Finite Automata Theory Based Optimization of Conditional Variable Binding

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

ABSTRACT

We present an efficient and highly optimized implementation of `destructuring-case` in Common Lisp. This macro allows the selection of the most appropriate destructuring lambda list of several given based on structure and types of data at run-time and thereafter dispatches to the corresponding code branch. We examine an optimization technique, based on finite automata theory applied to conditional variable binding and execution, and type-based pattern matching on Common Lisp sequences. A risk of inefficiency associated with a naive implementation of `destructuring-case` is that the candidate expression being examined may be traversed multiple times, once for each clause whose format fails to match, and finally once for the successful match. We have implemented `destructuring-case` in such a way to avoid multiple traversals of the candidate expression. This article explains how this optimization has been implemented.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; *Type theory*;

ACM Reference Format:
Jim E. Newton and Didier Verna. 2019. Finite Automata Theory Based Optimization of Conditional Variable Binding. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2635402

1 INTRODUCTION

The Common Lisp macro `destructuring-bind` [?] binds the variables specified in a given lambda list to the corresponding values in the tree structure resulting from the evaluation of a given expression. However, in the case that the tree structure of the expression does not coincide with the given lambda list, a run-time error is signaled. This error may pose a challenge to the programmer. The problem, simply stated, is that the destructuring lambda list [? , Section 3.4.5] is specified at compile time, and the expression is evaluated at run-time. Thus, it may not be possible to know until run-time that the input data is problematic. In certain cases the programmer would like to specify the run-time behavior to take if the match fails, rather than having an error signaled. This behavior cannot be specified portably using the condition system [? , Chapter 9], because the condition signaled is simply of type `error` with no additional information about exactly what failed. Furthermore, the programmer may not wish to signal an error at all, but rather detect the actual run-time pattern of the input data and proceed differently depending on which format of data is discovered.

```
(destructuring-case expression
  ((X Y)
   (declare (type fixnum X Y))
   :clause-1)
  ((X Y)
   (declare (type fixnum X)
            (type integer Y))
   :clause-2)
  ((X Y)
   (declare (type (or string fixnum) X)
            (type number Y))
   :clause-3))
```

Figure 1: Example of `destructuring-case` usage.

We presented `destructuring-case` in [?] as a mechanism to test run-time adherence of the destructuring lambda list to the value of a candidate expression. An example usage of this macro can be seen in Figure 1. This example shows three clauses, each with the same lambda list, `(X Y)`, but with different type declarations. In general, a usage of `destructuring-case` may use radically different lambda lists, which differ in number of variables, having different `&optional` and `&key` sections, and also using different hierarchical structure of the variables.

The semantics of `destructuring-case` are that the value of the given `expression` is tested in turn against each of the given destructuring lambda lists, until a match is found, *i.e.* a match in both hierarchical structure and type of values. Only at such time are the indicated consequent expressions or any default values evaluated. This restriction is especially important if there are side-effects in the default values of optional arguments in the lambda lists such as `(... &optional (x (incf *global-var*)))`.

ELS'19, April 01–02 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2635402

```
(rte-case expression
  ((:cat fixnum fixnum)
   (destructuring-bind (X Y) expression
     :clause-1))
  ((:cat fixnum integer)
   (destructuring-bind (X Y) expression
     :clause-2))
  ((:cat (or string fixnum) number)
   (destructuring-bind (X Y) expression
     :clause-3)))
```

Figure 2: Expansion of `destructuring-case` from Figure 1 into `rte-case`.

```
(rte-case expression
  ((:cat fixnum fixnum)
    :clause-1)
  ((:cat fixnum integer)
    :clause-2)
  ((:cat (or string fixnum) number)
    :clause-3))
```

Figure 3: Simple example of `rte-case` from Figure 2.

The implementations of the macros discussed in this article, including `destructuring-case`, `rte-case`, `rte-ecase`, and `bdd-typecase`, are available in Quicklisp[1] via the package `:rte`.

2 FROM DESTRUCTURING-CASE TO RTE-CASE

Our implementation of `destructuring-case` converts its input of destructuring lambda lists to rte (regular type expression) and then outputs an invocation of `rte-case`. The essential part of such an expansion is shown in Figure 2. An rte, introduced in [?], is Common Lisp syntax to specify a set of sequences, *i.e.* a subtype of the `sequence` type. We explain in Section 2.2 how a destructuring lambda list is converted to an rte.

As can be seen in Figure 2, each destructuring lambda list has been converted to an rte such as `(:cat fixnum fixnum)` in the first clause, followed by a call to `destructuring-bind`. As is implied by the syntax, the `destructuring-bind` will only be executed at runtime if the value of the candidate expression matches the pattern designated by the rte.

We further notice in the simplistic example shown in Figure 2, that no `destructuring-bind` in the `rte-case` expansion plays any role. The variables bound by the `destructuring-bind` are not used in the expressions which follow. Therefore, in our further discussion we will refer to the even simpler, semantically equivalent code in Figure 3.

A straightforward expansion of `rte-case` might include successive type checks of `expression` such as suggested in Figure 4. Such an expansion would be semantically correct, but inefficient because the sequence `expression` would be traversed three times in the worst case, to determine which consequent clause to evaluate. As will be seen, our technique eliminates these redundant traversals, allowing one single traversal of the sequence to be executed and thereby determining which consequent expressions to evaluate.[2]

```
(typecase expression
  ((rte (:cat fixnum fixnum))
   :clause-1)
  ((rte (:cat fixnum integer))
   :clause-2)
  ((rte (:cat (or string fixnum) number))
   :clause-3))
```

Figure 4: Naive expansion of `rte-case` from Figure 2

2.1 Examples of rte Syntax

The grammar an rte is explicitly detailed in [?]. Nevertheless, the basic grammar can be understood intuitively, assuming the reader has a basic understanding of string-based regular expression syntax. The concatenation operator, `:cat` specifies a sequences successive elements: *e.g.*, `(:cat fixnum string)` denotes a sequence of exactly two elements, the first of type `fixnum` and the second of type `string`. To make the `string` optional use the syntax `(:cat fixnum (:? string))`. To specify the occurrence, zero or more times, of `fixnum` followed by an optional string, use `(:cat (:* fixnum) (:? string))`. Substitute `:+` for `:*` to express an occurrence of one or more times. Finally, expressions may be combined logically using `:and`, `:or`, and `:not`, *e.g.*, `(:or (:cat fixnum string) (:+ (:not number)))`.

2.2 From Destructuring Lambda List to rte

In this section we summarize how a destructuring lambda list and associated type declarations may be converted into an rte. The conversion procedure is explained in more detail in [?].

The set of lists which are valid argument lists for a given invocation of `destructuring-bind` with an optional set of type declarations can be characterized by an rte. A destructuring lambda list, such as used in `destructuring-bind`, specifies a required portion, denoted by a leading sequence of variables; an optional portion, delimited by `&optional`; and a repeating portion of keyword value pairs, delimited by `&key`. To construct the rte corresponding to a given destructuring lambda list, we construct the *required-rte*, the *optional-rte*, and the *repeating-rte*, and concatenate them using the `:cat` operator.

`(:cat` *required-rte optional-rte repeating-rte* `)`

As an example, consider the lambda list shown in Figure 5. The required portion and optional portions are easy.

required-rte = `(:cat string string)`
optional-rte = `(:? list)`

[1] Quicklisp, `https://www.quicklisp.org/`, is a public repository, maintained by Zach Beane, consisting of user contributed Common Lisp libraries.

[2] The reader may well notice that a fourth traversal is also necessary to evaluate the `destructuring-bind` which is present in each of the consequent clauses. In this paper we do not address the elimination of this fourth traversal.

```
(destructuring-bind (A B &optional Q &key X Y)
    expression
  (declare (type string A B)
           (type list Q)
           (type real X)
           (type integer Y))
  ...)
```

Figure 5: Example `destructuring-bind` with declarations

The repeating portion deserves careful attention; we consider two restrictions.

(1) If `&allow-other-keys` *is not given*, such as is the case in Figure 5, then the only allowed keywords are those explicitly specified. In our case the only allowed keywords are `:X` and `:Y`, meaning the repeating portion is also of the form

```
(:* (:cat (member :X :Y) t)).
```

(2) Type declarations such as `(declare real X)` only restrict the value associated with the *first* occurrence of each keyword in an argument list, because only the first such occurrence is bound the the associated variable [? , Section 3.3.4]. A keyword portion of the argument list such as `(:X 1.2 :X 'not-real)` is perfectly valid, whereas `(:X 'not-real :X 1.2)` is not. Thus, we iterate over all specified keywords, generating one pattern for each. The pattern handling `&key X` requires that either there is either no `:X` given, or that the first `:X` is followed by a `real`. See the note `restriction 2` in Figure 6.

Putting all these restrictions together, we have the rte in Figure 6 representing the `destructuring-bind` with type declarations in Figure 5.

There are several other features of `destructuring-bind` which are supported by `destructuring-case`, but whose details we omit in this discussion, including tree structure variables/data, default values, *supplied-p-parameter*, `&allow-other-keys`, and others.

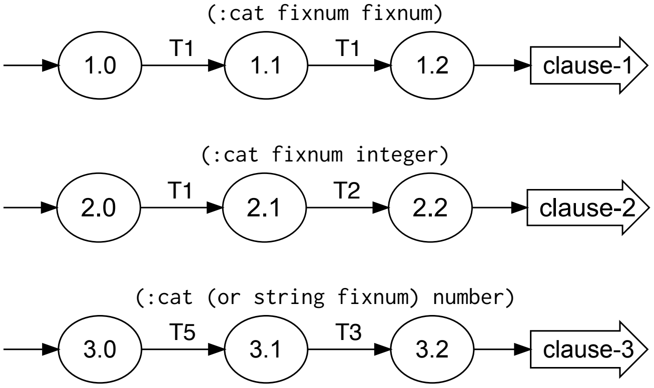
3 FROM RTE-CASE TO INDIVIDUAL DFAS

Each rte shown in Figure 3 can be converted to efficient type checking Common Lisp code, as explained in [?]. Such conversion involves first converting each rte to a deterministic finite automaton (DFA), where the transition labels represent type checks for successive elements of the candidate expression. Figure 7 shows the three DFAs corresponding to the `rte-case` in Figure 3.

We now summarize how a deterministic finite automata (DFA) is constructed, given an rte. Some approaches to such generation, such as [? ?], involve constructing a non-deterministic finite automaton and thereafter determinizing it. We use the technique presented by Brzozowski [?] and clarified by Owens [?]. The Brzozowski algorithm uses a technique called the rational derivative, to construct a DFA, and thereby obviating the necessity to determinize the result. In [? ?], we explain how the rational derivative can be extended to accommodate Common Lisp types, in particular rather than calculating the rational derivative (as Owens suggests) with respect to each letter of the alphabet, instead we calculate the derivative with respect each type calculated in the maximal disjoint type decomposition as explained in [?].

```
(:cat
 ;; required-rte
 (:cat string string)

 ;; optional-rte
 (:? list)

 ;; repeating-rte
 (:and
  ;; restriction 1
  (:* (:cat (member :X :Y) t))

  ;; restriction 2 for :X real
  (:or (:* (:cat (:not (eql :X)) t))
       (:cat (:* (:cat (:not (eql :X)) t))
             (eql :X) real
             (:* t)))

  ;; restriction 2 for :Y integer
  (:or (:* (:cat (:not (eql :Y)) t))
       (:cat (:* (:cat (:not (eql :Y)) t))
             (eql :Y) integer
             (:* t)))))
```

Figure 6: The rte representing the `destructuring-bind` and type declarations from Figure 5.



Label	Type specifier
T_1	`fixnum`
T_2	`integer`
T_3	`number`
T_5	`(or string fixnum)`

Figure 7: Automata for clauses of `rte-case` in Figure 2

3.1 Constructing States and Transitions

The algorithm can be summarized as follows. Each state in the DFA represents all the possible futures which are accepting. Moreover, there is a (not necessarily unique) rte which expresses that set of futures. For example, let:

$$P_1 = \texttt{(:or (:cat number string) (:cat fixnum float))}$$

be the rte representing all the sequences of either a `number` followed by a `string` or a `fixnum` followed by a `float`. Suppose there is a state in the DFA associated with this rte. Now we consider all the possible types of the first element of such a sequence. And for each such first element type, we calculate what the remaining future would be given that the first element of that type. If the first element is a `fixnum`, then the future is a sequence containing either a `string` or a `float`. Such a sequence is denoted by the rte `(:or string float)`. In terms of the rational derivative we say:

$$P_2 = \partial_{\texttt{fixnum}} P_1 = \texttt{(:or string float)}\,.$$

If, on the other hand, the first element is not a `fixnum` but is a `number`, then the remaining sequence whose only element is a `string`. That is to say:

$$P_3 = \partial_{\texttt{(and number (not fixnum))}} P_1 = \texttt{string}\,.$$

Since there is no other possible first element of P_1, we construct two additional states, P_2 and P_3 and construct two transitions $P_1 \longrightarrow P_2$ labeled `fixnum`, and $P_1 \longrightarrow P_3$ labeled `(and number (not fixnum))`.

We continue this process until all the futures of each state have been calculated, generating all the possible states, and all the possible transitions between the states.

3.2 Associating Code with Accepting States

DFAs used for matching pattern languages such as regular expressions, normally represent Boolean functions; returning TRUE if the sequence matches the expression, and FALSE otherwise. In our case each accepting state of the DFAs in Figure 7 indicate which code paths to take in the originating `rte-case`, Figure 3. This problem is easily addressed. We have simply extended our `state` object (CLOS class [? ?]) to contain a slot indicating a piece of continuation code to be serialized in the final macro expansion.

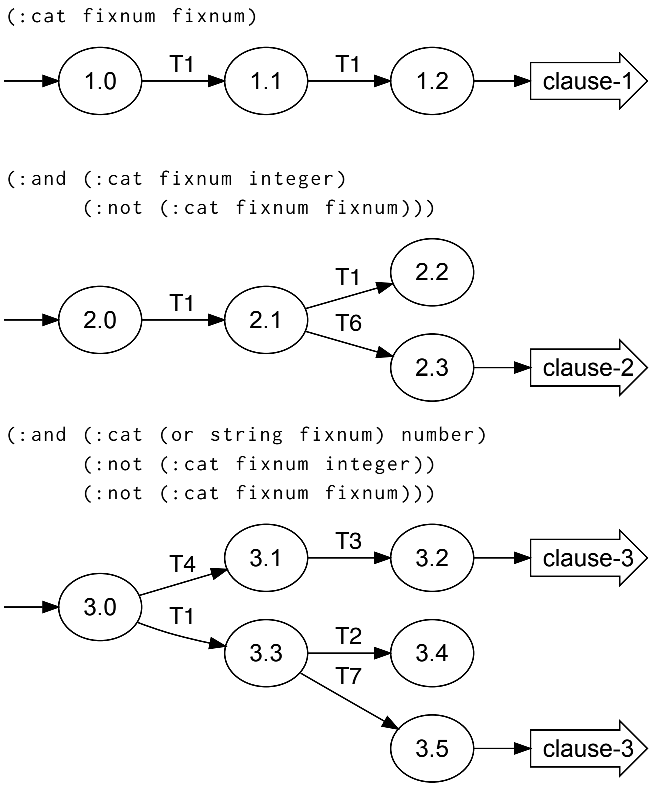
3.3 Overlapping Clauses

The *synchronized cross-product* (SXP) of two or more given DFAs is a single DFA whose behavior simultaneously emulates the behavior of the given DFAs. Typically such a cross-product implements the intersection or union languages of the input DFAs; however the semantics of such a cross-product can be taken to be any Boolean combination of the input.

For example, to implement the symmetric difference language we apply the Boolean XOR function; a state, X, in the SXP, corresponding to states A and B from two given DFAs, is marked as an *accepting* state if A XOR B are accepting (if either but not both are accepting). In our case we would like to select the code for evaluation corresponding to the code appearing first in the original `destructuring-case`; so we need priority based selection, rather than simply a Boolean function.

```
(rte-case expression
  ((:cat fixnum fixnum)
   :clause-1)
  ((:and (:cat fixnum integer)
         (:not (:cat fixnum fixnum)))
   :clause-2)
  ((:and (:cat (or string fixnum) number)
         (:not (:cat fixnum fixnum))
         (:not (:cat fixnum fixnum)))
   :clause-3))
```

Figure 8: Example of `rte-case` with pairwise disjoint patterns



```
(:cat fixnum fixnum)
```

```
(:and (:cat fixnum integer)
      (:not (:cat fixnum fixnum)))
```

```
(:and (:cat (or string fixnum) number)
      (:not (:cat fixnum integer))
      (:not (:cat fixnum fixnum)))
```

Label	Type specifier
T_1	`fixnum`
T_2	`integer`
T_3	`number`
T_4	`string`
T_6	`(and (not fixnum) integer)`
T_7	`(and (not integer) number)`

Figure 9: DFAs for disjoined clause-1, clause-2, and clause-3

An important property of the behavior of `rte-case` is that if more than one pattern matches the expression in question, then the clause appearing first has priority over the others. For example, in the code in Figure 3, if the value of `expression` is the list `(1 2)`, then all three rtes match; nevertheless `:clause-1` must be the return value.

An approach of addressing this ambiguity is to extend or augment the patterns so that they are mutually exclusive; *i.e.* assure that no two patterns simultaneously match any candidate expression. The code shown in Figure 8 is equivalent to that in Figure 3 but any input expression, `(1 2)`, for example, matches at most one pattern. This pattern augmentation can be accomplished as a code transformation. The pattern corresponding to `:clause-1` is unchanged, but the subsequent clauses have been augmented to emphasize that those clauses are never reached if any prior pattern matches.

These rtes correspond to the DFAs shown in Figure 9. The first DFA is exactly the same as before, but we notice in the second DFA that the state labeled 2.2 is non-coäccessible; *i.e.*, there is no path from state 2.2 to any accepting state. This non-useful state corresponds to `(:not (:cat fixnum fixnum))` in the input pattern, and it enforces that a sequence consisting of two objects of type `fixnum`, is a rejected sequence rather than a matching sequence. The third DFA in the figure contains a similar state, 3.4, but in addition, contains two states 3.2 and 3.5 which are equivalent to each other.

The disjoining process described here produces DFAs which have redundant or non-coäccessible states. Despite this fact, these slightly more complex DFAs play an important role in the SXP construction, because the process guarantees that the SXP construction will never encounter a situation where it must choose between two different pieces of code to execute on reaching an acceptance condition. If attempting to calculate the union of the three DFAs shown in Figure 7, the algorithm would have to deal with the fact that a sequence of `(1 2)` at run time should return `:clause-1` rather than `:clause-2`. However, if calculating the union of the DFAs from Figure 9, such ambiguity is averted. The union can be performed purely algebraically, with no consideration or order of priority.

4 MERGING DFAS INTO SYNCHRONIZED CROSS-PRODUCT DFA

We explain in detail in [?] how the type check associated with an rte is compiled to efficient Common Lisp code by first converting it to a deterministic finite automaton. It is further pointed out in the perspectives of [?] that it is desirable to *merge* these automata into a single automaton in order to share states between the various automata which serve the same function, and also to eliminate redundant traversals of the candidate expression. Having a single automaton which implements the union of the mutually exclusive patterns enables the candidate list to be traversed once and thereby matching any one of the expressions specified in the various clauses of the `rte-case`.

One advantage of the conversion from destructuring lambda list to rte is that rtes support an algebra sufficient for expressing sets of non-overlapping types, resulting in mutually exclusive patterns in the expansion to `rte-case`. As an additional feature of the implementation of `rte-case`, we have arranged so that it treats the code in Figure 3 and Figure 8 exactly the same, internally disjoining patterns which are not already disjoint.

The following is an explanation of how several automata are merged into such a single automaton.

We would like to merge the three DFAs shown in Figure 9 into a single DFA. There are well known techniques for merging multiple DFAs [? ?] into the SXP DFA. These techniques are not general enough for several reasons which we address in our approach.

dfa_2	dfa_3	**intersection**	**Target State**
T_1	T_1	T_1	(2.1, 3.3)
T_1	T_4	$\emptyset$	
T_1	$\top \setminus (T_1 \cup T_4)$	$\emptyset$	
$\top \setminus T_1$	T_1	$\emptyset$	
$\top \setminus T_1$	T_4	T_4	$(\bot,\ 3.1)$
$\top \setminus T_1$	$\top \setminus (T_1 \cup T_4)$	$\top \setminus (T_1 \cup T_4)$	$(\bot,\ \bot)$

Figure 10: Transition Computation for $\mathrm{dfa}_2 \times \mathrm{dfa}_3$

It is not necessary to explicitly consider the SXP of more than two DFAs, because the operation is associative. Therefore, given the Common Lisp function `synchronized-product`, we may compute the SXP of one or more DFAs as a call to `cl:reduce`.

```
(reduce #'synchronized-product dfas)
```

4.1 Calculating States and Transitions

We consider constructing the SXP of two DFAs, `dfa-1` (with n states) and `dfa-2` (with m states). We construct a DFA, `dfa-3`, having $m \times n$ states, worst case; one state for each pair (x, y) with $x \in \mathrm{dfa}_1$ and $y \in \mathrm{dfa}_2$. Fortunately, this worst case does not often occur in practice as many of the states are not accessible. For example, if computing the SXP of the first two DFAs of Figure 9, there is no possible input sequence which would put dfa_1 into state 1.1 while putting dfa_2 into state 2.2. Thus there will be no state in the product DFA corresponding to (1.1, 2.2).

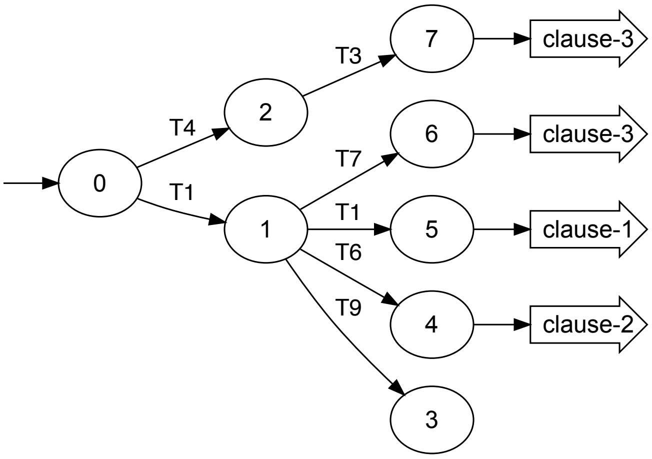
An efficient algorithm is described in [?]. We seed a work list with the one initial state. Next, we traverse the work list, growing it by adding new states as we construct them. All possible input types are considered for each state, and all possible transitions are generated.

An example will make this clearer. First start with dfa_2 and dfa_3, the second and third DFAs illustrated in Figure 9. The states list is initialized to $S = \{(2.0,\ 3.0)\}$.

We examine the behavior of states 2.0 and 3.0. We must characterize the behavior for every possible input. This infinite set of potential input values is partitioned into several disjoint types: those annotated on transitions exiting state 2.0 and 3.0, and the complement of their union. This complement type represents the set of all values for which an implicit transition leads to the virtual so-called *sync* state, denoted $\bot$. The sync state is a state which has exactly one exiting, all encompassing, transition: $\bot \xrightarrow{\top} \bot$.

State 2.0 has one explicit transition, namely $2.0 \xrightarrow{T_1} 2.1$. Thus, there is an implicit complement transition $2.0 \xrightarrow{\top\setminus T_1} \bot$, where $\top$ represents the universal type. State 3.0 has two explicit transitions: namely $3.0 \xrightarrow{T_1} 3.3$ and $3.0 \xrightarrow{T_4} 3.1$. Thus, there is an implicit complement transition $3.0 \xrightarrow{\top\setminus(T_1\cup T_4)} \bot$.

To compute the transitions from $(2.0,\ 3.0)$, we must consider all six pairwise intersections between the transition types of the two states (2.0 and 3.0). These intersections are shown in Figure 10, which also indicates the target states in the three non-empty cases.



Label	Type specifier
T_1	`fixnum`
T_3	`number`
T_4	`string`
T_6	`(and (not fixnum) integer)`
T_7	`(and (not integer) number)`

T_9	`(and integer`
	`(or (not integer) fixnum)`
	`(not fixnum))`

Figure 11: DFA for `rte-case` not yet reduced

Given an input of type `fixnum`, dfa$_2$ transitions from state 2.0 to state 2.1; and given the same input dfa$_3$ transitions from state 3.0 to state 3.3. So we add (2.1, 3.3) to S; $S = S = \{(2.0,\ 3.0),\ (2.1,\ 3.3)\}$, and add transition $(2.0,\ 3.0) \xrightarrow{T_1} (2.1,\ 3.3)$. Likewise, given an input of type `string`, dfa$_2$ transitions from state 2.0 to state $\perp$; and given the same input dfa$_3$ transitions from state 3.0 to state 3.1. So we add $(\perp,\ 3.1)$ to S; $S = S = \{(2.0,\ 3.0),\ (2.1,\ 3.3),\ (\perp,\ 3.1)\}$, and add transition $(2.0,\ 3.0) \xrightarrow{T_4} (\perp,\ 3.1)$. Finally, given an input of type `(and (not fixnum) (not string))`, dfa$_2$ transitions from state 2.0 to state $\perp$, and dfa$_3$ transitions from state 3.0 to state $\perp$. The state $(\perp, \perp)$ is the sync state of the cross product DFA so we need generate no additional transition from (2.0, 3.0).

Next, we to apply the same procedure to calculate any new states and transitions of any newly added elements of S. We continue the procedure until all elements of S have been visited, and no new states were generated.

After dfa$_2$ × dfa$_3$ has been computed, we can repeat the process via the `reduce` operation mentioned above to compute dfa$_1$ × dfa$_2$ × dfa$_3$. This procedure constructs a DFA isomorphic to that shown in Figure 11. We say *isomorphic* because the choice of state names is arbitrary. Figure 11 has states named 0 through 7 rather name names such as (1.0, 2.0, 3.0), (1.1, 2.1, 3.3) as suggested in the procedure description in Section 4.1.

The DFA shown in Figure 11 is not in minimal form. It has a non-coäccessible state, 3, from which there is no path to an accepting state. It also has indistinguishable states; *e.g.*, states 6 and 7 have the exact same future, albeit a trivial one of just returning the symbol `clause-3`. Since each of the states in the computed DFA and each of the transitions contribute to the number of lines of Common Lisp code which will be generated when the DFA is serialized in Section 5, we should simplify this DFA to reduce the lines of redundant code in the final macro expansion.

$s \in \mathbb{S}$	$v \in \Upsilon$	$\delta(s, v)$
0	T_1	1
0	T_4	2
1	T_1	5
1	T_6	4
1	T_7	6
2	T_3	7

$s \in \mathbb{S}$	$v \in \Upsilon$	$\psi_1(s, v) \in \Pi_0$
0	T_1	$\{0, 1, 2\}$
0	T_4	$\{0, 1, 2\}$
1	T_1	$\{5\}$
1	T_6	$\{4\}$
1	T_7	$\{6, 7\}$
2	T_3	$\{6, 7\}$

$s \in \mathbb{S}$	$\Phi_1(s)$
0	$\{\ (T_1, \{0, 1, 2\}),\ (T_4, \{0, 1, 2\})\ \}$
1	$\{\ (T_1, \{5\}),\ (T_6, \{4\}),\ (T_7, \{6, 7\})\ \}$
2	$\{\ (T_3, \{6, 7\})\ \}$
4	$\emptyset$
6	$\emptyset$
7	$\emptyset$

Figure 12: All values of the δ, ψ_1, and Φ_1 functions.

We eliminate non-coäccessible states by a simply *trimming* procedure based on graph traversal, finding states which lack a path to an accessible state. However, the procedure to coalesce indistinguishable states is more subtle, and we discuss it in Section 4.2.

4.2 DFA Simplification

The goal of simplification is to coalesce indistinguishable states such as states 6 and 7 in Figure 11, to result in the DFA in Figure 13.

In order to give a good explanation of the simplification algorithm we need some notation. Let $\mathbb{S}$ denote the set of states of the DFA, $\mathbb{S} = \{0, 1, 2, 4, 5, 6, 7\}$. Let Υ denote the set of all Common Lisp types annotated in the DFA: $\Upsilon = \{T_1, T_3, T_4, T_6, T_7\}$. Denote the state transfer function, δ, which given a state, $s_i \in \mathbb{S}$, and a type $v \in \Upsilon$, returns the target state, $s_j \in \mathbb{S}$ of the transition $s_i \xrightarrow{v} s_j$. The values of δ are given in Figure 12 (top left).

We will construct a sequence $\{\Pi_1, \Pi_2, ...\Pi_n, ...\}$ of partitions of $\mathbb{S}$. A *partition* of $\mathbb{S}$ is a set of mutually disjoint subsets of $\mathbb{S}$ for which the union of the subsets is $\mathbb{S}$ itself. Each element $\kappa \in \Pi_k$ is called a *k-equivalence class*. If $s_i, s_j \in \kappa$, then s_i and s_j are said to be *k-equivalent* to each other.

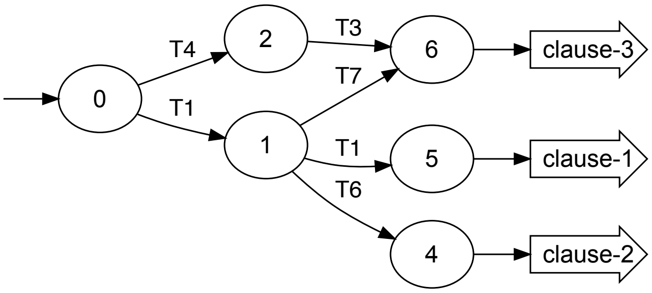
To construct the initial partition, Π_0, we group the set of all non-accepting states into one 0-equivalence class: $\{0, 1, 2\}$; thereafter, there is one 0-equivalence class per unique return value: `:clause-1`, `:clause-2`, and `:clause-3`: $\{5\}$, $\{4\}$, and $\{6, 7\}$ respectively.

$$\Pi_0 = \{\{0, 1, 2\}, \{4\}, \{5\}, \{6, 7\}\}$$

Next, we wish to construct $\Pi_1, \Pi_2, ...\ \Pi_n, \Pi_{n+1}$ in turn, continuing the iteration until $\Pi_n = \Pi_{n+1}$. Each Π_k is derived from Π_{k-1} as we will explain.

For each integer $k > 0$, to determine the k-equivalence classes we define two functions ψ_k and Φ_k.[3] In each case, we will construct

[3] ψ is referred to as the partition transformation function. Φ is referred to as the partition image function.



Label	Type specifier
T_1	`fixnum`
T_3	`number`
T_4	`string`
T_6	`(and (not fixnum) integer)`
T_7	`(and (not integer) number)`

Figure 13: DFA for `rte-case` simplified

ψ_{k+1} and Φ_{k+1} by examining Π_k. These two functions may be difficult to understand intuitively from their mathematical definitions. Nevertheless, the mathematical definitions help when coding the simplification function in Common Lisp.

ψ_{k+1} is a function which takes two arguments, $s \in \mathbb{S}$ and $v \in \Upsilon$, and returns a k-equivalence class $\kappa \in \Pi_k$. (*I.e.*, $\psi_{k+1} : \mathbb{S} \times \Upsilon \to \Pi_k$) To compute the value of $\psi_{k+1}(s, v)$, we select and return the unique $\kappa \in \Pi_k$ for which $\delta(s, v) \in \kappa$. Figure 12 (top right) shows all the values of ψ_1.

Φ_{k+1} takes an element $s \in \mathbb{S}$ and returns a set of order pairs, each of the form (v, κ) where $v \in \Upsilon$ and $\kappa \in \Pi_k$. $\Phi_{k+1}(s)$ is defined as the set of all pairs $(v, \psi_{k+1}(s, v))$, such that $v \in \Upsilon$, and such that $\psi_{k+1}(s, v)$ exists. Figure 12 (bottom) shows all the values of Φ_1.

Now we construct the (k+1)-equivalence classes by splitting the k-equivalence classes; *i.e.* we refine Π_k to construct Π_{k+1}, so that every $\kappa \in \Pi_{k+1}$ contains those elements which have the same value of Φ_{k+1}. This rule implies that if κ has is a singleton set (*e.g.* $\{4\} \in \Pi_0$, and $\{5\} \in \Pi_0$), then $\kappa \in \Pi_{k+1}$ (*i.e.* $\{4\} \in \Pi_1$, and $\{5\} \in \Pi_1$).

Consider the 0-equivalence class $\{0, 1, 2\} \in \Pi_0$. Since $\Phi_1(0)$, $\Phi_1(1)$, and $\Phi_1(2)$ have three different values, then we must further partition $\{0, 1, 2\}$ into three distinct 1-equivalence classes $\{0\}$, $\{1\}$, and $\{2\}$.

Consider the 0-equivalence $\{6, 7\}$. Since $\Phi_1(6) = \Phi_1(7)$, then $\{6, 7\}$ is a 1-equivalence class, and $\{6, 7\} \in \Pi_1$.

$$\Pi_1 = \{\{0\},\ \{1\},\ \{2\},\ \{4\},\ \{5\},\ \{6, 7\}\}$$

If we repeat this process, generating the functions ψ_2 and Φ_2, and use Φ_2 to construct Π_2, we would find that $\Pi_2 = \Pi_1$, which means Π_1 is a fixed point of the procedure.

$$\Pi_2 = \{\{0\},\ \{1\},\ \{2\},\ \{4\},\ \{5\},\ \{6, 7\}\}$$

We can use Π_1, directly, to construct the minimum DFA shown in Figure 13. We simply merge the states which are 1-equivalent. We have determined that states 6 and 7 are 1-equivalent, and no others. We can thus construct the DFA in Figure 13 by merging states 6 and 7 from Figure 11.

```
(let* ((g1 expression)
       (g2 g1))
 (block check
  (tagbody
   s.0
     (unless g1 (return-from check nil))
     (typecase (pop g1)
       (fixnum (go s.2))
       (string (go s.1))
       (t (return-from check nil)))
   s.1
     (unless g1 (return-from check nil))
     (typecase (pop g1)
       (number (go s.3))
       (t (return-from check nil)))
   s.2
     (unless g1 (return-from check nil))
     (typecase (pop g1)
       (fixnum
        (go s.4))
       ((and (not integer) number)
        (go s.3))
       ((and (not fixnum) integer)
        (go s.5))
       (t (return-from check nil)))
   s.3
     (unless g1 (return-from check
       (destructuring-bind (X Y) g2
        (declare (type (or string fixnum) X)
                 (type number Y))
        :clause-3)))
     (case (pop g1)
      (t (return-from check nil)))
   s.4
     (unless g1 (return-from check
       (destructuring-bind (X Y) g2
        (declare (type fixnum X Y))
        :clause-1)))
     (case (pop g1)
      (t (return-from check nil)))
   s.5
     (unless g1 (return-from check
       (destructuring-bind (X Y) g2
        (declare (type fixnum X)
                 (type integer Y))
        :clause-2)))
     (case (pop g1)
      (t (return-from check nil))))))
```

Figure 14: Macro expansion of `rte-case` from Figure 2 and consequently of `destructuring-case` from Figure 1.

5 OPTIMIZED CODE GENERATION

Figure 14 shows the essential part of the final macro expansion of the `rte-case` shown in Figure 2. Each state in the DFA corresponds to a label within a tagbody, a conditional unless checking for end of sequence, and a typecase with one branch per transition in the DFA, including the implicit transition to ⊥. We have used typecase in this example output, but the reader may well notice that there are several occurrences of redundant type checks in the output. For example, the typecase at label s.2 in Figure 14 contains multiple checks for fixnum and integer. We showed in [**?**] how these redundant type checks might be eliminated simply by replacing typecase with bdd-typecase.

6 PREVIOUS WORK

Attempts to implement destructuring-case are numerous. We mention three here. R7RS Scheme provides case-lambda [**?** , Section 4.2.9], allowing fixed length argument lists, but lacking any sort of destructuring; the implementation of destructuring-case provided in [**?**] is missing tree-structure-based clause selection; the implementation provided in [**?**], provides tree-structure-based clause selection, but not within the &optional nor &key portion. In none of these cases does the clause selection consider the types of the objects within the list being destructured.

Manuel and Ramanujam [**?**] introduce automata over infinite alphabets, which seems to be an interesting theoretical approach of viewing DFA whose transitions are Common Lisp types. Manuel and Ramanujam do not investigate questions of construction and simplification as we have investigated in our approach.

6.1 Conclusion and Perspectives

The simplification algorithm described in Section 4.2 may not guarantee a minimum result. For example, reconsider Φ_1 in Figure 12 (bottom). Suppose $T_3 = T' \cup T''$, and suppose there exists $s \in \mathbb{S}$ such that $\Phi(s) = \{(T', \{6, 7\}),\ T'', \{6, 7\}\}$. In such a case, states 2 and s would be indistinguishable, but not mergable with the simplfication algorithm we have described. More research is needed to determine whether such a case can occur, and what the most general form is. Such analysis is necessary to accomplish our goal of generalizing finite automata theory on finite alphabets to handle infinite alphabets representable as disjoinable types.

In the procedure described in Section 4, we constructed the SXP starting with DFAs which were sub-optimal. The DFAs shown in Figure 9 have states which are not coäccessible: states 2.2 and 3.4. Furthermore, one of the DFAs has states 3.2 and 3.5 which are indistinguishable. If we choose to trim and simplify the input DFAs before constructing the SXP there seem to be cases where we reduce the number of state pairs which need to be visited.

A natural question is whether it is better to simplify the input DFAs before computing the SXP, simplify after, or both. One might be tempted to claim that we should always simplify DFAs before computing the SXP. However, we do not currently have enough data to confidently support this claim.

We also discussed in Section 3.3 a technique for making the DFAs match non-overlapping languages before attempting to calculate the SXP. This technique avoids having to make priority based decisions when the languages overlap. We thereafter saw that this technique produces DFAs with non-coäccessible states. It may well be worth investigation whether robustly implementing the priority based SXP procedure is more efficient, as the input DFAs would themselves be smaller in many cases, and be absent the non-coäccessible states.

The rte-case macro we discuss in this paper does not attempt to answer questions about exhaustiveness. It is possible however, to enhance the rte-case macro with rte-ecase (exhaustive rte-case) which would append a final *otherwise* clause, (:* t). This clause would serve at compile time to detect whether the leading clauses are exhaustive; for if no state in the DFA corresponds to this :otherwise-clause, then the given rte patterns are exhaustive. However, if there is a path in the DFA from an initial state to the :otherwise-clause, then the type labels on such a path form a type signature for such a counter example. The types of the elements of such a counter-example sequence could easily be generated by finding any transit through the DFA, and clipping away any loops it contains. The macro might also produce a compiler warning, as well as insert a call to error in the code in case the code path is taken at run-time.