Recognizing heterogeneous sequences by rational type expression

Jim E. Newton
Didier Verna
jnewton@lrde.epita.fr
didier@lrde.epita.fr
EPITA/LRDE
Le Kremlin-Bicêtre, France

ABSTRACT

We summarize a technique for writing functions which recognize types of heterogeneous sequences in Common Lisp. The technique employs sequence recognition functions, generated at compile time, and evaluated at run-time. The technique we demonstrate extends the Common Lisp type system, exploiting the theory of rational languages, Binary Decision Diagrams, and the Turing complete macro facility of Common Lisp. The resulting system uses meta-programming to move an $\Omega(2^n)$ complexity operation from run-time to a compile-time operation, leaving a highly optimized $\Theta(n)$ complexity operation for run-time.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; *Type theory*; • **Computing methodologies → Representation of Boolean functions**;

ACM Reference Format:
Jim E. Newton and Didier Verna. 2018. Recognizing heterogeneous sequences by rational type expression. In *Proceedings of Workshop on Meta-Programming Techniques and Reflection (META'18).* ACM, New York, NY, USA, 9 pages.

1 INTRODUCTION

In this article, we present code generation techniques related to run-time type checking of heterogeneous sequences. Traditional regular expressions [HMU06, YD14] can be used to recognize well defined sets of character strings, called *rational languages* or sometimes *regular languages*. Newton *et al.* [NDV16] present an extension whereby a dynamic language may recognize a well defined set of heterogeneous sequences, such as lists, vectors, and user defined sequence types [Rho09]. Even though our work applies equally well to other sequence types, for the rest of the paper we will refer mainly to lists.

As with the analogous string matching regular expression theory, matching these *regular type expressions* can also be achieved by using a finite state machine (DFA, deterministic finite automata) [HMU06]. Constructing such a DFA can be time consuming, typically exponential in complexity [Hro02]. Without the use of meta-programming, which runs at compile-time outputting targeted functions for use at run-time, the program may be forced to construct the DFA at run-time. The excessively high cost of such a construction may far outweigh the time needed to match a string against the expression.

Our technique involves hooking into the Common Lisp type system via the `deftype` macro. The first time the compiler encounters a relevant type specifier, the appropriate DFA is created, which may be a $\Omega(2^n)$ operation (complexity bounded below by an exponential), from which specific low-level code is generated to match *that* specific expression. Thereafter, when the type specifier is encountered again, the same pregenerated function can be used. The code generated is $\Theta(n)$ complexity (bounded above and below by a linear function) at run-time. We refer the reader to Wegener [Weg87, Section 1.5] for a discussion of Ω and Θ notation.

A complication of this approach, which we explain in the paper, is that to build the DFA we must calculate a maximal disjoint type decomposition (MDTD, Section 4.2) which is both time consuming, and also leads to sub-optimal uses of run-time type predicates. (See Section 2 for an explanation of the Common Lisp `typecase` macro.) To handle this we use our own macro `optimized-typecase` in our machine generated code. Uses of that macro are also implicitly expanded at compile time. Our macro expansion uses BDDs (Binary Decision Diagrams) [Bry86, Bry92, Ake78, Col13, And99, Knu09] to optimize the `optimized-typecase` into low level code, maintaining the `typecase` semantics but eliminating redundant type checks.

2 BACKGROUND

Common Lisp is a programming language defined by its specification [Ans94] and with several implementations, both open source and commercial. For this research, we have used SBCL [New15] as implementation of choice. All implementations share the common specified core of functionality, and each implementation extends that functionality in various ways.

Two Common Lisp features which we exploit are its macro system and its type system. The following is a very high level summary of Common Lisp types and macros.

The Common Lisp macro system [Gra96, Section 10.2] allows programmers to write code which writes code. A macro may be define using `defmacro` and such macros are normally *expanded* at compile time into code which is compiled and this is available for execution at run time. Thanks to the homoiconicity [McI60, Kay69] of the Common Lisp language, macros take arguments which are Lisp objects (symbols, strings, numbers, lists etc), and return complete or incomplete program fragments which are also Lisp objects. This contrasts with the Scala macro system introduced by Burmako [Bur13]. Scala macros programmatically map abstract syntax trees to valid abstract syntax trees. Programmers writing

META'18, November 4-9 2018, Boston MA, USA
2018.

Common Lisp macros may use any feature of the language in the macros themselves, in particular, the input to a Common Lisp macro need not be valid Common Lisp code, but may as well be homoiconic data which the compiler might otherwise reject, but which the macro treats as DSL. We exploit this feature by arranging that the input to our macro be a DSL denoting a regular type expression the syntax of which we describe in Section 3. There are several, well understood, caveats associated with Common Lisp macros. Costanza [CD10] explores the most notably of these hygienic issues. Expert Common Lisp programmers understand these restrictions and normally write macros exploiting the package system and the `gensym` function to avoid name conflicts.

The Common Lisp type system [Ans94, Section 4] can be understood by thinking of types as sets of objects. Subtypes correspond to subsets. Supertypes correspond to supersets. The empty type, called `nil`, corresponds to the empty set. We sometimes refer to the empty type as $\perp$ as it appears at the bottom of the type lattice. The system comes with certain predefined types, such as `number`, `fixnum` (an integer which is not a `bignum`), `ratio` (the exact quotient of two integers) and many more. Additionally, programmers may compose *type specifiers* which are syntax for expressions types in terms of other types. These types are intended to be both human and machine readable. For example, `(or number string)` expresses the union of the set of numbers with the set of strings. Likewise, type specifiers may use the operators `and`, `not`, `member`, and `satisfies` respectively for conjunction, complementation, enumeration, and definition by predicate.

Common Lisp allows types to be used in variable and function declarations, slot definitions within objects, and element definitions within arrays. These declarations are normally considered during program compilation. In addition, Common Lisp provides several other built-in macros and functions for type-based run-time reflection: `typep`, `typecase`, `subtypep`, `check-type`. The programmer may associate new type names with composed types by using `deftype`, whose syntax is similar to that of `defmacro`.

The Common Lisp `typecase` macro [Ans94, Section TYPECASE] allows the conditional execution of a body of forms in a clause that is selected by matching the value of a given expression on the basis of its type. Pierce [Pie02, p. 341] explains that the addition of a `typecase`-like facility (which he calls `typecase`) to a typed λ-calculus permits arbitrary run-time pattern matching. A simple usage example of Common Lisp `typecase` should suffice to give an impression of how it works.

```
(typecase object
  (string "it's a string")
  (array  "it's an array")
  ((not real) "it's not a real number")
  ((or ratio bignum) "it's either a ratio or bignum")
  (t "not any of the above"))
```

The type of the value of `object` is tested at run-time. That value might be of one or more of the types specified. The first such matching clause is activated, and the code within the clause is executed. Since every Lisp object is of type *true*, the final clause in this example serves as the default clause, executing if no preceding clause matches.

`:*`	match zero or more times.
`:+`	match one or more times.
`:?`	match zero or one time.
`:cat`	concatenation operator.
`:or`	disjunction operator.
`:and`	conjunction operator.

Figure 1: Regular type expression keywords

3 HETEROGENEOUS LISTS

A *rational type expression* [NDV16] abstractly denotes a pattern of types within lists. The concept is envisioned to be intuitive to the programmer in that it is analogous to patterns described by regular expressions, which we assume the reader is already familiar with.

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

A rational type expression is expressed in mathematical notation using superscripts and infix operators. A more conventional and machine friendly, s-expression-based syntax, called *regular type expression* is used to encode a *rational type expression* into ASCII characters amenable to the Common Lisp compiler. This syntax replaces the infix and post-fix operators in the rational type expression with prefix notation based s-expressions. The regular type expression `(:cat string (:* number) symbol)` corresponds to the rational type expression $(string \cdot number^* \cdot symbol)$.

We have implemented a parameterized type named **rte** (regular type expression), via `deftype`. The call-by-name argument of **rte** is a *regular type expression*. The members of such a type are all lists matching the given regular type expression.

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

Rational language syntax usually have operators for concatenation, conjunction, disjunction, and repetition. Regular type expressions follow this model using the operators, `:cat`, `:and`, `:or`, and `:*`, as shown in Table 1. In addition rational language syntax often has short-cut syntax for matching one or more times, or matching zero or one time. In keeping with this tradition, regular type expressions employ the operators `:+` and `:?`. Noticeably missing from Table 1 is a complementation operation. We consider a `:not` operator as an important enhancement ad leave it for future research.

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

`(rte (:cat number number number))`
corresponds to the rational type expression $(number \cdot$

```
(defclass C ()
  ((point :type (rte (:cat number number)))
   ...))

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

Figure 2: Example uses of type declarations

number · *number*) and matches a list of exactly three numbers.

`(rte (:or number (:cat number number number)))`
corresponds to (*number* + (*number* · *number* · *number*)) and matches a list of either one or three numbers.

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

`(rte (:* (:cat cons number)))`
corresponds to (*cons* · *number*)* and matches a list of a `cons` followed by a `number` repeated zero or more times, *i.e.*, a list of even length.

The **rte** type can be used anywhere Common Lisp expects a type specifier as the following examples illustrate. The `point` slot of the class `C`, in Figure 2 expects a list of two numbers, *e.g.*, `(1 2.0)`. Also in Figure 2, the Common Lisp type specified by `(cons number)` is the set of non-empty lists headed by a number, as specified in [Ans94, System Class CONS]. The `Y` argument of the function `F` must be a possibly empty list of such objects, because it is declared as type `(rte (:* (cons number)))`. *E.g.*, `((1.0) (2 :x) (0 :y "zero"))`.

4 GENERATING A STATE MACHINE

A standard technique [HMU06, Chapters 3,4] used in regular expression matching is to convert the regular expression into a DFA (deterministic finite automaton, sometimes called a *finite state machine*). Thereafter, there are two general approaches to deciding whether a given list matches. If the DFA is available at run-time, it can be used along with a generic matching function which takes the DFA and candidate list as input parameters. The function would then iterate through the list associating a state with each iteration. Finally, when the list is exhausted, if the DFA is in a *final* state, return true. We initially experimented with this approach, but quickly abandoned it for performance reasons. We elected to pursue an alternate approach based on meta programming.

This alternate approach, which we use, is to build the DFA at compile time triggered by the compiler's treatment of **rte** type declarations. Several steps are involved.

(1) Convert a parameterized **rte** type specifier into code that will perform run-time type checking.
(2) Convert the regular type expression to DFA.
(3) Convert the DFA into code which will perform run-time execution of the DFA.

The DFA is a compile-time-only object.

4.1 Type definition

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

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

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

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

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

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

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

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

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

Finally, the type specifier `(rte (:cat number number number))` expands to the following.

```
(and list
     (satisfies |(:cat number number number)|))
```

4.2 Constructing a DFA

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

In 1964, Janusz Brzozowski [Brz64] introduced the concept of the *Rational Language Derivative*, and provided a theory for converting

a regular expression to a DFA. Owens *et al.* [ORT09] makes a fresh presentation of the algorithm in easy to follow steps. We found the explanation of Owens amenable to our needs. We omit the details of that algorithm at this time, except to say that Brzozowski argues that the algorithm terminates because repeatedly calculating this derivative results in a finite number of calculated expressions, each of which correspond to a state in the state machine. Additionally, each derivative is calculated *with respect to* some term. Each such term is a type specifier in our case, and each corresponds to a transition between two states in the state machine. We refer the reader to Newton *et al.* [NDV16] for specific details.

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

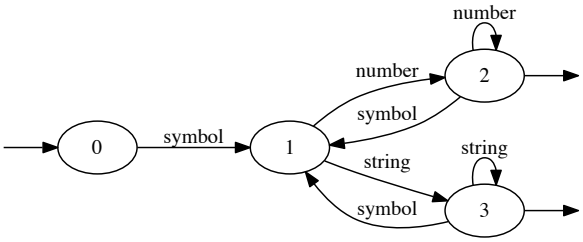

Figure 3: Example DFA

Consider the extended rational type expression:

$$(symbol \cdot (number^{+} + string^{+}))^{+} .$$

We can use the Brzozowski algorithm to construct a DFA which recognizes lists matching this pattern. Such a DFA is illustrated in Figure 3.

There is a delicate but important matter when the mapping of list-of-objects to list-of-type-specifiers: the mapping is not unique. The issue is that the Brzozowski algorithm assumes the creation of a *deterministic* state machine. If intersecting types are encountered, such as `number` and `integer`, the Brzozowski algorithm, applied naively, produces an invalid result. The solution is to decompose the user specified types into a set of disjoint types whose union is the same as the original set. Newton [NVC17] examines algorithms for calculating the *maximal disjoint type decomposition* (MDTD).

A *disjoint type decomposition* of a given set of types is a set of disjoint types whose union is the same as the union of the given set of types. If the given set of types is finite (*i.e.* finite number of types, not necessarily types denoting finite sets of values), a disjoint type decomposition exists. Every disjoint type decomposition is finite, and there is exactly one whose size is larger than any other. Thus there exists a `maximal disjoint type decomposition`. The proof is beyond the scope of this article, but may be found in [New17].

As an example of a rational type expression involving intersecting types, consider the follow,

$$((number \cdot integer) + (integer \cdot number)),$$

whose **rte** is `(:or (:cat number integer) (:cat integer number))`. The corresponding DFA is shown in Figure 4. Notice that the transition between state 0 and 1 is governed by the type specifier `(and number (not integer))` even though this type is not explicitly mentioned in the **rte**.

4.3 Optimized code generation

The mechanism we chose for implementing the execution (as opposed to the generation) of the DFA was to generate specialized code based on `optimized-typecase`, `tagbody`, and `go`. As an example, consider the DFA shown in Figure 3. The code in Figure 5 was generated given this DFA as input. The `tagbody` and `go` special operators are built-in to Common Lisp; however `optimized-typecase` is a macro whose definition is explained in Section 6. For the moment, it is only important to understand its semantics as being exactly those of the `typecase` built in to Common Lisp.

While the function is iterating through the list, if it encounters an unexpected end of list, or an unexpected type, the function returns *false*. These clauses are commented as `rejecting`. Otherwise, the function will eventually encounter the end of the list and return *true*. These clauses are commented `accepting` in the figure.

There is one label (`L0`, `L1`, `L2` ...) per state in the state machine, corresponding to the states, 0, 1, 2 ... in the DFA in Figure 3. At each step of the iteration, a check is made for end-of-list. Depending on the state either *true* or *false* is returned depending on whether that state is a final state of the DFA or not.

The next element of the list is examined by a variant of `typecase`, called `optimized-typecase`, and depending of the type of the object encountered, control is transferred (via `go`) to a label corresponding to the next state.

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

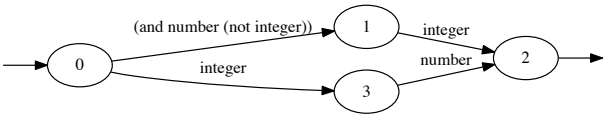

Figure 4: Example DFA with disjoint types. This DFA is derived from the rational type expression $((number \cdot integer) + (integer \cdot number))$**.**

```
(lambda (list)
  (declare
    (optimize (speed 3) (debug 0) (safety 0)))
  (tagbody
   L0
     (when (null list)
       (return nil)) ; rejecting
     (optimized-typecase (pop list)
       (symbol (go L1))
       (t (return nil)))
   L1
     (when (null list)
       (return nil)) ; rejecting
     (optimized-typecase (pop list)
       (number (go L2))
       (string (go L3))
       (t (return nil)))
   L2
     (when (null list)
       (return t)) ; accepting
     (optimized-typecase (pop list)
       (number (go L2))
       (symbol (go L1))
       (t (return nil)))
   L3
     (when (null list)
       (return t)) ; accepting
     (optimized-typecase (pop list)
       (string (go L3))
       (symbol (go L1))
       (t (return nil)))))
```

Figure 5: Generated code recognizing an RTE

5 REPRESENTING TYPES AS ROBDD

Before looking at the macro expansion of `optimized-typecase` in Section 6, we first look at a binary decision diagram representation of Common Lisp types.

In Common Lisp, types are sets. In keeping with this, intersecting types correspond to intersecting sets; likewise for union and complements of sets. The notation is intuitive. *E.g.*, the following s-expression

```
(or (not number)
    (eql 42)
    (and fixnum (not unsigned))
    (and unsigned (not fixnum)))
```

is a type specifier representing the set-union of the four sets:

- $\overline{number}$
- $\{42\}$
- $fixnum \setminus unsigned$
- $unsigned \setminus fixnum$

Newton *et al.* [NVC17] explain how a Common Lisp type is represented as a *reduced ordered binary decision diagram* (ROBDD). Such an ROBDD representing the type shown above is illustrated in Figure 6.

An ROBDD is a special type of BDD (Binary Decision Diagram) which has been subjected to an *ordering* and a *reduction*. That it is ordered indicates that every path which starts at the top node (labeled `fixnum` Figure 6), follows the arrow direction, and after any number of steps arrives at one of the bottom node (labeled $\bot$ and $\top$ Figure 6) is guaranteed to encounter the type names in the same order; possibly with some type names omitted, but never two different orders in two different paths. The particular order does matter for the semantics of the diagram but may effect the resulting size of the diagram, or may effect the efficiency of the resulting code. An important concern in pattern matching compilation is that finding the best ordering is known to be coNP-Complete [Bry86]. However, when using BDDs to represent Common Lisp type specifiers, we enforce a consistent ordering; finding the *best* ordering is not necessary for our application.

Andersen [And99] and Grοpl [GPS98] describe the reduction steps that are necessary to implement an ROBDD, but in short, the reduction steps assure that there are no redundant subtrees in the diagram.

The ROBDD represents a program to test the value of a Boolean function of some number of variables. Commonly, the Boolean expression and the corresponding ROBDD may represent logic in an electronic circuit [BRB90, Lee59], or a constraint in a model checker [BCM+92, ST98, Col13]. Since Common Lisp types are most generally expressed in terms of Boolean combinations of types as shown above, we can use an ROBDD to represent the execution of the type predicate function. The function evaluates to *true* if the object in question is of the specified type, *false* otherwise. To use the ROBDD to evaluate such a type check, we start at the top node, *number*, and ask whether the object is of type *number*. If the answer is *yes*, we follow the green solid arrow to the next node, otherwise we follow the red dotted arrow to the next node, and in either case, repeat the process until we arrive at the bottom of the

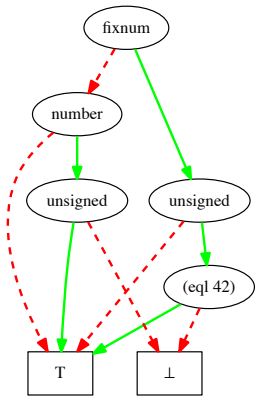

Figure 6: Common Lisp type, `(or (not number) (eql 42) (and fixnum (not unsigned)) (and unsigned (not fixnum)))` represented as ROBDD. The symbols $\bot$ and $\top$ refer to *false* and *true* respectively.

```
(bdd-typecase obj
  ((and unsigned (not (eql 42)))
   body-forms-1...)
  ((eql 42)
   body-forms-2...)
  ((and number (not (eql 42)) (not fixnum))
   body-forms-3...)
  (fixnum
   body-forms-4...))
```

Figure 7: Invocation of `bdd-typecase` with intersecting types

diagram. If we landed in the node labeled ⊤, then the object is of the type in question.

In analogous fashion, we may reconstruct a Common Lisp type specifier given an ROBDD, as shown in Figure 6. To do so, simply find the set of all paths from the top-most node to the ⊤ leaf node (the figure has 4 such paths). Each such path contributes a conjunction, having the base types either negated or not depending on whether the positive or negative arrow is follows. This algorithm applied to the ROBDD in Figure 6 results in the type specifier:

```
(or (and (not fixnum) (not number))
    (and (not fixnum) number unsigned)
    (and fixnum (not unsigned))
    (and fixnum unsigned (eql 42)))
```

The path specifier is rarely the simplest representation of the type. The fact that the *derived* type specifier is different than the original one, `(or (not number) (eql 42) (and fixnum (not unsigned)) (and unsigned (not fixnum)))`, is not a problem. There are infinitely many type specifiers which represent the same type in Common Lisp; nevertheless they all correspond to the same ROBDD.

There are many conventions for drawing ROBDDs (and similar decision diagrams) in the literature. We use the convention that red dashed arrows denote a negative result of the type check at a node, and the green solid arrow represents a positive result.

An important feature of this ROBDD is that there are no redundant type checks. Part of this comes from an inherent feature of ROBDDs, namely that no path from the root node to a leaf node ever passes through the same type check twice. But in addition to that, our ROBDDs are aware of the Common Lisp type system. They remove redundant supertype checks. As seen in Figure 6, `number` is only checked if the `fixnum` check failed. Why? Because, if an object is not a `number`, then it is necessarily not a `fixnum`. Likewise `(eql 42)` is only checked if the `fixnum` check succeeded, because if an object is not a `fixnum` then it is necessarily not equal to 42.

6 EXPANSION OF `OPTIMIZED-TYPECASE`

Newton *et al.* [NV18] explain how to use pseudo-predicates in conjunction with the Common Lisp `satisfies` type to represent an arbitrary call to `typecase` as an ROBDD. Take the `optimized-typecase` in Figure 7 as an example.

The technique is to convert such a `typecase` invocation into a type specifier, by substituting appropriate so-called pseudo-predicates in place of the various body forms. A pseudo-predicate is

```
(or (and (and unsigned (not (eql 42)))
         (satisfies P1))
    (and (eql 42)
         (not (and unsigned (not (eql 42))))
         (satisfies P2))
    (and (and number (not (eql 42)) (not fixnum))
         (not (and unsigned (not (eql 42))))
         (not (eql 42))
         (satisfies P3))
    (and fixnum
         (not (and unsigned (not (eql 42))))
         (not (eql 42))
         (not (and number
                   (not (eql 42))
                   (not fixnum)))
         (satisfies P4)))
```

Figure 8: Type specifier equivalent to `typecase` from Figure 7

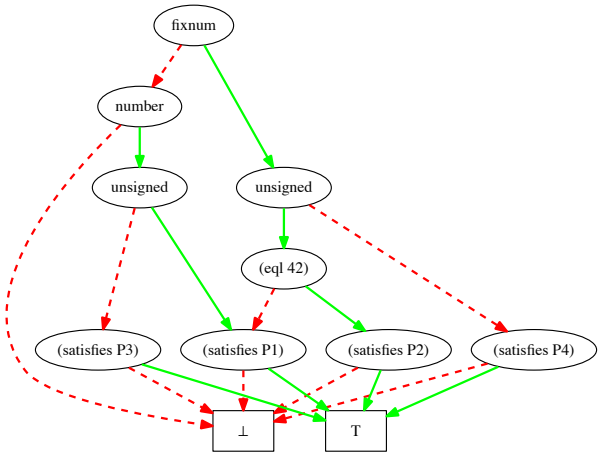

Figure 9: ROBDD from `typecase` clauses in Example 7

an object representing a function name, that if called would return a Boolean value, but may also have a side-effect. However, it has been carefully arranged that manipulating the type specifier never calls this function, and never accidentally optimizes it away. For example the `typecase` in Figure 7 is converted to the type specifier in Figure 8. Note that `P1`, `P2` *etc.* appear as predicates in combination with the `satisfies` type specifier. Thus the *pseudo* terminology—it appears as a predicate, but would execute a side-effect if called.

Using this type specifier, the ROBDD shown in Figure 9 is generated.

Similar to how the code shown in Figure 5 is generated we can also generate optimized code from an ROBDD. Such code is shown in Figure 10.

```
(lambda (obj)
 (tagbody
   L1 (if (typep obj 'fixnum)
          (go L2)
          (go L4))
   L2 (if (typep obj 'unsigned)
          (go L3)
          (go P4))
   L3 (if (typep obj '(eql 42))
          (go P2)
          (go P1))
   L4 (if (typep obj 'number)
          (go L5)
          (return nil))
   L5 (if (typep obj 'unsigned)
          (go P1)
          (go P3))
   P1 (return (progn body-forms-1...))
   P2 (return (progn body-forms-2...))
   P3 (return (progn body-forms-3...))
   P4 (return (progn body-forms-4...))))
```

Figure 10: Alternate expansion of Example 7 using `tagbody`/`go`

7 RELATED WORK

A discussion of the theory of rational languages in which our research is grounded, may be found in [HMU06, Chapters 3,4].

Weitz [Wei15] introduces portable Perl-compatible regular expressions for Common Lisp including an s-expression-based DSL syntax. Since users of regular expressions targeting string matching prefer to encode their regular expressions into strings, Weitz also provides a string based surface syntax. Newton *et al.* [New16] also provide such a string-based syntax for regular type expressions in a very special case; *i.e.* the case where the target sequence is a string, and the patterns only concert character identity. We leave the question open to future research as to whether such a conventional string-based regular expression syntax might be useful to describe types in sequences.

The **rte** type along with `typecase` enables something similar to pattern matching in the XDuce language [HVP05]. The XDuce language allows the programmer to define a set of functions with various lambda lists, each of which serves as a pattern available to match particular target structure within an XML document. Which function gets executed depends on which lambda list matches the data found in the XML data structure.

XDuce introduces a concept called *regular expression types* which indeed seems very similar to *regular type expressions*. In [HVP05] Hosoya *et al.* introduce a *semantic type* approach to describe a system which enables their compiler to guarantee that an XML document conform to the intended type. The paper deals heavily with assuring that the regular expression types are well defined when defined recursively, and that decisions about subtype relationships can be calculated and exploited.

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

The concept of regular trees, is more general than what **rte** supports, posing interesting questions regarding apparent shortcomings of our approach. The semantic typing concept described in [HVP05] indeed seems to have many parallels with the Common Lisp type system in that types are defined by a set of objects, and sub-types correspond to subsets thereof. These parallels would suggest further research opportunities related to **rte** and Common Lisp. However, the limitation that **rte** cannot be used to express trees of arbitrary depth, as discussed in Section 4.1, seems to be a significant limitation of the Common Lisp type system. Furthermore, the use of `satisfies` in the **rte** type definition, seriously limits the `subtypep` function's ability to reason about the type. Consequently, programs cannot always use `subtypep` to decide whether two **rte** types are disjoint or equivalent, or even if a particular **rte** type is empty. Neither can the compiler dependably use `subtypep` to make similar decisions to avoid redundant assertions in function declarations.

Recently, several languages have started to introduce tuple types (C++ [Str13, Jos12], Scala [OSV08, CB14], Rust [Bla15]), and our work provides similar capability of such tuple types for a dynamic programming language. The Shapeless [Che17] library allows Scala programmers to exploit the type-level programming capabilities through heterogeneous lists.

Another commonly used algorithm for constructing a DFA was inspired by Ken Thompson [YD14, Xin04] and involves decomposing a rational expression into a small number of cases such as base variable, concatenation, disjunction, and Kleene star, then following a graphical template substitution for each case. While this algorithm is easy to implement, it has a serious limitation. It is not able to easily express automata resulting from the intersection or complementation of rational expressions. We rejected this approach as we would like to support regular type expressions containing the keywords `:and` and `:not`, such as in `(:and (:* (:cat t integer)) (:not (:* (:cat float t))))`.

There is a large amount of literature about Binary Decision Diagrams of many varieties [Bry86, Bry92, Ake78, Col13, And99]. In particular Knuth [Knu09, Section 7.1.4] discusses worst-case and average sizes. Newton *et al.* [NVC17] discuss how the Reduced Ordered Binary Decision Diagram (ROBDD) can be used to manipulate type specifiers, especially in the presence of subtypes. Castagna [Cas16] discusses the use of ROBDDs (he calls them BDDs in that article) to perform type algebra in type systems which treat types as sets [HVP05, CL17, Ans94].

BDDs have been used in electronic circuit generation[CBM90], verification, symbolic model checking[BCM+92], and type system models such as in XDuce [HVP05]. None of these sources discusses how to extend the BDD representation to support subtypes.

Common Lisp does not provide explicit pattern matching [Aug85] capabilities, although several systems have been proposed such as Optima[1] and Trivia[2]. Pierce [Pie02, p. 341] explains that the addition of a `typecase`-like facility (which he calls `typecase`) to a typed λ-calculus permits arbitrary run-time pattern matching.

[1] `https://github.com/m2ym/optima`
[2] `https://github.com/guicho271828/trivia`

Decision tree techniques are useful in the efficient compilation of pattern matching constructs in functional languages[Mar08]. An important concern in pattern matching compilation is finding the best ordering of the variables which is known to be coNP-Complete [Bry86]. However, when using BDDs to represent type specifiers, we obtain representation (pointer) equality, simply by using a consistent ordering; finding the *best* ordering is not necessary for our application.

Charniak [CM85] explains a technique called *discrimination net* to represent nested if-then-else machines. PCL (Portable Common Loops)[BKK+86] which is the heart of many implementations of the CLOS (the Common Lisp Object System) [Kee89, Ano87], in particular the implementation within SBCL [New15], uses discrimination nets to optimize generic function dispatch. In this article we do not investigate the similarities and differences between the discrimination net approach and the ROBDD approach, except to note that there is some obvious overlap in the problem space they purport to solve.

8 CONCLUSION AND PERSPECTIVES

It is not clear whether Common Lisp could provide a way for a type definition in an application program to extend the behavior of `subtypep`. Having such a capability would allow such an extension for **rte**. Rational language theory provides a well defined algorithm for deciding such questions given the relevant rational expressions [HMU06, Sections 4.1.1, 4.2.1].

This article demonstrates the use of reflection and meta-programming to provide an elegant tool to the application programmer, namely regular type expressions. These regular type expressions extend the already reflective Common Lisp type system, enabling the run-time program more flexibility in reasoning about heterogeneous lists.

Meta-programming is essential in our integration into the Common Lisp type system. Our approach involves computationally intensive finite state machine based calculations which are executed whenever the compiler encounters a new **rte** type declaration. The result is that these calculated functions are thereafter available at run-time for efficient type matching of lists in question.

REFERENCES

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978.

[And99] Henrik Reif Andersen. An introduction to binary decision diagrams. Technical report, Course Notes on the WWW, 1999.

[Ano87] Anonymous. The Common Lisp Object Standard (CLOS), October 1987. 1 videocassette (VHS) (53 min.).

[Ans94] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[Aug85] Lennart Augustsson. Compiling pattern matching. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, Stefik, M., and F. Zdybel. Common Loops, merging Lisp and object-oriented programming. *j-SIGPLAN*, 21(11):17–29, November 1986.

[Bla15] Jim Blandy. *The Rust Programming Language: Fast, Safe, and Beautiful.* O'Reilly Media, Inc., 2015.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Jun 1990.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.

[Bur13] Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[Cas16] Giuseppe Castagna. Covariance and contravariance: a fresh look at an old issue. Technical report, CNRS, 2016.

[CB14] Paul Chiusano and Rnar Bjarnason. *Functional Programming in Scala.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2014.

[CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, London, UK, UK, 1990. Springer-Verlag.

[CD10] Pascal Costanza and Theo D'Hondt. Embedding hygiene-compatible macros in an unhygienic macro system. *Journal of Universal Computer Science*, 16(2):271–295, jan 2010.

[Che17] Tongfei Chen. Typesafe abstractions for tensor operations. *CoRR*, abs/1710.06892, 2017.

[CL17] G. Castagna and V. Lanvin. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.*, (1, ICFP '17, Article 41), sep 2017.

[CM85] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[Col13] Maximilien Colange. *Symmetry Reduction and Symbolic Data Structures for Model Checking of Distributed Systems.* Thèse de doctorat, Laboratoire de l'Informatique de Paris VI, Université Pierre-et-Marie-Curie, France, December 2013.

[GPS98] Clemens Gröpl, Hans Jürgen Prömel, and Anand Srivastav. Size and structure of random ordered binary decision diagrams. In *STACS 98*, pages 238–248. Springer Berlin Heidelberg, 1998.

[Gra96] Paul Graham. *ANSI Common Lisp.* Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hro02] Juraj Hromkovič. Descriptional complexity of finite automata: Concepts and open problems. *J. Autom. Lang. Comb.*, 7(4):519–531, September 2002.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, January 2005.

[Jos12] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley Professional, 2nd edition, 2012.

[Kay69] Alan C. Kay. *The Reactive Engine.* PhD thesis, University of Hamburg, 1969.

[Kee89] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[Knu09] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 12th edition, 2009.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[Mar08] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[McI60] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Commun. ACM*, 3:214–220, April 1960.

[NDV16] Jim Newton, Akim Demaille, and Didier Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In *European Lisp Symposium*, Kraków, Poland, May 2016.

[New15] William H. Newman. Steel Bank Common Lisp user manual, 2015.

[New16] Jim Newton. Report: Efficient Dynamic Type Checking of Heterogeneous Sequences. Technical report, EPITA/LRDE, 2016.

[New17] Jim Newton. Analysis of algorithms calculating the maximal disjoint decomposition of a set. Technical report, EPITA/LRDE, 2017.

[NV18] Jim Newton and Didier Verna. Strategies for typecase optimization. In *European Lisp Symposium*, Marbella, Spain, April 2018.

[NVC17] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic manipulation of Common Lisp type specifiers. In *European Lisp Symposium*, Brussels, Belgium, April 2017.

[ORT09] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[Rho09] Christophe Rhodes. User-extensible Sequences in Common Lisp. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 13:1–13:14, New York, NY, USA, 2009. ACM.

[ST98] Karsten Strehl and Lothar Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '98, pages 686–692, New York, NY, USA, 1998. ACM.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions*. John Wiley & Sons, Inc., New York, NY, USA, 1987.

[Wei15] Edmund Weitz. *Common Lisp Recipes: A Problem-solution Approach*. Apress, 2015.

[Xin04] Guangming Xing. Minimized Thompson NFA. *Int. J. Comput. Math.*, 81(9):1097–1106, 2004.

[YD14] Francois Yvon and Akim Demaille. *Théorie des Langages Rationnels*. EPITA LRDE, 2014. Lecture notes.