\Sigma$ and $A \setminus B \in \mathcal{L}_\Sigma$[11].

2.1 Rational expressions

The definition of rational language given in section 2 provides a top-down mechanism for identifying regular languages. *I.e.*, languages are rational if they can be decomposed into other rational languages via certain set operations such as union, intersection, and concatenation. Conversely, new rational languages can be *discovered* by combining given rational languages in well defined ways.

Another way to identify rational languages is a bottom-up approach. This approach is based on the letters, rather than the sets. Rational expressions allow us to specify pattern based rules for determining which words are in a given language. We will say that a rational expression *generates* a language.

A rational expression is an algebraic expression, using the intuitive algebraic operators. A rational expression *generates* a language. The notation $L = [\![r]\!]$, means that the rational expression, r generates the language L.

We denote the set of all rational expressions as $\mathcal{E}_{rat}$.

$$[\![\emptyset]\!] = \emptyset \qquad [\![\varepsilon]\!] = \{\varepsilon\} \qquad \forall a \in \Sigma^1, [\![a]\!] = \{a\}$$
$$[\![r+s]\!] = [\![r]\!] \cup [\![s]\!] \qquad [\![r^*]\!] = [\![r]\!]^* \qquad [\![rs]\!] = [\![r \cdot s]\!] = [\![r]\!] \cdot [\![s]\!] \qquad [\![r \cap s]\!] = [\![r]\!] \cap [\![s]\!]$$

This abuse of notation is commonplace in rational language theory. The same symbol a is used to denote a letter, $a \in \Sigma$, a word of length one, $a \in \Sigma^1$, and a rational expression, a $s.t.[\![a]\!] = \{a\} \subset \Sigma^*$. Analogously, the symbol ε is abused to denote both the empty word, $\varepsilon \in \Sigma^*$, and a rational expression ε $s.t.$ $[\![\varepsilon]\!] = \{\varepsilon\} \subset \Sigma^*$. Further, $\emptyset \subset \Sigma^*$ denotes the empty language, and also the rational expression, $\emptyset$ $s.t.$ $[\![\emptyset]\!] = \emptyset \subset \Sigma^*$.

It can be proven that the operations $+$ and $\cdot$ are associative[11], which means that without ambiguity we may write $(a+b+c)$ and $(a \cdot b \cdot c)$ omitting additional parentheses. However, we must define a precedence order to give an unambiguous meaning to expressions such as $a^*b+c\cdot d^*$. The precedence order from highest precedence to lowest is defined to be $(^*, \cdot, +)$, so that $a^*b+c\cdot d^*$ unambiguously means $((a^*)\cdot b)+(c\cdot(d^*))$.

As an example, let $\Sigma \supset \{a,b,c,d,e,f\}$; the rational expression $a\cdot(b\cdot d^*+c\cdot e^*)\cdot f$, or equivalently $a(bd^*+ce^*)f$, can be understood to be a rational expression generating the set (language) of words which start with exactly one a, end with exactly one f, and between the a and f is either exactly one b followed by zero or more d's or exactly one c followed by zero or more e's.

The definition trivially implies that $\forall r \in \mathcal{E}_{rat}\ \exists R \in \mathcal{L}_\Sigma \mid [\![r]\!] = R \in \mathcal{L}_\Sigma$. Conversely, $\forall R \in \mathcal{L}_\Sigma \exists r \in \mathcal{E}_{rat} \mid [\![r]\!] = R$.

2.2 Regular expressions

We would like to avoid confusion between the terms `regular expression` and `rational expression`. We use the term `regular expression` to denote programmatic implementations such as provided in `grep` and `Perl`. We assume the reader is familiar, at least with the usage of, UNIX based regular expressions.

By contrast, we reserve the term *rational expression* to denote the algebraic expressions as described in section 2.1.

There are regular expression libraries available for a wide variety of programming languages. Each implementation uses different ASCII characters to denote the rational language operations, often equipped with additional operations which are eventually reducible to the atomic operations shown above, and whose inclusion in the implementation adds expressivity in terms of syntactic sugar.

One of the oldest applications of regular expressions was in specifying the component of a compiler called a "lexical analyzer". The UNIX command `lex` allows the specification of tokens in terms of regular expressions in UNIX style and associates code to be executed when such a token is recognized[11].

The same style regular expressions are built into several standard UNIX utilities such as `grep`, `egrep`, `sed` and several other programs. These implementations provide useful notations such as:

$+$ `"ab+c"`, one or more times, is equivalent to $a \cdot b \cdot b^* \cdot c$

`?` `"ab?c"`, zero or one time, is equivalent to $a \cdot (b+\varepsilon) \cdot c$

`.` `"a.c"`, any character, is equivalent to $a \cdot \Sigma^1 \cdot c$

The PCRE (Perl Compatible Regular Expressions)[] library available in many languages such as C, SKILL[5] represent the rational expression shown above as `"a(bd*|cd*)f"`.

2.3 Finite Automata

Finite automata provide a computational model for implementing recognizers for rational languages[16].

A *DFA* (Non-Deterministic Finite Automaton) A is a 5-tuple $A = (\Sigma, Q, I, F, \delta)$ where:

Σ is an alphabet, (an alphabet is finite by definition)

Q is a finite set whose elements are called *states*

$I \subset Q$ is a set whose elements are called *initial states*

$F \subset Q$ is a set whose elements are called *final states*

$\delta \subset Q \times \Sigma \times Q$ is a set whose elements are called *transitions*.

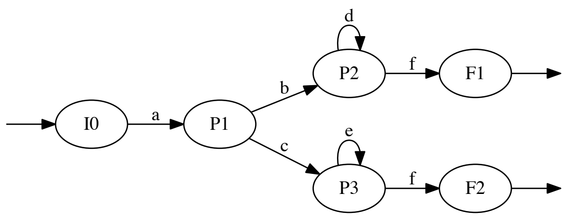


Figure 1: DFA recognizing the regular expression

Each transition can be denoted $\alpha \xrightarrow{a} \beta$ for $\alpha, \beta \in Q$ and $a \in \Sigma$. Figure 1 shows a finite automaton. It has initial state $I = \{I_0\}$, final states $F = \{F_1, F_2\}$, and the following transitions: $\delta = \{I_0 \xrightarrow{a} P_1, P_1 \xrightarrow{b} P_2, P_2 \xrightarrow{d} P_2, P_2 \xrightarrow{f} F_1, P_1 \xrightarrow{c} P_3, P_3 \xrightarrow{e} P_3, P_3 \xrightarrow{f} F_1\}$.

2.4 Equivalence of Rational Expressions and Finite Automata

It has been proven[11] that the following statements are equivalent.

1. $L \in \mathcal{L}_\Sigma$
2. $\exists r \in \mathcal{E}_{rat} \mid L = [\![r]\!]$
3. $L \subset \Sigma^*$ is recognizable by a finite automaton

In fact Figure 1 illustrates a finite automaton which recognizes the regular expression `a(bd*+ce*)f`.

2.5 The Rational Expression Derivative

There are several algorithms for generating a finite automaton from a given rational expression. One very commonly used algorithm was inspired by Ken Thompson[26, 25] and involves straightforward pattern substitution. While this algorithm is easy to implement it has a serious limitation. It is not able to easily express automata resulting from the intersection of two rational expressions.

Because of this limitation we have chosen to use the algorithm based on regular expression *derivatives*. This algorithm was first presented in 1964 by Janusz Brzozowski[6]. While Brzozowski's result was applied to digital circuits, Scott Owens *et al.* [16] extended the principle to generalize regular pattern recognition for sequences of characters.

Before defining the derivative, it is useful to first define nullability. A rational language is said to be *nullable* if it contains the empty word; *i.e.*, a language $L \subset \Sigma^*$ is *nullable* if $\varepsilon \in L$. Likewise, a rational expression r is *nullable* if $[\![r]\!]$ nullable. As will be seen below, to calculate the derivative of some rational expressions, we must calculate whether the rational expression is nullable. The function ν (the Greek letter nu) calculates nullability. $\nu : \mathcal{E}_{rat} \to \{\emptyset, \varepsilon\} \subset \mathcal{E}_{rat}$ as defined according to the recursive rules in Figure 2. If $\nu(r) = \varepsilon$ then r is nullable. If $\nu(r) = \emptyset$ then r is not nullable.

$$\begin{aligned}
\nu(\emptyset) &= \emptyset && (1)\\
\nu(\varepsilon) &= \varepsilon && (2)\\
\nu(a) &= \emptyset \ \forall \ a \in \Sigma && (3)\\
\nu(r+s) &= \nu(r) + \nu(s) && (4)\\
\nu(r \cdot s) &= \nu(r) \ \cap \ \nu(s) && (5)\\
\nu(r \ \cap \ s) &= \nu(r) \ \cap \ \nu(s) && (6)\\
\nu(r^*) &= \varepsilon && (7)
\end{aligned}$$

Figure 2: Recursive rules defining the nullability function ν

Definition Given a language $L \subset \Sigma^*$ and a word $w \in \Sigma^*$, the *derivative of L with respect to w* denoted $\partial_w L$ is a language $\partial_w L = \{v \mid w \cdot v \in L\}$.

For example. Suppose $L = \{this, that, those, fred\}$, then $\partial_{th} L = \{is, at, ose\}$. Basically take the words which start with the given prefix, and remove the prefix.

It can be proven that if $L \in \mathcal{L}_\Sigma$, then $\partial_w L \in \mathcal{L}_\Sigma \ \forall \ w \in \Sigma^*$[16]. However, it is not implied nor is it true in general that $\partial_w L \subset L$.

If $[\![S]\!] = L$, and $w \in L$, then a *derivative of S with respect to w* is denoted $\partial_w S$. Moreover, $\partial_w S \in \mathcal{E}_{rat}$ and $[\![\partial_w S]\!] = \partial_w L$. Otherwise stated, we can speak of either the derivative of the language L or the derivative of a rational expression.

Given a rational expression, we would like to be able to calculate the rational expression representing its derivative. To do this the reduction rules shown in Figure 3 can be recursively applied.

$$\begin{aligned}
\partial_a \emptyset &= \emptyset && (8)\\
\partial_a \varepsilon &= \emptyset && (9)\\
\partial_a a &= \varepsilon && (10)\\
\partial_a b &= \emptyset \text{ for } b \neq a && (11)\\
\partial_a (r+s) &= \partial_a r + \partial_a s && (12)\\
\partial_a (r \cdot s) &= \begin{cases} \partial_a r \cdot s, & \text{if } \nu(r) = \emptyset \quad (13)\\ \partial_a r \cdot s + \partial_a s, & \text{if } \nu(r) = \varepsilon \quad (14)\\ \partial_a r \cdot s + \nu(r) \cdot \partial_a s, & \text{in either case} \quad (15) \end{cases}\\
\partial_a (r \ \cap \ s) &= \partial_a r \ \cap \ \partial_a s && (16)\\
\partial_a (r^*) &= \partial_a r \cdot r^* && (17)\\
\partial_\varepsilon r &= r && (18)\\
\partial_{u \cdot v} r &= \partial_v (\partial_u r) && (19)
\end{aligned}$$

Figure 3: Rules for the Brzozowski derivative

Note that (15) is useful for theoretical and hand calculation but is problematic for algorithmic calculation. In the case that $\nu(r)$ is $\emptyset$, (15) is equivalent to (13), but warning, the calculation of

$\partial_a s$ may result in an infinite recursion. Thus, algorithmically, (13) and (14) should be used instead of (15).

To compute the automaton corresponding to a rational expression[16]:

1. Start with an initial state labeled by the rational expression itself, S.
2. For each letter $a \in \Sigma$, we calculate $\partial_a S$.
3. If there is not already a state labeled with the derivative, create one.
4. Create a transition $S \xrightarrow{a} \partial_a S$.
5. Each state labeled with a nullable rational expression is a final state.

There are a couple of useful optimization steps.

If the derivative is $\emptyset$, there is really no reason to explicitly add the a null state to the automaton. Doing so would clutter the graphical representation with arrows leading to this state.

It is not necessary that there be a 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 there there might be one *canonical reduced* expression which could be arrived at given a finite set of identities such as $\emptyset + L = L = L + \emptyset, \varepsilon \cdot L = L = L \cdot \varepsilon, (L^*)^* = L^*, L + K = K + L$, *etc.* In fact, there is no finite set of identities which permits to deduce all identities between rational expressions[18].

It suffices to allow the same derivative in two different algebraic forms to be represented by multiple states as long as it is a reasonable number. There must be some reduction step in the derivative calculation to limit the number of forms expressed, but the reduction need not actually reduce every expression to a unique, canonical form.

3 Heterogeneous sequences in Common Lisp

The Common Lisp language supports heterogeneous sequences in the form of sequentially accessible lists and several arbitrarily accessible vectors. A sequence is an ordered collection of elements, implemented as either a vector or a list [23]. The lisp reader recognizes syntax supporting several types of sequence.

```
"a string is a sequence of characters"
(list of 9 elements including "symbols" "strings and" a number)
#(vector of 9 elements including "symbols" "strings and" a number)
```

The Common Lisp function `map` iterates a given client function over the successive elements of a sequence. When given first argument as `nil`, `map` ignores the return value of the client function. Example usages of the `map` function.

```
(map nil #'(lambda (char)
             (princ (char-upcase char)))
     "abcde")

(map nil #'(lambda (num)
             (princ (* num num)))
     #(1 3 2 4 6))
```

3.1 Types in Common Lisp

As stated earlier, a *type* is a (possibly infinite) set of objects at a particular point of time during the execution of a program [23][1]. An object can belong to more than one type. Types are never

[1]In Common Lisp, types and functions may be redefined. Also an object of a particular class may be victim to the `change-class` function. Both of these situations as well as several others may cause a type to change its members while a program is running.

explicitly represented as objects by Common Lisp. Instead, they are referred to indirectly by the use of *type specifiers*, which are objects that denote types.

New types can be defined using `deftype`, `defstruct`, `defclass`, and `define-condition`. But type specifiers indicating compositional types are often used on their own, such as in the expression `(typep x '(or string (eql 42)))`, which evaluates to `true` either if `x` is a string, or is the integer `42`.

Two important Common Lisp functions pertaining to types are `typep` and `subtypep`. The function `typep`, a set membership test, is used to determine whether a given object is of a given type. The function `subtypep`, a subset test, is used to determine whether a given type is a *recognizable* subtype of another given type. The function call `(subtypep T1 T2)` distinguishes three cases:

That `T1` is a subtype of `T2`,

That `T1` is not a subtype of `T2`, or

That subtype relationship cannot be determined.

Section 6.2 discusses situations for which the subtype relationship cannot be determined.

3.2 The regular type expression

We have implemented a Common Lisp parameterized type named `rte` (regular type expression), via `deftype`. Some implementation details are explained in Section 8. Having this definition allows us to use the type `rte` anywhere Common Lisp expects a type specifier. The arguments to `rte` are *regular type expressions*. A syntactically correct *regular type expression* is either a Common Lisp type specifier, such as `number`, `(cons number)`, `(eql 12)`, or `(and integer (satisfies oddp))`, or rather a list whose first element is one of a limited set of keywords shown in Section 3.4, and whose trailing elements are other regular type expressions. Here are some examples.

`(rte number number number)` matches a sequence of exactly three numbers.

`(rte (:or (:cat number number) (:cat number number number)))` matches a list of either two or three numbers.

`(rte number number (:0-1 number))` matches a sequence of two mandatory numbers followed optionally by exactly one more number. This happens to be equivalent to the previous example: `(rte number number (:0-1 number))`.

The following example declares a class whose `point` slot is a list of two numbers. A subtlety to note is that `rte` is a subtype of `sequence` not of `list`. This means that `(rte number number)` will match not only the list `(1 2.0)` but also the vector `#(1 2.0)`.

```
(defclass F ()
  ((point :type (and list (rte number number)))
   #|...|#))
```

The following is the definition of a function whose second argument must be a list of exactly 2 strings or 3 numbers.

```
(defun F (X Y)
  (declare (type Y (and list
                        (rte (:or (:cat number number number)
                                  (:cat string string))))))
  #|...|#)
```

The following declares types named `point-2d`, `point-3d`, and `point-sequence` which can be used in other declarations:

```
(deftype point-2d ()
  "A list of exactly two numbers."
  '(and list (rte number number)))

(deftype point-3d ()
  "A list of exactly three numbers."
  `(and list (rte number number number)))

(deftype point-sequence ()
  "A list or vector of points, each point may be 2d or 3d."
  '(rte (:or (:0-* point-2d) (:0-* point-3d))))
```

3.3 Clarifying some confusing points about regular type expressions

There are a couple of potentially confusing points to note about the syntax of the regular type expression.

The arguments of `rte` are one or more regular type expressions which may be either common lisp type specifiers or other regular type expressions, and this is not ambiguous. Consider an example with the `cons` type specifier. In Common Lisp an object of type `cons` is a non-nil list. An object of type `(cons number)` is a list whose first element is of type `number`.

`(rte (:cat cons number))` — A sequence of length 2 whose respective elements are a non-empty list and a number.

`(rte cons number)` — Same as `(rte (:cat cons number))` because the outer `:cat` is implicit.

`(rte (:cat (cons number)))` — A sequence of length 1 whose element is a list whose first element is a number.

`(rte (cons number))` — Same as `(rte (:cat (cons number)))`.

Another potentially confusing point about the syntax is that `and` and `:and` (similarly `or` and `:or`) may both be used but have different meanings in most cases. The Common Lisp type specifiers, `and` and `or` match exactly one object. For example: `(or string symbol)` matches one object which must either be a `string` or a `symbol`. The arguments of `and` and `or` are Common Lisp type specifiers. For example `(and (:1-* string) (:0-* number))` is not valid because `(:1-* string)` and `(:0-* number)` are not valid Common Lisp type specifiers.

Contrast that with the regular type expression keywords `:and` and `:or` whose arguments are regular type expressions. For example `(rte (:or (:1-* string) (:0-* number)))`.

Additionally, regular type expressions may reference Common Lisp type specifiers. For example: `(rte (:or (:1-* string) (and list (not null))))`, which matches either a non-empty sequence of strings, or a singleton sequence whose element is a non-empty list.

It may be confusing the difference between `(:cat number symbol)` and `(rte (:cat number symbol))`. We refer to an expression such as `(:cat number symbol)` as a regular type expression, and the corresponding Common Lisp type is specified by `(rte (:cat number symbol))`. In fact `(rte (:cat number symbol))` can be used, within Common Lisp code, anywhere a lisp type specifier is expected. However, `(:cat number symbol)` is not a Common Lisp type specifier; it may only be used where a regular type expression is expected. A subtle point is that any Common Lisp type specifier is a valid regular type expression (but not vice versa). So `(rte (:cat number symbol))` may also be used where a regular type expression is expected, including being used recursively within another regular type expression. Compare the following:

`(rte (:cat number (rte (:cat symbol symbol))))` — matches a sequence of length exactly two, whose first element is a number, and whose second element is a sequence of exactly two symbols. E.g., `(1.1 (a b))`

`(rte (:cat number (:cat symbol symbol)))` — matches a sequence of length exactly three whose first element is a number, and whose next two elements are symbols. E.g., `(1.1 a b)`

3.4 Regular type expression keywords

Here is a detailed explanation of the keywords available within the structure of the `rte` type specifier.

:0-* match zero or more times. The following example matches a sequence of `string`, `number` and `list` repeated zero or more times, *e.g.*, `()`, `("abc" 1.2 (a b c))`, or `("abc" 1.2 (a b c) "xyz" 3 ())`, but not `("abc" 1.2 (a b c) 100)`

```
(:0-* string number list)
```

:1-* match one or more times. Similar to `:0-*` but refuses to match zero times.

```
(:1-* string number list)
```

:0-1 match zero or one time. *E.g.*, the following matches `()` and `("abc" 1.2 (a b c))`, but not `("abc" 1.2 (a b c) "xyz" 3 ())`.

```
(:0-1 string number list)
```

:cat match exactly once. The following example matches a list of three numbers. They keywords `:0-*`, `:1-*`, and `:0-1` act as they have an implicit `:cat` so that the following are equivalent.

```
(:0-* number string list)
(:0-* (:cat number string list)
```

:or match any of the regular type expressions. The following example matches a sequence which consists either of all strings or all symbols.

```
(:or (:0-* string) (:0-* symbol))
```

:and match all of the regular type expressions. The following example matches a sequence which starts with two strings, and also ends with two strings.

```
(:and (:cat string string (:0-* t))
      (:cat (:0-* t) string string))
```

Note that this is different from `(rte string string (:0-* t) string string)`, as the former matches a list of exactly two strings, and the latter does not.

:permute match all of the descriptors once but in any order. The following example matches `(z x y)` and `(z y x)` but neither `(x y)` nor `(x y x)`.

```
(:permute (eql x) (eql y) (eql z))
```

4 Constructing an automaton

In order to write a function in Common Lisp which verifies whether a given sequence matches a given regular type expression, we would like to first convert the regular type expression to a DFA. The Brzozowski algorithm, explained in section 2.5, can be used for this conversion if the set of sequences is a rational language. 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.[2]

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

[2]The computation model of Common Lisp assumes infinite memory. In reality the memory is finite, but as far as theoretical considerations we assume the memory, and thus the set of all potential objects is infinite.

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 problem that the mapping of sequence of objects to sequence of type specifiers is not unique. This problem is discussed in section 5. For the moment, we ignore this complication as it would obfuscate the derivation of the DFA.

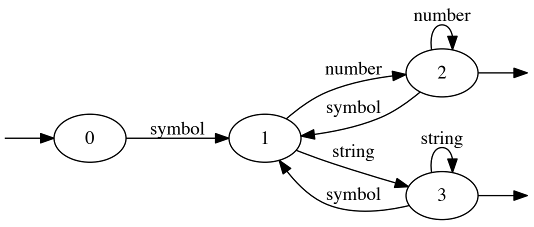


Figure 4: Example DFA

Consider the following regular type expression. We wish to construct a finite automaton which recognizes sequences matching this pattern. Such an automaton is shown in Figure 4.

```
(:0-* symbol (:or (:0-* number)
                  (:0-* string))))
```

This corresponds to the rational type expression:

$$P_0 = (symbol \cdot (number^* + string^*))^* \tag{20}$$

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

Next we proceed, to calculate the derivative with respect to each type specifier mentioned in P_0. Actually, as will be seen, it suffices to differentiate with respect to the type specifiers 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 $\emptyset$.

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

Thus we need only calculate one derivative: $\partial_{symbol} P_0$.

$$\begin{aligned}
\partial_{symbol} P_0 &= \partial_{symbol}((symbol \cdot (number^* + string^*))^*) && \text{By (20)}\\
&= \partial_{symbol}(symbol \cdot (number^* + string^*))\\
&\quad \cdot (symbol \cdot (number^* + string^*))^* && \text{By (17)}\\
&= (\partial_{symbol} symbol \cdot (number^* + string^*)\\
&\quad +\nu(symbol) \cdot \partial_{symbol}(number^* + string^*))\\
&\quad \cdot (symbol \cdot (number^* + string^*))^* && \text{By (14)}\\
&= (\varepsilon \cdot (number^* + string^*) + \emptyset \cdot \partial_{symbol}(number^* + string^*))\\
&\quad \cdot (symbol \cdot (number^* + string^*))^* && \text{By (10) and (3)}\\
P_1 &= (number^* + string^*) \cdot (symbol \cdot (number^* + string^*))^* && (21)
\end{aligned}$$

The corresponding regular type expression is

```
P1 = (:cat (:or (:0-* number) (:0-* string))
           (:0-* symbol (:or (:0-* number) (:0-* string))))
```

Since there is not yet a state in the automaton labeled with this expression, we create one named P_1. We also create a transition $P_0 \xrightarrow{symbol} P_1$. This transition is labeled with *symbol* because $\partial_{symbol} P_0 = P_1$. The transition corresponds to the arrow on the graph in Figure 4 from P_0 to P_1 labeled *symbol*. We now proceed to calculate the derivatives of P_1.

$$P_2 \quad = \quad \partial_{number} P_1 = number^* \cdot (symbol \cdot (number^* + string^*))^*$$

The corresponding regular type expression is

```
P2 = (:cat (:0-* number)
           (:0-* symbol (:or (:0-* number) (:0-* string))))
```

We add a state P_2 to the state machine with a transition $P_1 \xrightarrow{number} P_2$.

$$P_3 \quad = \quad \partial_{string} P_1 = string^* \cdot (symbol \cdot (number^* + string^*))^*$$

The corresponding regular type expression is

```
P3 = (:cat (:0-* string)
           (:0-* symbol (:or (:0-* number) (:0-* string))))
```

We add a state P_3 to the state machine with a transition $P_1 \xrightarrow{string} P_3$.
If we continue calculating the derivatives, we find that we have exhausted all the unique forms.

$$\begin{aligned}
\partial_{symbol} P_1 &= P_1 \\
\partial_{number} P_2 &= P_2 \\
\partial_{string} P_3 &= P_3 \\
\partial_{symbol} P_2 &= P_1 \\
\partial_{symbol} P_3 &= P_1
\end{aligned}$$

From these derivatives we create the following transitions thus completing the transitions in the state machine (Figure 4): $P_1 \xrightarrow{symbol} P_1$, $P_2 \xrightarrow{number} P_2$, $P_3 \xrightarrow{string} P_3$, $P_2 \xrightarrow{symbol} P_1$, and $P_3 \xrightarrow{symbol} P_1$.

The final step is to determine which of the states are nullable.

$$\begin{aligned}
\nu(P_0) &= \nu((symbol \cdot (number^* + string^*))^*) && \text{By (20)} \\
&= \varepsilon && \text{By (7)}
\end{aligned}$$

$$\begin{aligned}
\nu(P_1) &= \nu((number^* + string^*) \cdot (symbol \cdot (number^* + string^*))^*) && \text{By (21)} \\
&= \nu((number^* + string^*) \cdot \varepsilon) && \text{By (7)} \\
&= \nu(number^* + string^*) \\
&= \nu(number^*) \cap \nu(string^*) && \text{By (4)} \\
&= \varepsilon \cap \varepsilon && \text{By (7)} \\
&= \varepsilon
\end{aligned}$$

In similar manner we find that all the expressions are nullable; $\nu(P_0) = \nu(P_1) = \nu(P_2) = \nu(P_3) = \varepsilon$. This means that all the states are final state.

5 The problem of overlapping types

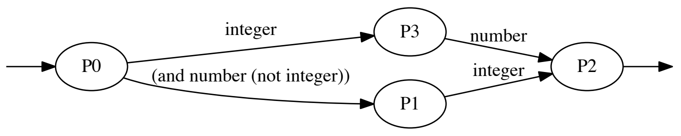


Figure 5: Example DFA with subtypes

In the examples shown thus far the types used have been disjoint types. If the same method is used with types which are intersecting, the automaton which results is not a valid representation of the rational expression. Consider the following rational expression: $P_0 = ((number \cdot integer) \cap (integer \cdot number))$ Clearly the only sequence which matches this expression is a sequence of two integers.

```
(rte (:and (:cat number integer) (:cat integer number)))
```

Unfortunately, when we calculate $\partial_{number} P_0$ and $\partial_{integer} P_0$ we don't arrive at anything useful.

$$\begin{aligned}
\partial_{number} P_0 &= \partial_{number}((number \cdot integer) \ \cap \ (integer \cdot number)) \\
&= \partial_{number}(number \cdot integer) \ \cap \ \partial_{number}(integer \cdot number) \\
&= ((\partial_{number} number) \cdot integer + \nu(integer)) \\
&\quad \cap \ ((\partial_{number} integer) \cdot number + \nu(number)) \\
&= (\varepsilon \cdot integer + \emptyset) \ \cap \ (\emptyset \cdot number + \emptyset) \\
&= integer \ \cap \ \emptyset \\
&= \emptyset \\
\text{Similar,} & \\
\partial_{integer} P_0 &= \emptyset
\end{aligned}$$

The problem is that if a rational type expression is treated blindly as an ordinary rational, then $number \neq integer$ end of story. But if we wish to create a DFA which will allow validation of Common Lisp sequences of objects, rather than simply sequences of type specifiers, we must extend the theory slightly to accommodate intersecting types.

The troublesome rule we are introduced in Figure 3 is equation (11), indicating that $\partial_a b = \emptyset$ for $b \neq a$. The rules in Figure 6 show derivatives of type expressions with respect to particular types. Most notably, Figure 6 augments Figure 3 in the case disjoint types.

$$\begin{aligned}
\partial_A B &= \varepsilon \text{ if } A = B && (21) \\
\partial_A B &= \emptyset \text{ if } A \cap B = \emptyset && (22) \\
\partial_A B & \quad \text{is undefined otherwise.}
\end{aligned}$$

Figure 6: Rules for derivative of regular type expressions

Proof. Arguments justifying (21) and (22).

Let B_{seq} be a non empty set of sequences of length one, each of whose first elements is an object of type B_{type}. $\partial_A B$ by definition is a particular possibly empty subset of the set of suffixes of B_{seq}. Call that subset S. Now, $\partial_A B = \mathit{Suff}\{S\}$. Since every element of B_{seq} has length one, every suffix and consequently every element of S has length zero. The unique zero length sequence is denoted ε. Thus $\partial_A B$ is either ε[3] or $\emptyset$. In particular if $S = \emptyset$ then $\partial_A B = \emptyset$; if $S \neq \emptyset$ then $\partial_A B = \varepsilon$. What remains is to determine for which which cases (21) and (22) is S empty.

(21) Since $A = B_{type}$, $S = B_{seq}$. Since S is not empty, $\partial_A B = \varepsilon$.

(22) $S \subset B_{seq}$ is a set of singleton sequences each of whose element is of type A. B_{seq} is a set of sequences whose first element is of type B_{type}. Since no element of type A is an element of B_{type}, S must be empty. Thus $\partial_A B = \emptyset$.

$\square$

To use these differentiation rules, we note that $\partial_A B$ is undefined when A and B are *partially overlapping*. Practically this means we must only differentiate a given rational expression with respect to disjoint types. Figure 5 shows an automaton expressing the rational expression $P_0 = ((\mathit{number} \cdot \mathit{integer}) \cap (\mathit{integer} \cdot \mathit{number}))$ but only using types for which the derivative is defined. $P_3 = \partial_{\mathit{integer}} P_0$ and $P_1 = \partial_{(\mathit{and\ number\ (not\ integer)})} P_0$. Figure 5 does show transitions from $P_3 \xrightarrow{\mathit{number}} P_2$ and $P_1 \xrightarrow{\mathit{integer}} P_2$ using intersecting types. This is not, however, a violation of the rules in Figure 6 because P_3 and P_1 are different states.

We need an algorithm (in this case implemented in Common Lisp) which takes a list of type specifiers, and computes a list of disjoint sub types, such that union of the two sets of types is the same. *E.g.*, given the list `(integer number)` returns the list `(integer (and number (not integer)))`. Section 6 explains how this is done.

[3] Recall the abuse of notation that ε denotes both the empty word and the set containing the empty word.

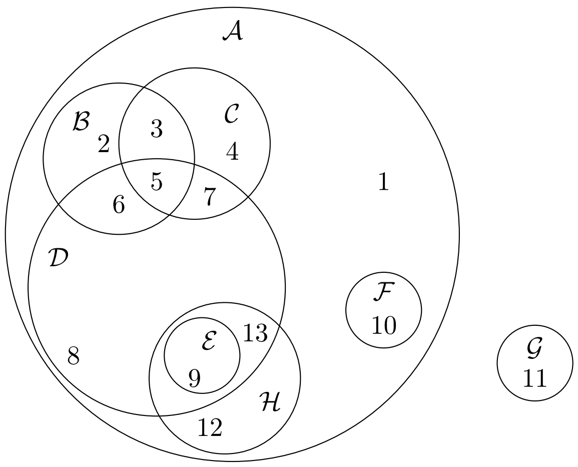


Figure 7: Example Venn Diagram

6 Type segmentation/decomposition

Consider the Venn diagram in Figure 7. The figure shows a set of potentially overlapping sets $\{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{H}\}$. We would like to compute a set of non-empty disjoint subsets of the designated set whose union is he same as the union of the given sets. We wish to find non-empty sets $\mathcal{X}_2, \mathcal{X}_2, ...\mathcal{X}_N$, such that $\mathcal{X}_i \bigcap\limits_{i \neq j} \mathcal{X}_j = \emptyset$ and $\bigcup\limits_{k=1}^{N} \mathcal{X}_k = \mathcal{A} \cup \mathcal{B} \cup \mathcal{C} \cup \mathcal{D} \cup \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{H}$. Such a decomposition is shown in Figure 8.

Disjoint Set	Derived Expression
{ 1 }	$\mathcal{A} \cap \overline{\mathcal{B}} \cap \overline{\mathcal{C}} \cap \overline{\mathcal{D}} \cap \overline{\mathcal{F}} \cap \overline{\mathcal{H}}$
{ 2 }	$\mathcal{B} \cap \overline{\mathcal{C}} \cap \overline{\mathcal{D}}$
{ 3 }	$\mathcal{B} \cap \mathcal{C} \cap \overline{\mathcal{D}}$
{ 4 }	$\mathcal{C} \cap \overline{\mathcal{B}} \cap \overline{\mathcal{D}}$
{ 5 }	$\mathcal{B} \cap \mathcal{C} \cap \mathcal{D}$
{ 6 }	$\mathcal{B} \cap \mathcal{D} \cap \overline{\mathcal{C}}$
{ 7 }	$\mathcal{C} \cap \mathcal{D} \cap \overline{\mathcal{B}}$
{ 8 }	$\mathcal{D} \cap \overline{\mathcal{B}} \cap \overline{\mathcal{C}} \cap \overline{\mathcal{H}}$
{ 9 }	$\mathcal{E}$
{ 10 }	$\mathcal{F}$
{ 11 }	$\mathcal{G}$
{ 12 }	$\mathcal{H} \cap \overline{\mathcal{D}}$
{ 13 }	$\mathcal{D} \cap \mathcal{H} \cap \overline{\mathcal{E}}$

Figure 8: Disjoint Decomposition of Sets from Figure 7

Section 6.1 summarizes the algorithm we used as part of RTE. Section 6.3 discusses an alternate solution by viewing this problem as a variant of the SAT problem. Section 6.4 summarizes an algorithm based on a connectivity graph.

6.1 RTE Algorithm for set disjoint decomposition

The algorithm we use in RTE is shown in Figure 9. This algorithm is straightforward and brute force,[9] and heavily depends on the Common Lisp `subsetp` and the Common Lisp functions shown in Figure 10. A great feature of this algorithm is that it easily fits in 40 lines of Common Lisp code.

1. Let U be the set of sets. Let V be the set of disjoint sets, initially $V = \emptyset$.
2. Identify all the sets which are disjoint from each other and from all the other sets;
3. Remove these sets from U and collect them in V.
4. If there are no sets remaining, you are finished. V is the set of disjoint sets.
5. Otherwise, find one pair of sets, $\mathcal{X} \in U$ and $\mathcal{Y} \in U$, for which $\mathcal{X} \cap \mathcal{Y} \neq \emptyset$.
6. From $\mathcal{X}$ and $\mathcal{Y}$ derive at most three new sets $\mathcal{X} \cap \mathcal{Y}$, $\mathcal{X} \setminus \mathcal{Y}$, and $\mathcal{Y} \setminus \mathcal{X}$, preforming logic reductions as necessary. There are three cases to consider:
   (a) If $\mathcal{X} \subset \mathcal{Y}$, then $\mathcal{X} \cap \mathcal{Y} = \mathcal{X}$ and $\mathcal{X} \setminus \mathcal{Y} = \emptyset$. Thus update U by removing $\mathcal{Y}$, and adding $\mathcal{Y} \setminus \mathcal{X}$.
   (b) If $\mathcal{Y} \subset \mathcal{X}$, then $\mathcal{X} \cap \mathcal{Y} = \mathcal{Y}$ and $\mathcal{Y} \setminus \mathcal{X} = \emptyset$. Thus update U by removing $\mathcal{X}$, and adding $\mathcal{X} \setminus \mathcal{Y}$.
   (c) Otherwise, update U by removing $\mathcal{X}$ and $\mathcal{Y}$, and adding $\mathcal{X} \cap \mathcal{Y}$, $\mathcal{X} \setminus \mathcal{Y}$, and $\mathcal{Y} \setminus \mathcal{X}$.
7. Repeat steps 2 through 6 until $U = \emptyset$, at which point you have collected all the disjoint sets in V.

Figure 9: Algorithm for disjoint set decomposition

Implementing this algorithm is easy when you are permitted to look into the sets. This makes it easy to decide whether two given sets have an intersection. In a programming language, a type can be thought of as a set of (potential) values. In this case where set decomposition is really *type decomposition*, the problem could be trickier. For the algorithm to work, you must have operators to test for set-equality, disjoint-ness, and subset-ness (subtype-ness). It turns out that if you have an empty type and subset-ness predicate, it is possible to express equality and disjoint-ness in terms of them.

The Common Lisp language has flexible type calculus which makes the computation possible. If `T1` and `T2` are Common Lisp type specifiers, then the type specifier `(and T1 T2)` designates the intersection of the types. Likewise `(and T1 (not T2))` and `(and (not T1) T2)` are the two type differences. Furthermore, the Common Lisp function `subtypep` can be used to decide whether two given types are equivalent or disjoint, and `nil` designates the empty type.[3] See Figure 10 for definitions of the Common Lisp functions `type-intersection`, `types-disjoint-p`, and `types-equivalent-p`.

```
(defun type-intersection (T1 T2)
  `(and ,T1 ,T2))

(defun types-disjoint-p (T1 T2)
  (subtypep (type-intersection T1 T2) nil))

(defun types-equivalent-p (T1 T2)
  (multiple-value-bind (T1<=T2 okT1T2) (subtypep T1 T2)
    (multiple-value-bind (T2<=T1 okT2T2) (subtypep T2 T1)
      (values (and T1<=T2 T2<=T1) (and okT1T2 okT2T2)))))
```

Figure 10: Definition of type calculus helper functions

The function `types-disjoint-p` works because a type is empty if it is a subtype of `nil`. The function `types-equivalent-p` works because two types (sets) contain the same elements if each is a subtype (subset) of the other.

See section 6.5 for a description of the performance of this algorithm.

6.2 Sub-type relationship not always recognizable

There is an important caveat. The `subtypep` function is not always able to determine whether the named types have a subtype relationship or not.[3] In such a case, `subtypep` returns `nil` as its second argument. This situation occurs most notably in the cases involving the `satisfies` type specifier. Consider the following example using the types `(satisfies evenp)` and `(satisfies oddp)`.

The first problem we face is that if we attempt to test type membership using such a predicate we may be met errors, such as when the argument of the `oddp` function is not an integer.

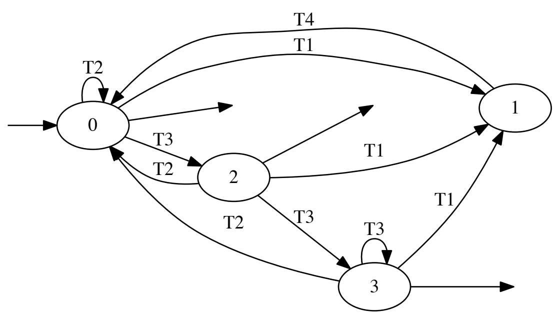
```
(typep 1 '(satisfies oddp))
==> T
(typep 0 '(satisfies oddp))
==> NIL
(typep "hello" '(satisfies oddp))

The value "hello" is not of type INTEGER.
   [Condition of type TYPE-ERROR]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1012F08003}>)

Backtrace:
  0: (ODDP "hello")
  1: (SB-KERNEL:%%TYPEP "hello" #<SB-KERNEL:HAIRY-TYPE (SATISFIES ODDP)> T)
  2: (SB-INT:SIMPLE-EVAL-IN-LEXENV (TYPEP "hello" (QUOTE (SATISFIES ODDP)))
         #<NULL-LEXENV>)
  3: (EVAL (TYPEP "hello" (QUOTE (SATISFIES ODDP))))
```

For this reason it is a little easier to define two types `odd` and `even` using `deftype`. Figure 11 shows the initial type definitions which will be improved upon later. We see very quickly that the system has difficulty reasoning about these types.

```
(deftype odd ()
  '(and integer (satisfies oddp)))
==> ODD

(deftype even ()
  '(and integer (satisfies evenp)))
==> EVEN

(subtypep 'odd 'even)
==> NIL, NIL

(subtypep 'odd 'string)
==> NIL, NIL
```

Figure 11: Initial version of type definitions of odd and even

The `subtypep` function returns `nil` as its second value, indicating that SBCL is unable to



T_1 — `(or string (and odd (not even)))`

T_2 — `(and (not odd) even)`

T_3 — `(and odd even)`

T_4 — `even`

Figure 12: DFA in the case of SATISFIES type

determine whether `odd` is a subtype of `even`. Similarly, SBCL is not able to determine that `odd` is not a subtype of `string`. This behavior is compliant. According to the Common Lisp specification, the `subtypep` function is permitted to return the values `false` and `false` (among other reasons) when at least one argument involves type specifier `satisfies`.[23].

SBCL cannot know that the functions `oddp` and `evenp` never return `true` for the same argument. The human can see that the types `odd` and `even` are non-empty and disjoint, and thus neither is a subtype of the other.

Notice also that `(subtypep 'odd 'string)` returns `nil`. At a glance it would seem that `(and integer (satisfies oddp))` is not a subset of `string`, because `(and integer (satisfies oddp))` is a subset of `integer` which is disjoint from `string`. But there's a catch. The system does not know that `odd` and `even` are non-empty. If a type `A` is empty, then in fact `(and integer A)` is a subtype of `string` because $integer \cap A = \emptyset \subset string$.[12]

Despite the system's inability to peer into the types specified by `satisfies`, we may nevertheless use such types in rational type expressions. Doing so we, get correct but sub-optimal results.

Consider the rational type expression:[4] $((string + odd)^? \cdot even)^*$ which corresponds to the regular type expression:

```
(:0-* (:0-1 (:or string
                 (satisfies oddp)))
      (satisfies evenp))
```

The corresponding DFA is shown in figure 12. Although the results are technically correct, they are more complicated than necessary. In particular, transition label T_1, `(or string (and odd (not even)))` is equivalent to `(or string odd)`. In addition, consider the transition labels T_2 and T_4, `(and (not odd) even)` and `even` respectively. These correspond to the same type.

Furthermore, consider state 3. This state is only reachable via transitions $2 \xrightarrow{T_3} 3$ and $3 \xrightarrow{T_3} 3$. The transition label, T_4 corresponds to type `(and odd even)`, which we know is an empty type; no value is both even and odd. Thus state 3 could be eliminated.

[4]In this case we use the notation of a super-scripted *?* to indicate an *optional* expression. Such notation is common in literature relating to regular language theory.

We can improve the result. Recall the human knows that `odd` is not a subtype of `string`, but that the lisp system does not. The difficulty is that lisp does not know that `odd` is non-empty. We can modify the definitions of the `odd` and `even` types as in Figure 13.

```
(deftype odd ()
  `(and integer
        (or (eql 1) (satisfies oddp))))
==> ODD

(deftype even ()
  `(and integer
        (or (eql 0) (satisfies evenp))))
==> EVEN

(subtypep 'odd 'even)
==> NIL, NIL

(subtypep 'odd 'string)
==> NIL, T
```

Figure 13: Intermediate version of type definitions of odd and even

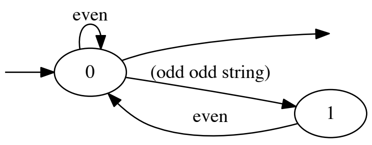


Figure 14: DFA with good deftype

These definitions shown in Figure 13 allow the `subtypep` function to figure out that `odd` is not a subtype of `string`. However, `subtypep` still cannot reason about the relationship of `odd` and `even`.[5] The SBCL implementation of the `subtypep` function is not able to look inside the `oddp` and `evenp` functions to figure out that the types `(satisfies oddp)` and `(satisfies evenp)` are disjoint. However, we can give the system some more clues by stating what is already obvious to the human.

We can further augment the definitions to allow `subtypep` to reason about the relation of `odd` and `even`.

Given the definitions of the types `even` and `odd` in figure 15, the `disjoint-types-p` function is able to figure out that types such as `string` and `odd` are disjoint.

With the these final type definitions, the state machine representing the expression `(:0-* (:0-1 (:or string odd)) even)` is shown in Figure 14.

It is perhaps worth repeating that the state machines in Figures 12 and 14 recognize the same sequences. The types specifiers marking the transitions of the former are correct, but less efficient those in the latter. Additionally the number of states has been reduced in the latter to two states. However, to achieve this minimal state machine, it is necessary to supply redundant information in the type definitions.

[5]SBCL version 1.3.0 has a bug in which `(subtypep 'odd 'even)` returns `NIL,T`. *I.e.*, it dubiously gets the correct answer. The reasoning is faulty and bug number 1528837 has been filed reporting the issue.

```
(deftype odd ()
  '(and integer
        (not (satisfies evenp))
        (satisfies oddp)))
==> ODD

(deftype even ()
  '(and integer
        (not (satisfies oddp))
        (satisfies evenp)))
==> EVEN

(subtypep 'odd 'even)
==> NIL, T

(subtypep 'odd 'string)
==> NIL, T
```

Figure 15: Final version of type definitions of odd and even

6.3 Set disjoint decomposition as SAT problem

This problem of how to decompose sets, like those shown in Figure 7 into disjoint subsets as shown in Figure 8 can be views as a variant of the well known *Satisfiability Problem*, commonly called SAT.[11] The problem is this: given a Boolean equation in n variables, find a solution. This is to say: find an assignment (either *true* or *false*) for each variable which makes the equation evaluate to *true*. This problem is known to be NP-Complete.

The approach is to consider the correspondence between the solutions of the Boolean equation: $A+B+C+D+E+F+G+H$, versus the set of subsets of $\mathcal{A} \cup \mathcal{B} \cup \mathcal{C} \cup \mathcal{D} \cup \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{H}$. Just as we can enumerate the $2^8 - 1 = 255$ solutions of $A + B + C + D + E + F + G + H$ as we can analagously enumerate the subsets of $\mathcal{A} \cup \mathcal{B} \cup \mathcal{C} \cup \mathcal{D} \cup \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{H}$.

discard	0000 0000	$\overline{\mathcal{A}} \cap \overline{\mathcal{B}} \cap \overline{\mathcal{C}} \cap \overline{\mathcal{D}} \;\cap\; \overline{\mathcal{E}} \cap \overline{\mathcal{F}} \cap \overline{\mathcal{G}} \cap \overline{\mathcal{H}}$
1	1000 0000	$\mathcal{A} \cap \overline{\mathcal{B}} \cap \overline{\mathcal{C}} \cap \overline{\mathcal{D}} \;\cap\; \overline{\mathcal{E}} \cap \overline{\mathcal{F}} \cap \overline{\mathcal{G}} \cap \overline{\mathcal{H}}$
2	0100 0000	$\overline{\mathcal{A}} \cap \mathcal{B} \cap \overline{\mathcal{C}} \cap \overline{\mathcal{D}} \;\cap\; \overline{\mathcal{E}} \cap \overline{\mathcal{F}} \cap \overline{\mathcal{G}} \cap \overline{\mathcal{H}}$
3	1100 0000	$\mathcal{A} \cap \mathcal{B} \cap \overline{\mathcal{C}} \cap \overline{\mathcal{D}} \;\cap\; \overline{\mathcal{E}} \cap \overline{\mathcal{F}} \cap \overline{\mathcal{G}} \cap \overline{\mathcal{H}}$
	...	...
254	1111 1110	$\mathcal{A} \cap \mathcal{B} \cap \mathcal{C} \cap \mathcal{D} \;\cap \mathcal{E} \cap \mathcal{F} \cap \mathcal{G} \cap \overline{\mathcal{H}}$
255	1111 1111	$\mathcal{A} \cap \mathcal{B} \cap \mathcal{C} \cap \mathcal{D} \;\cap \mathcal{E} \cap \mathcal{F} \cap \mathcal{G} \cap \mathcal{H}$

Figure 16: Correspondence of Boolean true/false equation with boolean set equation

The approach here is to consider at every possible solution of the Boolean equation: $A + B + C+D+E+F+G+H$. There are $2^8 - 1 = 255$ such solutions, because every 8-tuple of 0's and 1's is a solution except 0000 0000. If we consider the enumerated set of solutions: 1000 0000, 0100 0000, 1100 0000, ... 1111 1110, 1111 1111. We can analagously enumerate the potential subsets of the union of the sets shown in Figure 7: $\mathcal{A} \cup \mathcal{B} \cup \mathcal{C} \cup \mathcal{D} \cup \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{H}$. Each is a potential solution represents an intersection of sets in $\{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{H}\}$. Such a correspondence is shown in Figure 16.

It remains only to eliminate the intersections which can be proven to be empty. For example, we see in 7 that $\mathcal{A}$ and $\mathcal{G}$ are disjoint, which implies $\emptyset = \mathcal{A} \cap \mathcal{G}$, which further implies line 1 of Table 16, $\emptyset = \mathcal{A} \cap \overline{\mathcal{B}} \cap \overline{\mathcal{C}} \cap \overline{\mathcal{D}} \;\cap\; \overline{\mathcal{E}} \cap \overline{\mathcal{F}} \cap \overline{\mathcal{G}} \cap \overline{\mathcal{H}}$.

In this as all SAT problems, certain of these 2^8 possibilities can be eliminated because of known constraints. The constraints are derived from the known subset and disjoint-ness relations of the given sets. Looking at the Figure 7 we see that $\mathcal{E} \subset \mathcal{H}$, which means that $\mathcal{E} \cap \overline{\mathcal{H}} = \emptyset$. So we know that all solutions where $H = 0$ and $E = 1$ can be eliminated. This means we can update the equation by multiplying (Boolean multiply) by $\overline{E\overline{H}}$: $(A + B + C + D + E + F + G + H) \cdot \overline{E\overline{H}}$.

Additionally notice that $\mathcal{A}$ and $\mathcal{G}$ are disjoint. So no solution may contain $A = 1$ and $G = 1$. This corresponds to a new constraint: $\overline{AG}$: $(A + B + C + D + E + F + G + H) \cdot \overline{E\overline{H}} \cdot \overline{AG}$.

There are as many as $\frac{8 \cdot 7}{2} = 28$ possible constraints imposed by pair relations. For each $\{X, Y\} \subset \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{H}\}$:

Subset If $X \subset Y$, multiply the by constraint $\overline{X\overline{Y}} = (\overline{X} + Y)$

Super-set If $Y \subset X$, multiply by the constraint $\overline{\overline{X}Y} = (X + \overline{Y})$

Disjoint If $X \cap Y = \emptyset$, multiply by the constraint $\overline{XY} = (\overline{X} + \overline{Y})$.

Otherwise no constraint.

A SAT solver will normally find one solution. That's just how they traditionally work. But the SAT flow can easily be extended so that once a solution is found, a new constraint can be generated by logically negating that solution, allowing the SAT solver to find a second solution. For example, when it is found that 1111 0000 (corresponding to $\mathcal{A} \cap \mathcal{B} \cap \mathcal{C} \cap \mathcal{D} \cap \overline{\mathcal{E}} \cap \overline{\mathcal{F}} \cap \overline{\mathcal{G}} \cap \overline{\mathcal{H}}$), the equation can be multiplied by the new constraint $(A\ B\ C\ D\ \overline{E}\ \overline{F}\ \overline{G}\ \overline{H})$, allowing the SAT solver to find another solution.

The process continues until there are no more solutions.

As a more concrete example of how the SAT approach works when applied to Common Lisp types, consider the case of the three types `array`, `sequence`, and `vector`. Actually, `vector` is the intersection of `array` and `sequence`.

First the SAT solver constructs (explicitly or implicitly) the set of candidates corresponding to the lisp types.

```
(and array       sequence       vector)
(and array       sequence       (not vector))
(and array       (not sequence) vector)
(and array       (not sequence) (not vector))
(and (not array) sequence       vector)
(and (not array) sequence       (not vector))
(and (not array) (not sequence) vector)
(and (not array) (not sequence) (not vector))
```

The void one `(and (not array) (not sequence) (not vector))` can be immediately disregarded.

Since `vector` is a subtype of `array`, all types which include `((not array) vector)` can be disregarded: `(and (not array) sequence vector)` and `(and (not array) (not sequence) vector)`. Furthermore since `vector` is a subtype of `sequence`, all types which include `((not sequence) vector)` can be disregarded. `(and array (not sequence) vector)` and `(and (not array) (not sequence) vector)` (which has already been eliminated by the previous step). The remaining ones are:

```
(and array       sequence       vector)       = vector
(and array       sequence       (not vector)) = nil
(and array       (not sequence) (not vector)) = (and array (not vector))
(and (not array) sequence       (not vector)) = (and sequence (not vector))
```

The algorithm returns a false positive. Unfortunately, this set still contains the `nil`, empty, type `(and array sequence (not vector))`. Figure 17 shows the relation of the Common Lisp types `array`, `vector`, and `sequence`. We can see that `vector` is the intersection of `array` and

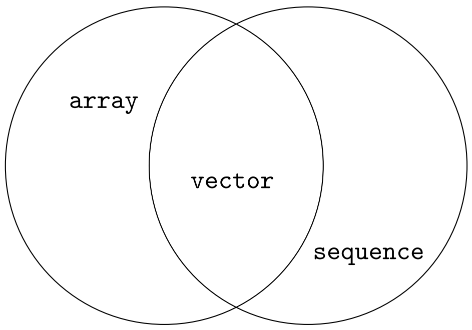


Figure 17: Relation of vector, sequence, and array

`sequence`. The algorithm discussed above failed to introduce a constraint corresponding to this identity which implies that $array \cap sequence \cap \overline{vector} = \emptyset$.

It seems the SAT algorithm greatly reduces the search space, but is not able to give the minimal answer. The resulting types must still be tested for vacuity. This is easy to do, just use the `subtypep` function to test whether the type is a subtype of `nil`. E.g., `(subtypep '(and array sequence (not vector)) nil)` returns `t`. Again, as mentioned in section 6.2, there are cases where the `subtypep` will not be able to determine the vacuity of a set. Consider the example: `(and fixnum (not (satisifies oddp)) (not (satisfies evenp)))`.

See section 6.5 for a description of the performance of this algorithm.

6.4 Set disjoint decomposition as graph problem

This algorithm is semantically very similar to the algorithm shown in 6.1 but rather than relying on Common Lisp primitives to make decisions about connectivity of sets/types, it initializes a graph representing the initial relationships, and thereafter manipulates the graph maintaining connectivity information. This algorithm is more complicated in terms of lines of code, 250 lines of Common Lisp code as opposed to 40 lines.

This more complicated algorithm is presented here for two reasons. (1) It has much faster execution times, especially for larger sets types. (2) We hope that presenting the algorithm in a way which obviates the need to use Common Lisp primitives makes it evident how the algorithm might be implemented in a programming language other than Common Lisp.

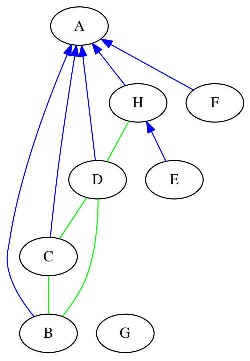


Figure 18: Partially directed topology graph, initial state 0

Figure 18 shows a graph representing the topology (connectedness) of the diagram shown in Figure 7. Blue lines are drawn from sub-set to super-set. Green lines are drawn between two sets which touch but fail to have a sub-set/super-set relationship.

The algorithm proceeds by breaking the green and blue connections in controlled ways until all the nodes become isolated. There are several cases to consider.

Strict sub-set Blue arrows indicate sub-set/super-set relations, they point from a sub-set to a super-set. A blue arrow from X to Y may be eliminated if conditions are met:

- X has no blue arrows pointing to it, and
- X has no green lines touching it.

On eliminating the blue arrow, replace the label Y by $Y \cap \overline{X}$.

Touching connections Green lines indicate partially overlapping sets. A green line connecting X and Y may be broken if the following condition is met:

- Neither X nor Y has a blue arrow pointing to it; *i.e.* neither is a super-set of something else in the graph.

Eliminating the green line *separates* X and Y. To do this X and Y must be replaced and a new node must be added to the graph.

- Introduce new node labeled $X \cap Y$.
  - Draw blue arrows from this node, $X \cap Y$, to all the nodes which either X or Y points to. *I.e.*, the super-sets of $X \cap Y$ are the union of the super-sets of X and of Y.
  - Draw green lines from $X \cap Y$ to all nodes which both X and Y connect to. *I.e.* the connections to $X \cap Y$ are the intersection of the connections of X and of Y.
  - (Exception) If there would be a green line between $X \cap Y$ and some node for which there is already a blue arrow, omit the green line.
- $X \leftarrow X \cap \overline{Y}$. *I.e.* replace X with its relative complement with respect to Y.
- $Y \leftarrow \overline{X} \cap Y$. *I.e.* replace Y with its relative complement with respect to X.

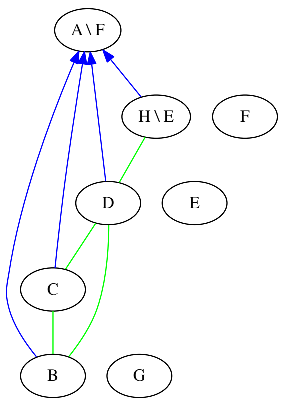


Figure 19: Partially directed topology graph, state 1

E and H in Figure 18 meet the *strict sub-set* conditions, thus the arrow connecting them can be eliminated, and H replace $H \leftarrow H \cap \overline{E}$, this is denoted as `H\E` in Figure 19.

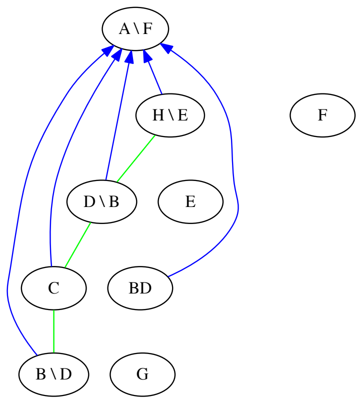


Figure 20: Partially directed topology graph, state 2

Nodes such as B and D in Figure 18 (and also Figure 19) meet the *touching connections* conditions and can thus be *separated* by breaking the connection (green line). Figure 20 shows the result of this operation. The new node $B \cap D$ has been introduced (labeled `BD` in the figure) , with blue arrow pointing to $A \cap \overline{F}$ (denoted `A\F` in the figure). B and E are relabeled as well. B is relabeled as $B \cap \overline{D}$, (denoted `B\D` in the figure). E is relabeled as $\overline{B} \cap D$, (denoted `D\B` in the figure).

These graph operations should continue until all the nodes have become isolated.

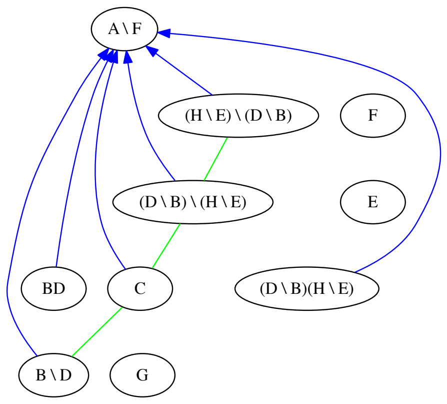


Figure 21: Partially directed topology graph, state 3

We continue the segmentation algorithm a couple more steps. In Figure 20, consider eliminating the green connection between nodes `D\B` and `H\E`, corresponding to $\overline{B} \cap D$ and $\overline{E} \cap H$, resulting in the graph shown in Figure 21. In this case we must introduce a new node $(\overline{B} \cap D) \cap (\overline{E} \cap H)$, corresponding to `(D\B)(H\E)` in the figure. We must also relabel $\overline{B} \cap D$ as $(\overline{B} \cap D) \cap \overline{\overline{E} \cap H}$, and relabel $\overline{E} \cap H$ as $\overline{\overline{E} \cap H} \cap (\overline{B} \cap D)$. These relabeled nodes correspond in the figure respectively to and `(D\B)\(H\E)` and `(H\E)\(D\B)`.

Next we eliminate the blue arrow from to in Figure 21 resulting in the graph in Figure 22. In this operation the node `A\F` is relabeled to `(A\F)\((D\B)(H\E))`.

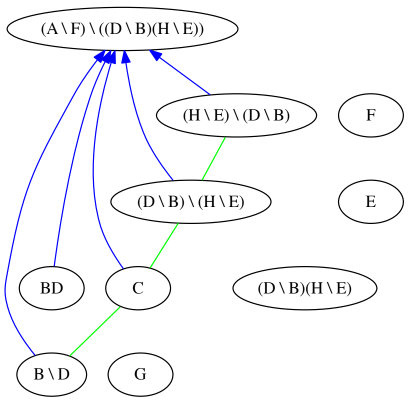


Figure 22: Partially directed topology graph, state 4

From Figure 22 it should be becoming clear that the complexity of the Boolean expressions in each node is becoming more complex. If we continue this procedure, eliminating all the blue arrows and green connecting lines, we will end up with 13 isolated nodes (each time a green line is eliminated one additional node is added).

There are some subtle corner cases which may not be obvious. In particular are some relatively exotic cases which we won't illustrate here. It is possible in these situations to end up with some disjoint subsets which are empty. It is possible also that the same subset is derived by two different operations in the graph, but whose equations are very different. To identify each of these cases, each of the resulting sets must be checked for vacuity, and uniqueness. No matter which programming language the algorithm is implemented, it is necessary to be implement these two checks.

In Common Lisp there are two possible ways to check for vacuity, *i.e.* to detect whether a type is empty. (1) Symbolically reduce the type specifier, *e.g.* `(and fixnum (not fixnum))` to a canonical form with is `nil` in case the specifier specifies the `nil` type. (2) Use the `subtypep` function to test whether the type is a subtype of `nil`. To test whether two specifiers specify the same type there are two possible approaches in Common Lisp. (1) Symbolically reduce each expression such as `(or integer number string)` and `(or string fixnum number)` to canonical form, and compare the results with the `equal` function. (2) Use the `subsetp` function twice to test whether each is a subtype of the other.

See section 6.5 for a description of the performance of this algorithm.

6.5 Performance analysis of type decomposition

Sections 6.1, 6.3 and 6.4 explained three different algorithm for calculating type decomposition. We look here at some performance charactistics of the three algorithms.

Just to give a broad idea of the performance difference of the algorithms we partitioned the type specifiers which denote types which are a subtype of `number`. SBCL has 22 types (excluding `nil`) whose names are in the `CL` package which are subtypes of `number`.

```
(array-rank     array-total-size  bignum        bit           char-code
 char-int       complex           double-float  fixnum        float
 float-digits   float-radix       integer       long-float    number
 ratio          rational          real          short-float
 signed-byte    single-float      unsigned-byte)
```

To partition these into the 15 disjoint types the SAT function required 30 seconds, and the RTE algorithm 1.5 seconds.

```
(bit
 complex
 double-float
 float-radix
 ratio
 single-float
 (and array-rank (not float-digits))
 (and array-total-size (not char-code))
 (and bignum (not unsigned-byte))
 (and bignum unsigned-byte)
 (and char-code (not array-rank))
 (and float-digits (not bit)
 (not float-radix))
 (and number (not complex) (not float)
 (and rational (not bignum) (not ratio) (not unsigned-byte))
 (not rational))
 (and unsigned-byte (not array-total-size) (not bignum)))
```

7 Application use cases

The following subsections 7.1, 7.2, 7.3, and 7.4 illustrate applications for regular type expressions.

7.1 RTE based string regular expressions

The `rte` type can be used to perform simple string regular expression checking.

To filter a given list of strings, retaining only the ones match a particular regular expression using `rte` we implemented the following two function `find-matches`. The function whose code is shown in Figure 23 exploits the fact that Common Lisp strings are sequences of characters to filter a given list of strings for those matching the regular expression `"(ab)*z*(ab)*"`.

This attempt to represent a string regular expression as a regular type expression is indeed possible as shown in Figure 23 but admittedly cumbersome. We provide the function `regexp-to-rte` as a solution to convert string regular expressions to regular type expressions. An example of its use is shown in Figure 24

The `regexp-to-rte` function does not implement full Perl compatible regular expressions as provided in CL-PPCRE[24]. Doing so would be a daunting task and would require departure from rational language theory as some of the operators provided by Perl compatible regular expressions do not conform to rational language operations.[6]. Rather, we implemented a small but powerful subset. The subset we chose is the one whose grammar is provided in publicly available lecture notes by Robert Cameron[7]. Starting with this published context free grammar, we were able to use the `CL-Yacc`[8] package written by Juliusz Chroboczek to parse a regular expression and convert it to a regular type expression.

Using the `regexp-to-rte` function we can simplify the function `find-matches-rte` as shown in Figure 25.

One potential application of such type of regular expression matching would be when matching arbitrary sequences rather than strings only.

For a short analysis of performance differences between `rte` and CL-PPCRE, see section 8.5.

7.2 Test cases based on extensible sequences

Climb[22] is an image processing library implemented in Common Lisp. It represents digital images in a variety of internal formats, including as a two dimensional array of pixels, or what

[6]Theoretically speaking a correct regular expression matchers is implementable as a finite state machine, with no additional memory required. One example where Perl regular expressions depart from theoretical rational expressions is that they require additional memory to store sub-matches and refer to them later in the expression

```
(defvar *data* '("ababababzabab"
                 "ababababzabababab"
                 "ababababzabababab"
                 "ababababzzzzabababab"
                 "abababababababzzzzzzabababab"
                 "abababababababababzzzzzzabababab"
                 "ababababababababababababababzzzzzzabababab"
                 "ababababzzzzzzababababababababzzzzzzabababab"
                 ))

(defun find-matches (data)
  (remove-if-not (lambda (str)
                   (typep str '(rte (:0-* (member #\a #\b))
                                    (:0-* (eql #\z))
                                    (:0-* (member #\a #\b)))))
                 data))

(find-matches *data*)
==>
("ababababzabab"
 "ababababzabababab"
 "ababababzabababab"
 "ababababzzzzabababab"
 "abababababababzzzzzzabababab"
 "abababababababababzzzzzzabababab"
 "ababababababababababababababzzzzzzabababab")
```

Figure 23: RTE based Function for string regular expression matching

```
(regexp-to-rte "(ab)*z*(ab)*")
==>
  (:cat (:0-* (member #\a #\b))
        (:0-* (eql #\z))
        (:0-* (member #\a #\b)))
```

Figure 24: Example usage of `regexp-to-rte`

conceptually serves the function of a 2-d array. The image potentially is populated with pixel values such as RGB objects or gray-scale scalars, but may also have boundary elements which may not be valid pixel values. Certain image calculations are expected to calculate new images. For testing purposes we would like to make assertions about rows and columns of the two dimensional arrays. For example, we'd like to be able to assert that the row vectors and column vectors (excluding the border elements) of a given image are RGB (red-green-blue) values, and the row and column vectors in the calculated image are gray-scale values. Unfortunately, Common Lisp 2-d arrays are not sequences. This means that 2-d image arrays are not natively compatible with regular type expressions.

To solve this problem, we exploit a feature of SBCL called *Extensible Sequences*[15, 20]. In Figure 26 we have created CLOS classes[23] named `row-vector` and `column-vector` which implement the sequence protocol, but which access a backing 2-d Common Lisp array. To implement the sequence protocol, an application such as Climb must implement methods on generic functions such as `length`, `elt`, and `(setf elt)`.

The unit tests for Climb are implementing using `Lisp-Unit`[21]. Figure 27 shows an example test which loads an RGB image named `"lena128.bmp"` and makes some assertions about the format of the internal lisp data structures. In particular it views the image as a sequence of

```
(defun find-matches (data)
  (remove-if-not (lambda (str)
                   (typep str `(rte ,(regexp-to-rte "(ab)*z*(ab)*"))))
                 data))
```

Figure 25: Function `find-matches` simplified using `regexp-to-rte`

```
(defclass 2d-array-as-sequence (sequence standard-object)
  ((2d-array :initarg :2d-array :reader 2d-array)))

(defclass row-vector (2d-array-as-sequence)
  ((row :type fixnum :initarg :row :accessor row)))

(defclass column-vector (2d-array-as-sequence)
  ((column :type fixnum :initarg :column :accessor column)))

(defmethod sequence:length ((seq column-vector))
  (array-dimension (2d-array seq) 0))

(defmethod sequence:elt ((seq column-vector) row)
  (aref (2d-array seq) row (column seq)))

(defmethod (setf sequence:elt) (value (seq column-vector) row)
  (setf (aref (2d-array seq) row (column seq))
        value))
```

Figure 26: Class and method definitions extending the definition of sequence

row-vectors, the first and last of which may contain any content `(rte (:0-* t))`, but the rows in between are of the form `(rte t (:0-* rgb) t)`.

```
(define-test io/2d-array-b
  (let* ((rgb-image (image-load (pathname (absolutepath "share/images/"
                                                         "lena128.bmp"))))
         (seq (make-instance '2d-array:vector-of-rows
                             :2d-array (climb::image-raw-data rgb-image))))

    (assert-true (climb::image-raw-data rgb-image))

    (assert-true (typep seq
                        '(rte (rte (:0-* t))
                              (:0-* (rte t (:0-* rgb) t))
                              (rte (:0-* t)))))))
```

Figure 27: Climb Unit Test using RTE to check image content

7.3 Complex pattern matching, recognizing correct lambda lists

As a complex yet realistic example we look at how to use regular type expressions to check Common Lisp lambda lists.

Common Lisp specifies several different kinds of lambda lists, used for different purposes. For example, the *ordinary lambda list* which is used to define lambda functions, the *macro lambda list* for defining macros, and the *destructuring lambda list* for use with `destructuring-bind`. Each of these lambda lists differs in its syntax rules.

```
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 28: CLHS Syntax of ordinary lambda list

The simplest kind of lambda list is the *ordinary lambda list*. Figure 28 shows the syntax rule for the ordinary lambda list, and Figure 29 shows examples of ordinary lambda lists which obey the specification, but the latter two may not mean what you think they mean.

```
(defun F1 (a b &rest other-args &key x (y 42) ((:Z U) nil u-used-p)
  ...)

(defun F2 (a b &key x &rest other-args)
  ...)

(defun F3 (a b &key ((Z U) nil u-used-p))
  ...)
```

Figure 29: Examples of ordinary lambda lists

The function `F2`, (from Figure 29) according to the Common Lisp specification, is a function with three possible keyword arguments, `x`, `&rest`, and `other-args`, which can be referenced at the call site with a bizarre function call such as `(F2 1 2 :x 3 :&rest 4 :other-args 5)`. However, what the programmer probably meant was one keyword argument and an `&rest` argument named `other-args`. This issue is quite subtle. In fact, some Common Lisp implementations consider this such a bizarre situation that they divert from the specification and flag this type of definition as a compilation error. Figure 30 shows the reaction of SBCL.

The function `F3` (from Figure 29) is defined with an *unconventional* `&key` which is not a symbol in the `keyword` package but rather in the current package. Thus the variable `U` is referenced from the call-site as `(F3 1 2 'Z 3)` rather than `(F3 1 2 :Z 3)`.

Because of these potentially confusing situations, we define what we call *conventional ordinary lambda list*. Figure 32 shows a sample implementation of the type `conventional-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 are indeed valid lambda lists for `defun`, although Common Lisp allows a more relaxed syntax. Figure 33 showing the corresponding DFA gives a vague idea of the complexity of the matching algorithm.

The *conventional ordinary lambda list* differs slightly from the *ordinary lambda lists*, in several

```
CL-USER>(defun F2 (a b &key x &rest other-args) nil)

misplaced &REST in lambda list: (A B &KEY X &REST OTHER-ARGS)
   [Condition of type SB-INT:SIMPLE-PROGRAM-ERROR]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1012F08003}>)

Backtrace:
  0: ((LAMBDA NIL :IN SB-C::ACTUALLY-COMPILE))
  1: ((FLET SB-C::WITH-IT :IN SB-C::%WITH-COMPILATION-UNIT))
```

Figure 30: Attempt to compile F2 in SBCL

```
(deftype var ()
  '(and symbol
        (not (or keyword
                 (member t nil)
                 (member &optional &key &rest &allow-other-keys &aux
                         &body &whole &env)))))
```

Figure 31: Definition of the var type

aspects. Figure 28 is an excerpt from the Common Lisp specification. The definition in 32 implements this specification with the exceptions explained in below.

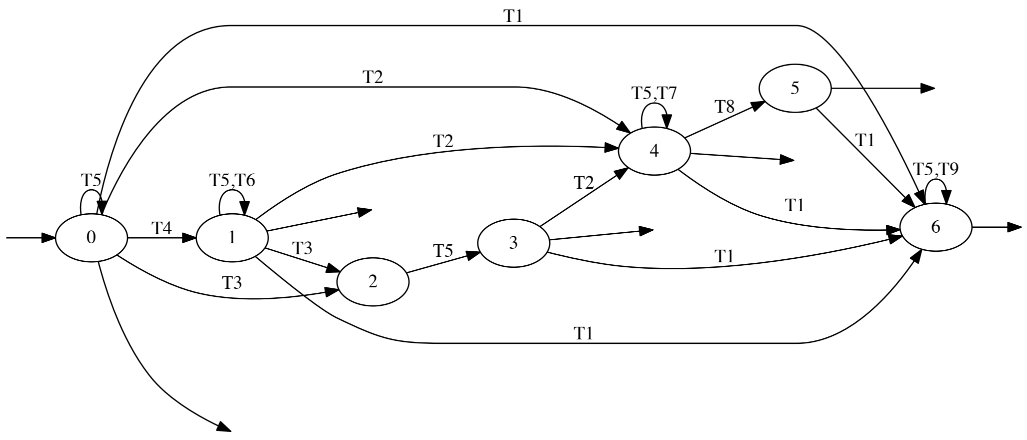
- Careful reading of the Common Lisp specification reveals that a lambda list such as `(&aux &key)` declares a variable named `&key` as an auxiliary variable. The `conventional-ordinary-lambda-list` type accepts only variable names of type `var` whose type definition is shown in Figure 31. In particular variable names such as `&key` are excluded.

- Common Lisp implementations are free to implement semantics for additional lambda list keywords. Figure 32 only implements: `&optional`, `&rest`, `&key`, `&allow-other-keys`, and `&aux`.

- The Common Lisp specification allows an ordinary lambda list to use non-keyword keyword-name symbols such as `(&key ((x y)))` to mean that the variable name is `y`, but the call-site syntax should use the non-keyword symbol `x`. This usage is allowed but unconventional. Figure 32 requires the keyword-name be a keyword recognizing a more conventional lambda list such as `(&key ((:x y)))`.

```
(deftype conventional-ordinary-lambda-list ()
  (let* ((optional-var '(:or var (:and list (rte (:1 var
                                                      (:? t
                                                          (:? var)))))))
         (optional `(:cat (eql &optional) (:* ,optional-var)))
         (rest '(:cat (eql &rest) var))
         (key-var '(:or var
                         (:and list
                               (rte (:or var (cons keyword
                                                  (cons var null)))
                                         (:? t
                                             (:0-1 var))))))
         (key `(:cat (eql &key)
                      (:0-* ,key-var)
                      (:0-1 (eql &allow-other-keys))))
         (aux-var '(:or var (:and list (rte (:1 var (:? t))))))
         (aux `(:cat (eql &aux) (:* ,aux-var))))

    `(rte
      (:* var)
      (:? ,optional)
      (:? ,rest)
      (:? ,key)
      (:? ,aux))))
```

Figure 32: Definition of the conventional-ordinary-lambda-list type



T_1 — `(eql &aux)`

T_2 — `(eql &key)`

T_3 — `(eql &rest)`

T_4 — `(eql &optional)`

T_5 — `var`

T_6 — `(and list (rte var (:0-1 t (:0-1 var))))`

T_7 — `(and list (rte (:or var (cons keyword (cons var null))) (:0-1 t (:0-1 var))))`

T_8 — `(eql &allow-other-keys)`

T_9 — `(and list (rte var (:0-1 t)))`

Figure 33: DFA recognizing conventional ordinary lambda list

7.4 List destructuring - `destructuring-case`

The reader may notice a similarity to XML pattern matching in the XDuce domain specific language.[10] The XDuce language allows the programmer to define a set of functions with various lambda lists, each of which serves as a pattern available match target structure within an XML document. Which function gets executed depends on which lambda list matches the data found in the XML data structure.

The existence of the `rte` type makes it possible to use `destructuring-bind` and `type-case` together in a similar way to pattern matching in XDuce. Notice in the code in Figure 34 that each `rte` clause of the `typecase` includes a call to `destructuring-bind` which is related. The function `F` is implemented such that the object being destructured is assured to be of the format expected by the corresponding destructuring lambda list.

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

Figure 34: Using rte with destructuring-bind

We provide a macro `destructuring-case` which combines the capability of Common Lisp `destructuring-bind` and `type-case`. Moreover, `destructuring-case`, constructs the `rte` type specifiers in an intelligent way, taking into account the structure of the destructuring lambda list and any given type declarations. An example usage of `destructuring-case` is shown in Figure 35.

```
(defun F (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 (:0-* string))))
     ...)))
```

Figure 35: Using rte with destructuring-case

This macro, via the function `destructuring-lambda-list-to-rte`, provided by the `rte` package, is able to parse any valid destructuring lambda list, and convert it to to a regular type expression. The destructuring lambda lists are allowed to contain any valid syntax, such as `&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)`.

One advantage of `destructuring-case` is that the regular type expression may get complicated and tedious to create by hand. Consider the call to `destructuring-case` with type declarations shown in Figure 36.

These two destructuring lambda lists correspond to the regular type expressions shown in Figures 37 and 38. To better understand the control flow of matching these two regular type

```
(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))
   ...))
```

Figure 36: Sample destructuring-case use case

expressions, especially the handling of `&key` with and without `&allow-other-keys`, Figures 39, 40, and 41 are provided. These figures show the DFAs which implement the regular type expressions. Also note that the two DFAs are topologically equivalent, even though the type specifiers on the corresponding state transitions are different.

```
(:cat (:cat fixnum (:and list (rte (:cat fixnum fixnum))))
      (:and
        (:0-* keyword t)
        (:or
          (:cat (:0-1 (eql :x) symbol (:0-* (not (member :y :z)) t))
                (:0-1 (eql :y) string (:0-* (not (eql     :z))    t))
                (:0-1 (eql :z) list   (:0-* t t)))
          (:cat (:0-1 (eql :y) string (:0-* (not (member :x :z)) t))
                (:0-1 (eql :x) symbol (:0-* (not (eql     :z))    t))
                (:0-1 (eql :z) list   (:0-* t t)))
          (:cat (:0-1 (eql :x) symbol (:0-* (not (member :y :z)) t))
                (:0-1 (eql :z) list   (:0-* (not (eql     :y))    t))
                (:0-1 (eql :y) string (:0-* t t)))
          (:cat (:0-1 (eql :z) list   (:0-* (not (member :x :y)) t))
                (:0-1 (eql :x) symbol (:0-* (not (eql     :y))    t))
                (:0-1 (eql :y) string (:0-* t t)))
          (:cat (:0-1 (eql :y) string (:0-* (not (member :x :z)) t))
                (:0-1 (eql :z) list   (:0-* (not (eql     :x))    t))
                (:0-1 (eql :x) symbol (:0-* t t)))
          (:cat (:0-1 (eql :z) list   (:0-* (not (member :x :y)) t))
                (:0-1 (eql :y) string (:0-* (not (eql     :x))    t))
                (:0-1 (eql :x) symbol (:0-* t t))))))
```

Figure 37: Regular type expression matching destructuring lambda list Case-1

```
(:cat (:cat fixnum (:and list (rte (:cat fixnum fixnum))))
      (:and (:0-* keyword t)
        (:or
         (:cat (:0-1 (eql :x) symbol (:0-* (eql :x)           t))
               (:0-1 (eql :y) string (:0-* (member :y :x)     t))
               (:0-1 (eql :z) list   (:0-* (member :z :y :x) t)))
         (:cat (:0-1 (eql :y) string (:0-* (eql     :y)       t))
               (:0-1 (eql :x) symbol (:0-* (member :x :y)     t))
               (:0-1 (eql :z) list   (:0-* (member :z :x :y) t)))
         (:cat (:0-1 (eql :x) symbol (:0-* (eql     :x)       t))
               (:0-1 (eql :z) list   (:0-* (member :z :x)     t))
               (:0-1 (eql :y) string (:0-* (member :y :z :x) t)))
         (:cat (:0-1 (eql :z) list   (:0-* (eql     :z)       t))
               (:0-1 (eql :x) symbol (:0-* (member :x :z)     t))
               (:0-1 (eql :y) string (:0-* (member :y :x :z) t)))
         (:cat (:0-1 (eql :y) string (:0-* (eql     :y)       t))
               (:0-1 (eql :z) list   (:0-* (member :z :y)     t))
               (:0-1 (eql :x) symbol (:0-* (member :x :z :y) t)))
         (:cat (:0-1 (eql :z) list   (:0-* (eql     :z)       t))
               (:0-1 (eql :y) string (:0-* (member :y :z)     t))
               (:0-1 (eql :x) symbol (:0-* (member :x :y :z) t))))))
```

Figure 38: Regular type expression matching destructuring lambda list Case-2

T_1 – `t`

T_2 – `list`

T_3 – `fixnum`

T_4 – `symbol`

T_5 – `keyword`

T_6 – `string`

T_7 – `(and list (rte (:cat fixnum fixnum)))`

T_8 – `(eql :x)`

T_9 – `(eql :y)`

T_{10} – `(eql :z)`

T_{11} – `(member :x :y)`

T_{12} – `(member :x :z)`

T_{13} – `(member :y :z)`

T_{14} – `(member :x :y :z)`

T_{15} – `(and keyword (not (eql :x)))`

T_{16} – `(and keyword (not (eql :y)))`

T_{17} – `(and keyword (not (eql :z)))`

T_{18} – `(and keyword (not (member :x :y)))`

T_{19} – `(and keyword (not (member :x :z)))`

T_{20} – `(and keyword (not (member :y :z)))`

Figure 39: Transition types for Case-1 Figure 40 and Case-2 Figure 41

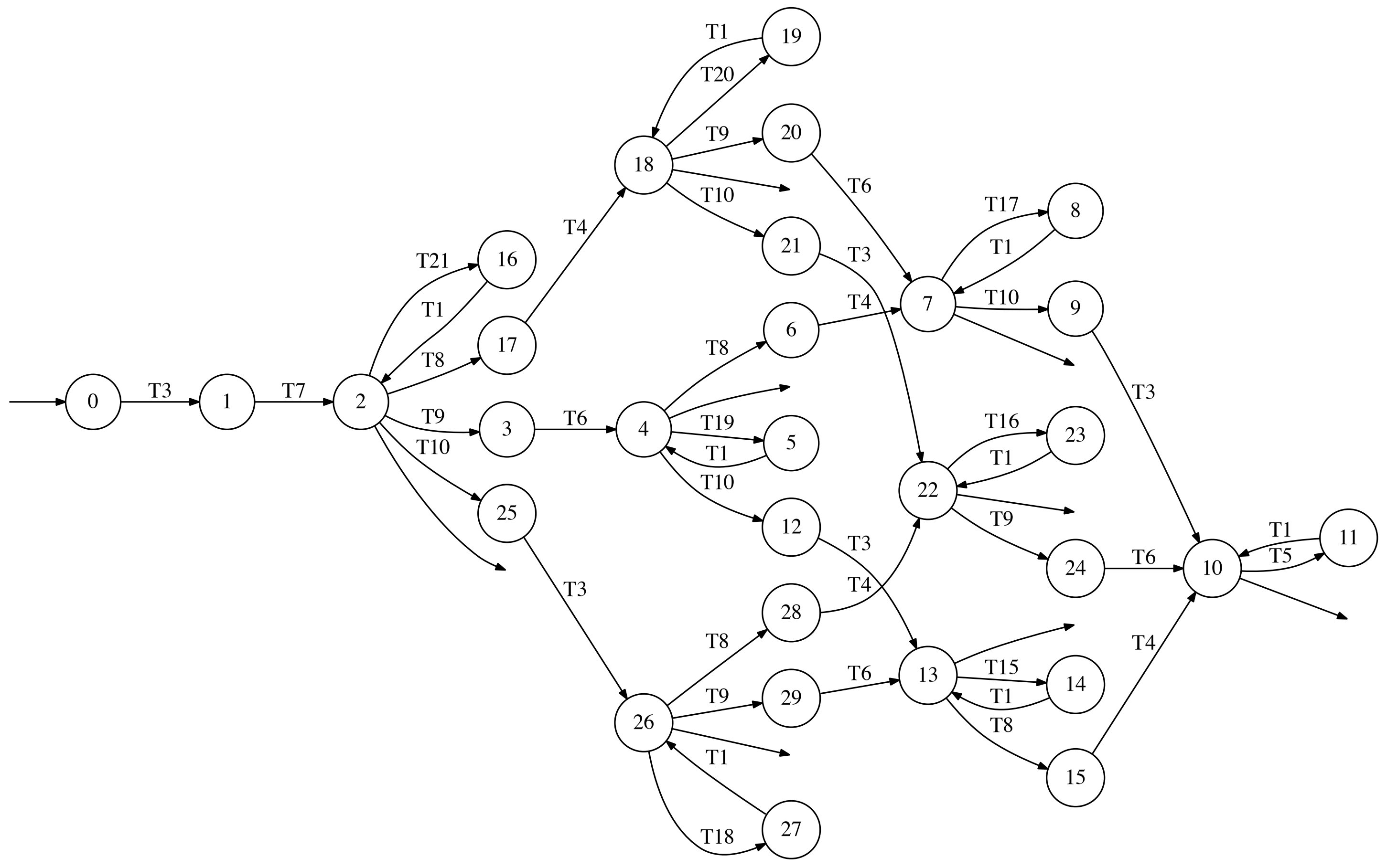


Figure 40: DFA recognizing the destructuring lambda list Case-1

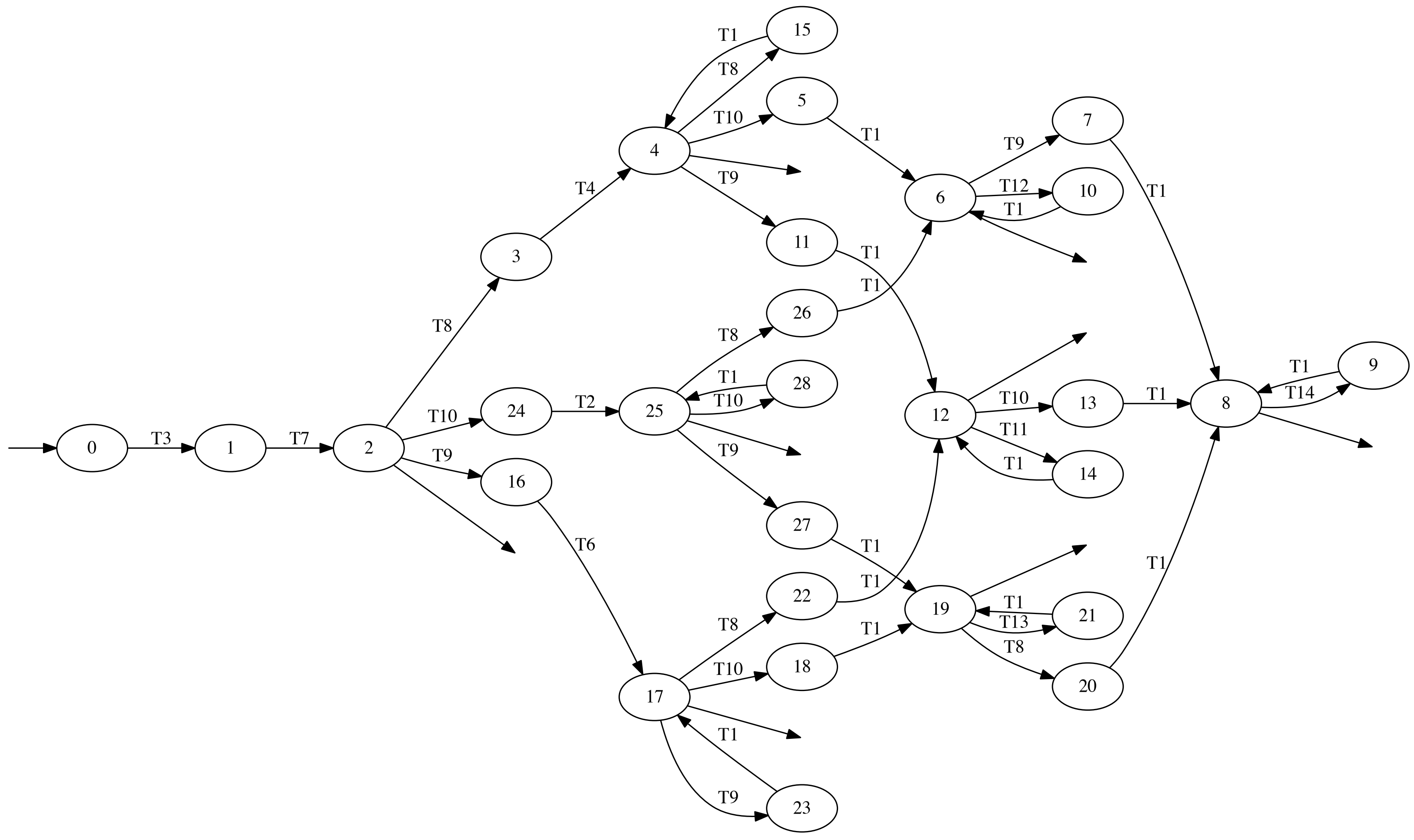


Figure 41: DFA recognizing the destructuring lambda list Case-2

There is a caveat to be aware of when using `destructuring-case`. We do not attempt to solve the problem presented if the actual type of the default value of an optional argument does not match the declared type. We bleieve this problem to be unsolvable in general, because the extreme case is equivalent to the halting problem. However, it could be solved for a wide range of special cases. An attempt at a partial solution might make it more confusing, as the user would be able to easily know if his case is one of those special cases.

```
(destructuring-case '(42)
  ((a &key (count (foo))) ; case-1
   (declare (type number a count))
   ...)
  ((a)   ; case-2
   (declare (type number a))
   ...))
```

Figure 42: Example destructuring-case use case

Figure 42 shows the general unsolvable case. Here the default value for the `count` key variable is the return value of the function `foo`. The issue is that we cannot know whether `foo` will return a value which is of type number as per the declaration. If `foo` returns a number the list `(42)` matches the first destructuring clause, case-1; otherwise `(42)` matches the 2nd destructuring clause, case-2. We cannot know this my examining the given data `(42)`, and we cannot build a state machine (nor any algorithm) which can make this decision without calling the function `foo`, and thus suffering its side effects, even if it turns out to not match.

We could, however implement some very common special cases, but we are not sure doing so would enhance the general usability.

Take the simplest special case for example, the case where no explicit default is specified for the `&key` variables (similar for `&optional` variables). We know that the default value in this case is specified as `nil`, and we know that `nil` is not of type number. Thus in `(42)` does not match case-1.

E.g., Figure 43, if `DATA='(42)`, then case-2 is satisfied. If `DATA='(42 :count 3)` then case-1 is satisfied.

```
(destructuring-case DATA
  ((a &key count) ; case-1
   (declare (type number a count))
   ...)
  ((a)   ; case-2
   (declare (type number a))
   ...))
```

Figure 43: Special case of destructuring-case

Similarly if there *IS* a default given which is a literal (literal string, quoted symbol, number etc) we can figure out (at compile time) whether that literal matches the declared value, in order to detrmine whether `:count` is actually required or optional in the destructured list.

E.g., Figure 44, if `DATA='(42)`, then case-2 is satisfied. If `DATA='(42 :count 3)` then case-1 is satisfied. Case-3 is redundant.

If the default for the `&key` variable is a symbol which is declared a constant, it reduces to the special case in Figure 44. However, it is inclear whether it is possible to know at macro expansion time whether a symbol names a constant.

E.g., Figure 45, if `DATA='(42)`, then case-1 is satisfied. Case-2 is redundant.

For these reasons we don't attempt to implement any of these special cases. One addtional argument is that SBCL doesn't even like the situation in the first place. The example in Figure

```
(destructuring-case '(42)
  ((a &key (count "hello")) ; case-1
   (declare (type number a)
            (type string count))
   ...)
  ((a &key (count 0)) ; case-2
   (declare (type number a count))
   ...)
  ((a)    ; case-3
   (declare (type number a))
   ...))
```

Figure 44: Special case of destructuring-case

```
(defconstant +ZERO+ 0)

(destructuring-case '(42)
  ((a &key (count +ZERO+)) ; case-1
   (declare (type number a count))
   ...)
  ((a)    ; case-2
   (declare (type number a))
   ...))
```

Figure 45: Special case of destructuring-case

46 shows warngins issued at compile time that the default value, nil, does not match the declared type. The same warning appears in the corresponding `destructuring-case` because it expands to `destructuring-bind`.

```
(destructuring-bind (a &key count ) DATA
  (declare (type number count))
  ...)

; in: DESTRUCTURING-BIND (A &KEY COUNT)
;     (IF (NOT (EQL #:G685 0))
;         (CAR (TRULY-THE CONS #:G685)))
; ==>
;   NIL
;
; caught STYLE-WARNING:
;   The binding of COUNT is not a NUMBER:
;     NIL
;   See also:
;     The SBCL Manual, Node "Handling of Types"
```

Figure 46: Warnings from dubious destructuring-bind

8 Implementation details

The function `match-sequence`, can be used to determine whether a given sequence matches a given pattern.

```
(defun match-sequence (input-sequence pattern)
  (declare (type list pattern))
  (when (typep input-sequence 'sequence)
    (let ((sm (or (find-state-machine pattern)
                  (remember-state-machine (make-state-machine pattern)
                                          pattern))))
      (some #'state-final-p
            (perform-transitions sm input-sequence)))))
```

This function takes an input sequence such as a `list` or `vector`, and a regular type expression, and returns `true` or `false` depending on whether the sequence matches the regular type expression. It works as follows:

1. If necessary it builds a finite state machine by calling `make-state-machine`, and caches it to avoid having to rebuild the state machine if the same pattern in used again.
2. Next, it executes the machine according to the input sequence.
3. Finally, it asks whether any of the returned states are final states.

The definition of the `rte` parameterized type is a bit more complicated that we'd like. We'd actually like to define it as follows as a type using `satisfies` of a function which closes over, or even embeds the given pattern, but neither of these is possible. In fact the argument of `satisfies` must be a symbol naming a global function; a function object is not accepted as argument of `satisfies`.

```
;; first INVALID type definition
(deftype rte (pattern)
  `(and sequence
        (satisfies ,(lambda (input-sequence)
                      (match-sequence input-sequence pattern)))))

;; second INVALID type definition
(deftype rte (pattern)
  `(and sequence
        (satisfies `(lambda (input-sequence)
                      (match-sequence input-sequence ,pattern)))))
```

Because of this limitation, the definition of `rte` is a bit more tedious. What the `rte` type definition actually happens is:

1. creates an *intermediate function* which closes over the given pattern
   `(lambda (input-sequence) (match-sequence input-sequence pattern))`
2. creates a function name unique for the given pattern.
3. uses `(setf symbol-function)` to define a function whose function binding is that intermediate function
4. the `deftype` expands to `(and sequence (satisfies` *`that-function-name`*`))`

An example may make it clearer.

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

The type `3-d-point` evokes the `rte` parameterized type definition with argument list `(number number number)`. The `deftype` of `rte` assures that a function is defined as follows. The function name, `|(number number number)|`[7] even if somewhat unusual, is so chosen to improve the error message and back-trace that occurs in some situations.

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

It is also assured that the finite state machine corresponding to `(:cat number number number)` is built and cached, to avoid unnecessary recreation at run-time. Finally the type specifier `(rte number number number)` expands to the following.

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

The following back-trace occurs when trying to evaluate the following failing assertion.

```
(the 3-d-point (list 1 2))

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

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1012A80003}>)

Backtrace:
  0: ((LAMBDA ()))
  1: (SB-INT:SIMPLE-EVAL-IN-LEXENV (THE 3-D-POINT (LIST 1 2)) #<NULL-LEXENV>)
  2: (EVAL (THE 3-D-POINT (LIST 1 2)))
 --more--
```

8.1 Optimized code generation

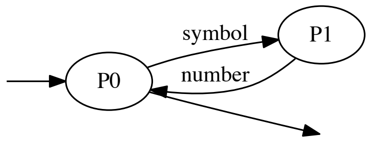


Figure 47: Example DFA for `(:0-* symbol number)`

In section 8 we say a general purpose implementation of a NDFA (non-deterministic finite state machine). There are several techniques which can be used to improve the run-time performance

[7]The `|...|` notation is the Common Lisp reader syntax to denote a symbol containing spaces or other delimiters characters. *E.g.*, `|(a b)|` is a symbol whose print-name is "(a b)".

of this algorithm. First we discuss some of the optimizations we have made, and in section 8.4 there is a discussion of the of the performance results.

One thing to notice is that although the implementation described above is general enough to support non-deterministic state machines, the development made in sections 5 and 6 obviate the need for this flexibility. In fact although each state in a state machine recognizing a rational type expression has multiple transitions to next states, we have assured that maximally one such is ever valid as each transition is labeled with a type disjoint from the other transitions from the same state. The result is that to make a state transition in the DFA case, type membership tests must be made only until one is found which matches, whereas in the NDFA case all type membership tests must be made from each state, and a list of matching next states must be maintained.

Another thing to notice is that rather than traversing the state machine to match a input sequence, we may rather traverse the state machine to produce code which can later match an input sequence. The generated code will be special purpose and will only be able to match a sequence matching the particular regular type expression. There are three obvious advantages of the code generation approach. 1) There will be much less code to execute at run time, that code being specifically generated for the specific pattern we are attempting to match. 2) We can avoid several function calls in the code by making use of `tagbody` and `go`. And 3) the lisp compiler can be given a chance to optimize the more specific (less generic) code.

The result of these two optimizations are that the code no longer makes use of the potentially costly call to `matchsequence`. In its place code is inserted specifically checking the regular type expression in question. Figure 48 shows a sample body of such a function which recognizes the regular type expression `((:0-* symbol number))`. The corresponding rational type expression is $(symbol \cdot number)^*$. The DFA can be seen in Figure 47. The code contains two sections, one for the case that the given sequence, `seq` is a `list` and another if the sequence is a `vector`. The purpose of the two sections is so that the generated code may use more specific accessors to iterate through the sequence, and also so the compiler can have more information about the types of structure being accessed.

Each section differs in how it iterates through the sequence and how it tests for end of sequence, but the format of the two sections is otherwise the same. Each section contains one label for each state in the state machine. Each transition in the DFA is represented as a branch of a `typecase` and `(go ...)` (the Common Lisp GOTO).

8.2 Sticky states

Consider the DFA shown in Figure 49. If the state machine ever reaches state P_2 it will remain their until the input sequence is exhausted, because the only transition is for the type t, and all objects of of this type. This state is called a *sticky state.* If the state machine ever reaches a sticky state which is also a final state, it is no longer necessary to continue examining the input string. The matching function can simply return `true`.

This type of pattern is fairly common such as `(:cat (:0-* symbol number) (:0-* t))`.

We have incorporated this optimization into both the generic DFA version (based on `match-sequence`) and also the auto-generated code version. To understand the consequence of this optimization consider a list of length 1000 which begins with a `symbol` followed by a `number`. With the sticky state optimization, checking the pattern against the sequence would involve:

- one check of `(typep obj symbol)`,
- one check of `(typep obj number)`, and
- 998 checks of `(typep obj t)`, all of which are sure to return `true`.

When this optimization is in effect, 1000 type checks are reduced to 2 type checks.

```
(lambda (seq)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (block check
    (typecase seq
      (list
       (tagbody
         (go P0)
        P0
         (when (null seq) (return-from check t))
         (optimized-typecase
             (if (null seq)
                 (return-from check nil)
                 (pop seq))
           (symbol (go P1)))
         (return-from check nil)
        P1
         (optimized-typecase (if (null seq)
                                 (return-from check nil)
                                 (pop seq))
           (number (go P0)))
         (return-from check nil)))
      (t
       (let ((i 0))
         (tagbody
           (go P0)
          P0
           (when (>= i (length seq)) (return-from check t))
           (optimized-typecase (if (>= i (length seq))
                                   (return-from check nil)
                                   (prog1 (aref seq i) (incf i)))
             (symbol (go P1)))
           (return-from check nil)
          P1
           (optimized-typecase (if (>= i (length seq))
                                   (return-from check nil)
                                   (prog1 (aref seq i) (incf i)))
             (number (go P0)))
           (return-from check nil)))))))
```

Figure 48: Machine generated code for recognizing a rational type expression

8.3 Redundant disjoint typecase

The code generation phase described in section 8.1 generates invocations to the macro `optimized-typecase` which is semantically and syntactically equivalent to the Common Lisp `typecase`. The difference between the two macros is that the former implements some optimizations which the latter does not necessarily implement.

The code generation algorithm within RTE has already assured that the types (in the invocation of `typecase`) are disjoint. Although this is critical for the derivative calculations explained in section 5, it poses a potential performance issue in the generated code, which `optimized-typecase` solves. Consider the following case.

```
(optimized-typecase obj
  (fixnum ...) ; clause 1
  ((and number (not fixnum)) ...) ; clause 2
  (string ...))
```

In order to decide whether clause 2 should be taken, the executing code must check twice whether the object is of type `fixnum`, once in clause 1, and again in clause 2.

The `optimized-typecase` macro expands to code which eliminates this redundancy.

```
(typecase obj
  (fixnum ...) ; clause 1
  (number ...) ; clause 2
  (string ...))
```

In the latter case we know that if clause 2 is ever reached, the object has already been assured to not be of type `fixnum`, there is no need to test it again.

The `optimized-typecase` macro is also available for use independent of the RTE system.

8.4 RTE performance vs hand-written code

A natural question to ask is how does the state-machine approach to pattern matching compare to hand written code. That is to say: what is the cost of the declarative approach?

To help answer this question consider the function `check-hand-written`. It is a straightforward handwritten function to check for a `list` matching the regular type expression `(:0-* symbol number)`.

```
(defun check-hand-written (obj)
  (or (null obj)
      (and (cdr obj)
           (symbolp (car obj))
           (numberp (cadr obj))
           (check-hand-written (cddr obj)))))
```

The test we constructed was to attempt to match 200 samples of lists of length 8K. The handwritten code was able to do this in 0.011351 seconds CPU time. The generic state machine code to do this was 0.879481 seconds, ignoring the initial cost of building the state machine. Using the optimization described in Section 8.1, this time dropped to 0.022239 seconds.

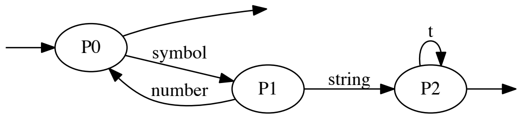


Figure 49: DFA with sticky state

Version	CPU time	Penalty
Hand written	0.011351	1x
Generic DFA	0.879481	77.5x
Generated Code	0.022239	2x

8.5 RTE performance vs CL-PPCRE

The `rte` type can be used to perform simple string regular expression checking. A generally accepted Common Lisp implementation of regular expressions for strings is `CL-PPCRE`[24].

The following example is similar to the one shown in section 7.1.

We would like to count the number of strings in a given list which match a particular regular expression. To analyze the performance we used two approaches: using `CL-PPCRE` and `rte`. In particular, we implemented the following two function `count-matches-ppcre` and `count-matches-rte` respectively.

```
(defvar *data* '("abababababzabab"
                 "ababababzabababab"
                 "ababababzabababab"
                 "ababababzzzzabababab"
                 "ababababababababzzzzzzabababab"
                 "abababababababababababzzzzzzabababab"
                 "abababababababababababababababzzzzzzabababab"
                 "ababababzzzzzzabababababababababzzzzzzabababab"
                 ))

(defvar *test-scanner* (cl-ppcre:create-scanner "^(ab)*z*(ab)*$"))

(defun count-matches-ppcre ()
  (count-if (lambda (str)
              (cl-ppcre:scan *test-scanner* str))
            *data*))

(defun count-matches-rte ()
  (count-if (lambda (str)
              (typep str `(rte ,(regexp-to-rte "(ab)*z*(ab)*"))))
            *data*))
```

The performance difference is significant. A loop executing each function one million times, shows that the `rte` approach runs about 35 % faster.

```
RTE> (time (dotimes (n 1000000) (rte::count-matches-rte)))
Evaluation took:
  6.185 seconds of real time
  6.149091 seconds of total run time (6.089164 user, 0.059927 system)
  99.42% CPU
  14,808,087,438 processor cycles
  32,944 bytes consed

NIL
RTE> (time (dotimes (n 1000000) (rte::count-matches-ppcre)))
Evaluation took:
  8.425 seconds of real time
  8.334411 seconds of total run time (7.750325 user, 0.584086 system)
  98.92% CPU
  20,172,005,693 processor cycles
  768,016,656 bytes consed

NIL
```

8.6 Exceptional situations

In Common Lisp `defclass` creates a class and a type of the same name. Every valid class name is simultaneously a type specifier. The type is the set of instances of that named class, which includes instances of all sub-classes of the class. Classes can be redefined, especially while developing and debugging an application. The implementation described in this article memoizes certain calculations for reuse later. For example, given the rational expression; *i.e.*, the argument list of `rte`, a finite automaton is generated, and cached with the rational expression. The generation of this automaton makes some assumptions about subtype relationships. If classes are redefined later, these relationships may no longer hold; consequently, the memoized automata may no longer be correct.

The Common Lisp Metaobject Protocol [17, 13] provides a mechanism for handling this situation in terms of a dependent maintenance protocol. The protocol allows applications to attach *observers* called "dependents" to classes. Thereafter whenever one of these classes changes an application specific method is called.

We exploit this feature of the CLOS MOP to flush the caches state machines associated with classes as they are redefined redefined.

8.7 Known Open Issue

With the current state of implementation of RTE there is a known serious limitation with respect to the compilation semantics. During each expansion of the **`rte deftype`**, the implementation notices whether this is the first time the given regular type expression has been encounted, and if so, creates a named function to check a sequence against the pattern. This flow is explained earlier in Section 8. After the named function has been created, the `deftype` expands to something like `(and sequence (satisfies rte::|(:* number)|))`, which is what is written to the fasl file being compiled. So the compiler replaces expression such as ``(typep obj `(rte (:* number)))`` with something like `(typep obj '(and sequence (satisfies rte::|(:* number)|)))`. And that's what goes into the fasl file.

The problem occurs the next time you re-start lisp, and load the fasl file. The loader encounters this expression `(typep obj '(and sequence (satisfies rte::|(:* number)|)))` and happily reads it. But when the call to `typep` is encountered at run-time, the function `rte::|(:* number)|` is undefined. It is undefined because the closure which was `setf`'ed to the symbol-function existed in the other lisp, but not in this one.

Apparently, according to [19] is a known limitation in the Common Lisp specification. Certain files which use **rte** based types, can be compiled, but cannot be re-loaded from the compiled file.

If only there were a way to indicate which files should be loaded from source, allowing others to be compiled and loaded from the compiled file. One might think ASDF[4] could be used for this purpose. Unfortunately it probably cannot be. There is no facility in ASDF to mark some files as *load-from-source* and others as *load-from-compiled* [4, Section 16.6.6].

8.7.1 Declaration based solution

In order to force the definition of the *missing* function to be compiled into the fasl file, you may use the declaration macro `defrte`. To use this approach you must declare any regular type expression with `defrte` before it appears within a function definition. Moreover, the text of the regular type expression must be EQUAL to the text declared by `defrte`.

```
(defrte (:* number number))

(defun F (a b)
  (declare (type (rte (:* number number)) a b))
  ...)
```

This usage does indeed seem redundant, but is a pretty easy work-around for this insidious problem.

8.7.2 ASDF based solution

This solution allows the asdf[4] `COMPILE-OP` operation to create an auxiliary file parallel to the fasl file in the compile directory. The file will have a `.rte` extension but the same base file name. Later when the fasl file is loaded via an asdf `LOAD-OP` operation, the `.rte` file will be loaded before the fasl file.

There are several steps you need to follow to exploit this workaround.

1. Include the `:defsystem-depends-on` keword in the `asdf:defsystem`, to register a dependency on `:rte`. You must use `:defsystem-depends-on` rather than simply `depends-on`, otherwise asdf won't be able to understand the use of `:rte-cl-source-file` which follows.

2. In the components section, use `:file` to declare any file which should be compiled and loaded normally, but use `:rte-cl-source-file` to register a file which contains a problematic regular type expression.

Here is an example of such a `defsystem` using `:rte-cl-source-file`.

```
(asdf:defsystem :rte-test
  :defsystem-depends-on (:rte)
  :depends-on (:rte-regexp-test
               :2d-array
               (:version :lisp-unit "0.9.0")
               :2d-array-test
               :ndfa-test
               :lisp-types-test)
  :components
  ((:module "rte"
    :components
    ((:file "test-rte")
     (:file "test-list-of")

     ;; CREATE and LOAD a .rte file
     (:rte-cl-source-file "test-re-pattern")

     (:file "test-destructuring-case-1")
     (:file "test-destructuring-case-2")
     (:file "test-destructuring-case")
     (:file "test-ordinary-lambda-list")))))
```

There are a few subtle points with this implementation. The keyword `:rte-cl-source-file` within the `:components` section of the asdf system definition triggers a custom compilation and loading procedure, governed by the CLOS class `asdf-user:rte-cl-source-file` which inherits directly from `asdf:cl-source-file`. This class `asdf-user:rte-cl-source-file` is defined in the `:rte` package whose loading is triggered by the `:defsystem-depends-on (:rte)` option in the system definition.

There are two methods specializing on the class `asdf-user:rte-cl-source-file`.

```
(defmethod asdf:perform :around ((operation asdf:compile-op)
                                 (file asdf-user::rte-cl-source-file))
  ...)

(defmethod asdf:perform :before ((operation asdf:load-op)
                                 (file asdf-user::rte-cl-source-file))
   ...)
```

The `asdf:perform :around` method intercepts the `asdf:compile-op` operation to determine which **rte** types and which **rte** patterns get defined by compiling the source file via `(call-next-method)`. Once this list is calculated, the `:around` method writes a `.rte` file along side the fasl file whose text defines pattern definition functions. The `:before` method simply loads this `.rte` from source; *i.e.* the `.rte` file is loaded from source before the fasl is loaded. This guarantees that the functions created as a side effect of compilation are also loaded when the fasl is loaded even if the fasl has already been compiled in another lisp image.

9 Alternatives

9.1 Use of cons construct to specify homogeneous lists

One simple and straightforward way to define types representing fixed types of homogeneous lists is illustrated here.

```
(defun list-of-fixnum (data)
  (every #'(lambda (n) (typep n 'fixnum))
         data))

(deftype list-of-fixnum ()
  `(and list (satisfies list-of-fixnum)))
```

Using this approach the developer could define several types for the various types of lists he needs to declare in his program.

The `cons` type construct can be used to declare the types of the `car` and `cdr` of a cons cell, *e.g.*, `(cons number (cons integer (cons string null)))`. The `cons` construct may be used any finite number of times explicitly, *e.g.* to declare a list of exactly three numbers, you may use: `(cons number (cons number (cons number null)))`

The syntax of the `cons` construct can be tedious when attempting to specify a list with length more than 2 or three. For example, to specify a list of 4 numbers, you would use `(cons number (cons number (cons number (cons number null))))`. It is easy to define an intermediate type to simplify the syntax.

```
(deftype cons* (&rest types)
  (cond
    ((cddr types)
     `(cons ,(car types)
            (cons* ,@(cdr types))))
    (t
     `(cons ,@types))))
```

Using the newly defined `cons*` type we can specify a list of 4 numbers as `(cons* number number number number null)`.

One might ask whether the RTE implementation might benefit by recognizing lists of fixed length and simply expanding to a Common Lisp type specifier using `cons`. We did indeed consider this during the development, but found it caused a performance penalty. Admittedly, we only investigated this potential optimization with SBCL, but experimentation showed roughly 5% penalty for lists of length 5. Moreover, the penalty seems to grow for longer lists: 25% with a list length of 10, 40% with a list length of 20.

Another disadvantage of the approach of using the `cons` specifier is that it is not possible to combine the two approaches above to generalize homogeneous list types of arbitrary length. One might attempt in vain to define a type for homogeneous lists recursively as follows in order to specify a type such as `(list-of number)`.

```
(deftype list-of (type)
  `(or null (cons ,type (list-of ,type))))
```

But this self-referential type definition is not valid,[14] because of the Common Lisp specification of `deftype` which 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.

An attempt to use such an invalid type definition will result in something like the following:

```
CL-USER> (typep (list 1 2 3) '(list-of fixnum))
INFO: Control stack guard page unprotected
Control stack guard page temporarily disabled: proceed with caution

debugger invoked on a SB-KERNEL::CONTROL-STACK-EXHAUSTED in thread
#<THREAD "main thread" RUNNING {1002AEC673}>:
  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.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.

(SB-KERNEL::CONTROL-STACK-EXHAUSTED-ERROR)
0]
```

9.2 Alternative implementation of `destructuring-case`

There is alternative approach to implementing the `destructuring-case` as discussed in Section 7.4. The approach would be to expand the macro invocation into a `cond` whose test attempts to destructure the lambda list to the argument in question, but does so within a `ignore-errors`, test the declared types, then execute the clause of the cond which does not trigger any errors and matches the declared. The second return value of `ignore-errors` is `nil` if no condition was raised.

We don't present such an alternate implementation, but rather suggest by example, what such an implementation might do. Figure 50 shows a usage of `destructuring-case`. Figure 51 shows how the macro might expand.

```
(destructuring-case (list 1 2)
  ((a (b c) & key (d (incf *X*)))
   (declare (type fixnum a)
            (type symbol b))
   (F3 a b c))
  ((a b & key (d (incf *Y*)))
   (declare (type fixnum a b))
   (F2 a b))
  ((a)
   (F1 a)))
```

Figure 50: Example usage of `destructing-case`

There are shown in 51 which must be resolved in such a macro definition.

1. Failed attempt to destructure a list which does not match the destructuring-lambda-list must catch conditions to be raised.

2. Supress any `UNUSED VARIABLE` warning issued by the compiler within the test; *i.e.*, generate `(declare (ignorable ...))` where necessary. However, don't supress warnings which are a result of the user's code.

3. Convert the type declarations into an explicit type check.

```
(let ((#:G1 (list 1 2)))
  (cond ((multiple-value-bind (#:G2 #:G3)
               (ignore-errors (destructuring-bind (a (b c) &key d) #:G1
                                (declare (ignore c d))
                                (and (typep a 'fixnum)
                                     (typep b 'symbol))))
             (and #:G2
                  (null #:G3)))
           (destructuring-bind (a (b c) &key (d (incf *X*))) #:G1
             (declare (type fixnum a)
                      (type symbol b))
             (F3 a b c)))
          ((multiple-value-bind (#:G2 #:G3)
               (ignore-errors (destructuring-bind (a b &key d) #:G1
                                (declare (ignore d))
                                (and (typep a 'fixnum)
                                     (typep b 'fixnum))))
             (and #:G2
                  (null #:G3)))
           (destructuring-bind (a b &key (d (incf *Y*))) #:G1
             (declare (type fixnum a b))
             (F2 a b)))))
          ((multiple-value-bind (#:G2 #:G3)
               (ignore-errors (destructuring-bind (a) #:G1
                                (declare (ignore a))
                                t))
             (and #:G2
                  (null #:G3)))
           (destructuring-bind (a) #:G1
             (F2 a b)))))
```

Figure 51: Example expansion of `destructing-case`

4. Assure that side effects are only executed in the `cond` clause which is taken, and that the side effects are executed the correct number of times. *I.e.*, filter side effects out of the destructuring lambda list used for the branch check.

5. In case no type declarations are found in a clause, assure that `ignore-errors` still returns `true` as first argument.

10 Extension to other languages

It is not at all clear how to extend this concept to other dynamic languages, especially those which are very different from Common Lisp. We would like to investigate whether the Python language, for example, has sufficient reflective capability to implement something like a type specifier, a type calculus, and the subtypep function.

10.1 Implementation in Python

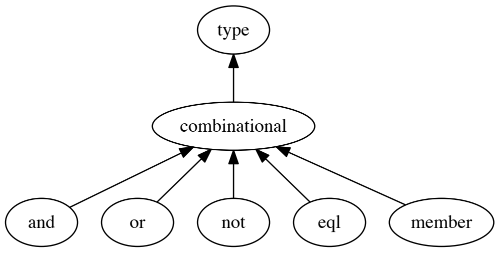


Figure 52: Proposed Python Class Hierarchy

We have begun a preliminary investigation/discussion about implementing this system in the Python programming language. It indeed appears Python has the necessary primatives, although our claim is not conclusive. It appears that the type called “typep” is sub-classable. The idea is to create a subtype of “type” called “combinational-type”, and subtypes thereof called “or”, “and”, “not”, “satisfies”, “eql”, and “member”. See Figure 52 Of course we’ll have to choose names which are valid in the Python language.

There are methods corresponding to the Common Lisp `typep` and `subtypep` which we can implement on the `combinational-type` class.

11 Discovered limitations

During the work on this project there were several notable issues we discovered.

11.1 Missing type specifier checker

There does not seem to be a way to detect whether an object is a valid type specifier. In fact there is some doubt as to exactly what it means to be a *valid type specifier*, whether it is a question of syntax, or whether it is a question of existance. *E.g.* does a value returned from `gensym` return a valid specifier of a not-yet-defined type?

Here is an attempt used in the **rte** implementation, but it does not work for all implementations. There are several discussions of this on `comp.lang.lisp`.

```
(defun valid-type-p (type-designator)
  #+sbcl (and (SB-EXT:VALID-TYPE-SPECIFIER-P type-designator)
              (not (eq type-designator 'cl:*)))
  #+(or clisp allegro)
      (ignore-errors (subtypep type-designator t))
  #-(or sbcl clisp allegro)
      (error "VALID-TYEP-P not implemented for ~A"
             (lisp-implementation-type))
)
```

11.2 SBCL subtypep issues with SATISFIES

`https://bugs.launchpad.net/sbcl/+bug/1528837`

There are several troublesome issues with the `subtypep` function in SBCL. First, if I define types using `satisfies` SBCL thinks it knows subclass information that it cannot know. Here is an example using two functions, `F` and `G`, which are explicitly not yet defined.

```
CL-USER> (deftype even ()
           `(and integer
                 (or (eql 0) (satisfies F))))
EVEN
CL-USER> (deftype odd ()
           `(and integer
                 (or (eql 1) (satisfies G))))
ODD
CL-USER> RTE> (subtypep 'odd 'even)
NIL
T
CL-USER> (subtypep 'even 'odd)
NIL
T
CL-USER>
```

The `subtypep` function returns `NIL,T` indicating that it is sure neither `odd` nor `even` is a subtype of the other. But it cannot know that without functions `F` and `G` being defined. Furthermore, even in the case the functions *are* defined, `subtypep` still returns the wrong value. Consider the case when `F` and `G` are the same. In this case the types are actually both a subtype of the other.

```
CL-USER> (defun F (x) t)
F
CL-USER> (defun G (x) t)
G
CL-USER> (deftype even ()
           `(and integer
                 (or (eql 0) (satisfies F))))
EVEN
CL-USER> (deftype odd ()
           `(and integer
                 (or (eql 1) (satisfies G))))
ODD
CL-USER> RTE> (subtypep 'odd 'even)
NIL
T
CL-USER> (subtypep 'even 'odd)
NIL
T
CL-USER>
```

The return value shoudl be `NIL,NIL` rather than `NIL,T` in this case.

11.3 SBCL subtypep issues with keywords

https://bugs.launchpad.net/sbcl/+bug/1533685

The SBCL implementation of subtypep is confused with regard to keyword symbols and the eql specializer.

```
CL-USER> (subtypep '(eql :x) 'keyword)
NIL
NIL
```

While this result is indeed conforming. The subtypep function is allowed by the specification to return NIL, NIL in any case involving eql. However, it is clear that the singleton set containing the :x keyword symbol is a subset of the set of all keywords. It would seem to be a better choice for subtypep to return T,T in this case.

A similar problem exists with member types such as (subtypep '(member :x) keyword) and (subtypep '(member :x :y) keyword) which should also return T,T.

11.4 SBCL subtypep issues with compiled-function

https://bugs.launchpad.net/sbcl/+bug/1537003

The SBCL implementation of subtypep does not know that there exists at least one object of type compiled-function.

```
CL-USER> (subtypep 'compiled-function nil)
NIL
NIL
```

However it should return NIL,T.

The Common Lisp specification for subtypep states: http://clhs.lisp.se/Body/f_subtpp.htm *subtypep never returns a second value of nil when both type-1 and type-2 involve only the names in Figure 4-2, or names of types defined by defstruct, define-condition, or defclass, or derived types that expand into only those names. While type specifiers listed in Figure 4-2 and names of defclass and defstruct can in some cases be implemented as derived types, subtypep regards them as primitive.*

Please notice that both NIL and COMPILED-FUNCTION both appear in Figure 4-2.

11.5 SBCL performance related issue with subtypep

Apparently it is a *known* issue in SBCL that subtypep has *performance issues* [2].

The issue seems to be that the subtypep function allocates lots of memory. In fact, one test shows that 781,353 calls to the function allocates 2,247,898,784 bytes of consumes 12.864 seconds.

The problem appears to be a bottleneck for the performance of the type segmentation algorithms explained in Section 6.

11.6 Closures and lambda expressions within SATISFIES types

In a type specifier the argument of SATISFIES must be a symbol naming a functions. The argument is not allowed to be a function object, nor a lambda expression such as (lambda (x) (foo x)). This limitation eventually leads to the problem explained in Section 11.7.

In order to implement a type such as greater, we would like to write the following, but is it not correct.

```
(deftype greaterp (n)
  `(and real
        (satisfies (lambda (m) (> m ,n)))))
```

What one must do instead is something like the following.

```
(defun generate-greater-function (n)
  (let ((name (gensym)))
     (fset (symbol-function name)
           (compile nil `(lambda (m) (> m ,n))))
     name))

(deftype greaterp (n)
  `(and real
        (satisfies ,(generate-greater-function n))))
```

This approach has several issues.

1. If the `greater` tyep is used in a function declaration, then a function is generated at compile time but not included in the fasl file. See Section 11.7.
2. If a call such as `(typep object '(greater 42))` occurs at run-time expecially something like `(typep object T1)` where `T1` has value `(greater 42)`, then the compiler will be called during run-time.
3. A call such as `(the (greater 42) 12)` will produce an incomprehensible error message which is likely to NOT contain the type name `(greater 42)`.

11.7 Side effects during compilation

Section 8.6 explains some workarounds for this problem. The problem basically is that during compilations it might be the case that certain functions are dynamically created, such as during type expansion.

11.8 Recursive types

11.9 Extensible subtype mechanism

The Common Lisp programmer is allowed to define new types using `deftype` and `satisfies`, but is not allowed to extend the capability of `subtypep`. It would seem natural for the programmer to wish to be able to extend `subtypep` to declare that `(satisfies oddp)` is a subtype of *integer* and in particular a non-empty subtype.

11.10 Type reflection

There is no standard way for an application to perform a type expansion. Neither is there a way to even ask whether there is a type definition (`deftype`) for a suspected type name. Furthermore, when a type definition changes there is no way for an application to be notified. Constrast this with the dependency mapping protocol described in Section 8.6. The lisp system will notify an application when the class lattice has changed because a class has been redefined, because the application may have cached information based on the class hierarchy. Such a protocol is missing for non-class types as defined by `deftype`.

12 Conclusions and Future work

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, out 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.

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'd 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 needs to have to support such implementation.

Several open questions remain:

Can regular type expressions can be extended to implement more things we'd expect from a regular expression library. For example, can we have grouping remember what was matched, and use that for regexp-search-and-replace? Additionally, would such a search and replace capability be useful?

Can this theory be extended to tackle unification. Can RTE be extended to implement unification in a way which adds value?

One problem in general with regular expressions is that if you use them to find *whether* a string (sequence in our case) does or does not match a pattern. We would often like to know `why` it fails to match. Questions such as "How far did it match?" or "Where did it fail to match?" would be nice to answer. It is currently unclear whether the RTE implement can at all be extended to support these features.

References

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

[2] Performance of subtypep, thread on sbcl-devel mailing list, 2016.

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

[4] Barlow, D. Asdf user manual for version 3.1.6, 2015.

[5] Barnes, T. SKILL: a CAD system extension language. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE* (Jun 1990), pp. 266–271.

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

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

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

[9] Duret-Lutz, A. Conversations concerning segmentation of sets, 2015.

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

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

[12] Katzman, D. Thread on SBCL Devel-list sbcl-devel@lists.sourceforge.net, 2015.

[13] Kiczales, G. J., des Rivières, J., and Bobrow, D. G. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[14] Margolin, B. declaring the elements of a list. Thread on comp.lang.lisp, December 2015.

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

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

[17] Paepcke, A. User-level language crafting – introducing the Clos metaobject protocol. In *Object-Oriented Programming: The CLOS Perspective*, A. Paepcke, Ed. MIT Press, 1993, ch. 3, pp. 65–99. Downloadable version at http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps.

[18] Pin, J.-E. Mathematical foundations of automata theory.

[19] Pitman, K. M. Using closures with satisfies, comp.lang.lisp, 2003.

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

[21] Riesbeck, C. Lisp unit. https://www.cs.northwestern.edu/academics/courses/325/readings/lisp-unit.html.

[22] Senta, L., Chedeau, C., and Verna, D. Generic image processing with Climb. In *European Lisp Symposium* (Zadar, Croatia, May 2012).

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

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

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

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