Strategies for typecase optimization

Jim E. Newton
jnewton@lrde.epita.fr

Didier Verna
didier@lrde.epita.fr

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

ABSTRACT

We contrast two approaches to optimizing the Common Lisp `typecase` macro expansion. The first approach is based on heuristics intended to estimate run time performance of certain type checks involving Common Lisp type specifiers. The technique may, depending on code size, exhaustively search the space of permutations of the type checks, intent on finding the optimal order. With the second technique, we represent a `typecase` form as a type specifier, encapsulating the side-effecting non-Boolean parts so as to appear compatible with the Common Lisp type algebra operators. The encapsulated expressions are specially handled so that the Common Lisp type algebra functions preserve them, and we can unwrap them after a process of Boolean reduction into efficient Common Lisp code, maintaining the appropriate side effects but eliminating unnecessary type checks. Both approaches allow us to identify unreachable code, test for exhaustiveness of the clauses and eliminate type checks which are calculated to be redundant.

CCS Concepts

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

1. INTRODUCTION

The `typecase` macro is specified in Common Lisp [4] as is a run-time mechanism for selectively branching as a function of the type of a given expression. Figure 1 summarizes the usage. The type specifiers used may be simple type names such as `fixnum`, `string`, or `my-class`, but may also specify more expressive types such as range checks `(float -3.0 3.5)`, membership checks such as `(member 1 3 5)`, arbitrary Boolean predicate checks such as `(satisfies oddp)`, or logical combinations of other valid type specifiers such as `(or string (and fixnum (not (eql 0))) (cons bignum))`.

```
(typecase keyform
  (Type.1 body-forms-1...)
  (Type.2 body-forms-2...)
  (Type.3 body-forms-3...)
  ...
  (Type.n body-forms-n...))
```

Figure 1: Synopsis of `typecase` syntax.

In this article we consider several issues concerning the compilation of such a `typecase` usage.

- *Redundant checks*[1] — The set of type specifiers used in a particular invocation of `typecase` may have subtype or intersection relations among them. Consequently, it is possible (perhaps likely in the case of auto-generated code) that the same type checks be performed multiple times, when evaluating the `typecase` at run-time.
- *Unreachable code* — The specification *suggests* but does not require that the compiler issue a warning if a clause is not reachable, being completely shadowed by earlier clauses. We consider such compiler warnings desirable, especially in manually written code.
- *Exhaustiveness* — The user is allowed to specify a non-exhaustive set of clauses. If it can be determined at compile time that the clauses are indeed exhaustive, even in the absence of a `t`/`otherwise` clause, then in such a case, the final type check may be safely replaced with `otherwise`, thus eliminating the need for that final type check at run-time.

The `etypecase` macro (exhaustive type case) promises to signal a run-time error if the object is not an element of any of the specified types. The question of whether the clauses are exhaustive is a different question, namely whether it can be determined at compile time that all possible values are covered by at least one of the clauses.

Assuming we are allowed to change the `typecase` evaluation order, we wish to exploit evaluation orders which are more likely to be faster at run-time. We assume that most type checks are fast, but some are slower than others. In particular, a `satisfies` check may be arbitrarily slow. Under certain conditions, as will be seen, there are techniques

[1]Don't confuse *redundant check* with *redundancy check*. In this article we address the former, not the latter. A type check is viewed as *redundant*, and can be eliminated, if its Boolean result can determined by static code analysis.

to *protect* certain type checks to allow reordering without effecting semantics. Such reordering may consequently enable particular optimizations such as elimination of redundant checks or the exhaustiveness optimization explained above. Elimination of redundant type checks has an additional advantage apart from potentially speeding up certain code paths, it also allows the discovery of unreachable code.

In this article we consider different techniques for evaluating the type checks in different orders than that which is specified in the code, so as to maintain the semantics but to eliminate redundant checks.

In the article we examine two very different approaches for performing certain optimizations of `typecase`. First, we use a *natural* approach using s-expression based type specifiers (Section 2), operating on them as symbolic expressions. In the second approach (Section 3) we employ Reduced Ordered Binary Decision Diagrams (ROBDDs). We finish the article with an overview of related work (Section 4) and a summary of future work (Section 5).

2. TYPE SPECIFIER APPROACH

We would like to automatically remove redundant checks such as `(eql 42)`, `(member 40 41 42)`, and `fixnum` in Example 1.

Example 1 (typecase with redundant type checks)**.**

```
(typecase OBJ
  ((eql 42)
   body-forms-1...)
  ((and (member 40 41 42) (not (eql 42)))
   body-forms-2...)
  ((and fixnum (not (member 40 41 42)))
   body-forms-3...)
  ((and number (not fixnum))
   body-forms-4...))
```

The code in Example 2 is semantically identical to that in Example 1, because a type check is only reached if all preceding type checks have failed.

Example 2 (typecase after removing redundant checks)**.**

```
(typecase OBJ
  ((eql 42) body-forms-1...)
  ((member 40 41 42) body-forms-2...)
  (fixnum body-forms-3...)
  (number body-forms-4...))
```

In the following sections, we initially show that certain duplicate checks may be removed through a technique called forward-substitution and reduction (Section 2.2). A weakness of this technique is that it sometimes fails to remove particular redundant type checks. Because of this weakness, a more elaborate technique is applied, in which we augment the type tests to make them mutually disjoint (Section 2.4). With these more complex type specifiers in place, the `typecase` has the property that its clauses are reorderable, which allows the forward-substitution and reduction algorithm to search for an ordering permitting more thorough reduction (Section 2.6). This process allows us to identify unreachable code paths and to identify exhaustive case analyses, but there are still situations in which redundant checks cannot be eliminated.

2.1 Semantics of type specifiers

There is some disagreement among experts of how to interpret certain semantics of type specifiers in Common Lisp. To avoid confusion, we state explicitly our interpretation.

There is a statement in the `typecase` specification that each *normal-clause* be *considered* in turn. We interpret this requirement not to mean that the type checks must be evaluated in order, but rather than each type test must assume that type tests appearing earlier in the `typecase` are not satisfied. Moreover, we interpret this specified requirement so as not to impose a run-time evaluation order, and that as long as evaluation semantics are preserved, then the type checks may be done in any order at run-time, and in particular, that any type check which is redundant or unnecessary need not be preformed.

The situation that the user may specify a type such as `(and fixnum (satisfies evenp))` is particularly problematic, because the Common Lisp specification contains a dubious, non-conforming example in the specification of `satisfies`. The problematic example in the specification says that `(and integer (satisfies evenp))` is a type specifier and denotes the set of all even integers. This claim contradicts the specification of the `AND` type specifier which claims that `(and A B)` is the intersection of types `A` and `B` and is thus the same as `(and B A)`. This presents a problem, because `(typep 1.0 '(and fixnum (satisfies evenp)))` evaluates to `nil` while `(typep 1.0 '(and (satisfies evenp) fixnum))` raises an error. We implicitly assume, for optimization purposes, that `(and A B)` is the same as `(and B A)`.

Specifically, if the `AND` and `OR` types are commutative with respect to their operands, and if type checks have side effects (errors, conditions, changing of global state, IO, interaction with the debugger), then the side effects cannot be guaranteed when evaluating the optimized code. Therefore, in our treatment of types we consider that type checking with `typep` is side-effect free, and in particular that it never raises an error. This assumption allows us to reorder the checks as long as we do not change the semantics of the Boolean algebra of the `AND`, `OR`, and `NOT` specifiers.

Admittedly, that `typep` never raise an error is an assumption we make knowing that it may limit the usefulness of our results, especially since some non-conforming Common Lisp programs may happen to perform correctly absent our optimizations. That is to say, our optimizations may result in errors in some non-conforming Common Lisp programs. The specification clearly states that certain run-time calls to `typep` even with well-formed type specifiers must raise an error, such as if the type specifier is a list whose first element is `values` or `function`. Also, as mentioned above, an evaluation of `(typep obj '(satisfies F))` will raise an error if `(F obj)` raises an error. One might be tempted to interpret `(typep obj '(satisfies F))` as `(ignore-errors (if (F obj) t nil))`, but that would be a violation of the specification which is explicit that the form `(typep x '(satisfies p))` is equivalent to `(if (p x) t nil)`.

There is some wiggle room, however. The specification of `satisfies` states that its operand be the name of a *predicate*, which is elsewhere defined as a function which returns. Thus one might be safe to conclude that `(satisfies evenp)` is not a valid type specifier, because `evenp` is specified to signal an error if its argument is not an `integer`.

We assume, for this article, that no such problematic type specifier is used in the context of `typecase`.

2.2 Reduction of type specifiers

There are legitimate cases in which the programmer has specifically ordered the clauses to optimize performance. A

production worthy `typecase` optimization system should take that into account. However, for the sake of simplicity, the remainder of this article ignores this concern.

We introduce a macro, `reduced-typecase`, which expands to a call to `typecase` but with cases reduced where possible. Latter cases assuming previous type checks fail. This transformation preserves clause order, but may simplify the executable logic of some clauses. In the expansion, in Example 3 the second `float` check is eliminated, and consequently, the associated `AND` and `NOT`.

Example 3 (Simple invocation and expansion of `reduced-typecase`)**.**

```
(reduced-typecase obj
   (float body-forms-1...)
   ((and number (not float)) body-forms-2...))

(typecase obj
   (float body-forms-1...)
   (number body-forms-2...))
```

How does this reduction work? To illustrate we provide a sightly more elaborate example. In Example 4 the first type check is `(not (and number (not float)))`. In order that the second clause be reached at run-time the first type check must have already failed. This means that the second type check, `(or float string (not number))`, may assume that `obj` is not of type `(not (and number (not float)))`.

Example 4 (Invocation and expansion `reduced-typecase` with unreachable code path)**.**

```
(reduced-typecase obj
   ((not (and number (not float))) body-forms-1...)
   ((or float string (not number)) body-forms-2...)
   (string body-forms-3...))

(typecase obj
   ((not (and number (not float))) body-forms-1...)
   (string body-forms-2...)
   (nil body-forms-3...))
```

The `reduced-typecase` macro rewrites the second type test `(or float string (not number))` by a technique called forward-substitution. At each step, it substitutes implied values into the next type specifier, and performs Boolean logic reduction. Abelson *et al.* [1] discuss lisp[2] algorithms for performing algebraic reduction; however, in addition to the Abelson algorithm reducing Boolean expressions representing Common Lisp types involves additional reductions representing the subtype relations of terms in question. For example `(and number fixnum ...)` reduces to `(and fixnum ...)` because `fixnum` is a subtype of `number`. Similarly, `(or number fixnum ...)` reduces to `(or number ...)`. Newton *et al.* [21] discuss techniques of Common Lisp type reduction in the presence of subtypes.

[2]In this article we use *lisp* (in lower case) to denote the family of languages or the concept rather than a particular language implementation, and we use *Common Lisp* to denote the language.

```
(not (and number (not float))) = nil
  ⟹ (and number (not float)) = t
                    ⟹ number = t
                and (not float) = t
                     ⟹ float = nil
               (or float string
                  (not number)) = (or nil string (not t))
                                = (or nil string nil)
                                = string
```

With this forward substitution, `reduced-typecase` is able to rewrite the second clause `((or float string (not number)) body-forms-2...)` simply as `(string body-forms-2...)`. Thereafter, a similar forward substitution is made to transform the third clause from `(string body-forms-3...)` to `(nil body-forms-3...)`.

Example 4 illustrates a situation in which a type specifier in one of the clauses reduces completely to `nil`. In such a case we would like the compiler to issue warnings about finding unreachable code, and in fact it does (at least when tested with SBCL[3]) because the compiler finds `nil` as the type specifier. The clauses in Example 5 are identical to those in Example 4, and consequently the expressions `body-forms-3...` in the third clause cannot be reached. Yet contrary to Example 4, SBCL, AllegroCL[4], and CLISP[5] issue no warning at all that `body-forms-3...` is unreachable code.

Example 5 (Invocation of `typecase` with unreachable code)**.**

```
(typecase obj
   ((not (and number (not float))) body-forms-1...)
   ((or float string (not number)) body-forms-2...)
   (string body-forms-3...))
```

2.3 Order dependency

We now reconsider Examples 1 and 2. While the semantics are the same, there is an important distinction in practice. The first `typecase` contains mutually exclusive clauses, whereas the second one does not. *E.g.*, if the `(member 40 41 42)` check is moved before the `(eql 42)` check, then `(eql 42)` will never match, and the consequent code, `body-forms-2...` will be unreachable.

For the order of the type specifiers given Example 1, the types can be simplified, having no redundant type checks, as shown in Example 2. This phenomenon is both a consequence of the particular types in question and also the order in which they occur. As a contrasting example, consider the situation in Example 6 where the first two clauses of the `typecase` are reversed with respect to Example 1. In this case knowing that `OBJ` is not of type `(and (member 40 41 42) (not (eql 42)))` tells us nothing about whether `OBJ` is of type `(eql 42)` so no reduction can be inferred.

[3]We tested with SBCL 1.3.14. SBCL is an implementation of ANSI Common Lisp. `http://www.sbcl.org/`

[4]We tested with the International Allegro CL Free Express Edition, version 10.1 [32-bit Mac OS X (Intel)] (Sep 18, 2017 13:53). `http://franz.com`

[5]We tested with GNU CLISP 2.49, (2010-07-07). `http://clisp.cons.org/`

Example 6 (Re-ordering clauses sometimes enable reduction)**.**

```
(typecase OBJ
  ((and (member 40 41 42) (not (eql 42)))
   body-forms-2...)
  ((eql 42)
   body-forms-1...)
  ((and fixnum (not (member 40 41 42)))
   body-forms-3...)
  ((and number (not fixnum))
   body-forms-4...))
```

Programmatic reductions in the `typecase` are dependent on the order of the specified types. There are many possible approaches to reducing types despite the order in which they are specified. We consider two such approaches. Section 2.6 discusses automatic reordering of disjoint clauses, and Section 3 uses decision diagrams.

As already suggested, a situation as shown in Example 6 can be solved to avoid the redundant type check, `(eql 42)`, by reordering the disjoint clauses as in Example 1. However, there are situations for which no reordering alleviates the problem. Consider the code shown in Example 7. We see that some sets of types are reorderable, allowing reduction, but for some sets of types such ordering is impossible. We consider in Section 3 `typecase` optimization where reordering is futile. For now we concentrate on efficient reordering where possible.

Example 7 (Re-ordering cannot always enable reduction)**.**

```
(typecase OBJ
  ((and unsigned-byte (not bignum))
   body-forms-1...)
  ((and bignum (not unsigned-byte))
   body-forms-2...))
```

2.4 Mutually disjoint clauses

As suggested in Section 2.3, to arbitrarily reorder the clauses, the types must be disjoint. It is straightforward to transform any `typecase` into another which preserves the semantics but for which the clauses are reorderable. Consider a `typecase` in a general form.

Example 8 shows a set of type checks equivalent to those in Figure 1 but with redundant checks, making the clauses mutually exclusive, and thus reorderable.

Example 8 (typecase with mutually exclusive type checks)**.**

```
(typecase OBJ
  (Type.1
   body-forms-1...)
  ((and Type.2
        (not Type.1))
   body-forms-2...)
  ((and Type.3
        (not (or Type.1 Type.2)))
   body-forms-3...))
  ...
  ((and Type.n
        (not (or Type.1 Type.2 ... Type.n-1)))
   body-forms-n...))
```

In order to make the clauses reorderable, we make them more complex which might seem to defeat the purpose of optimization. However, as we see in Section 2.6, the complexity can sometimes be removed after reordering, thus resulting in a set of type checks which is *better* than the original. We discuss what we mean by *better* in Section 2.5.

We proceed by first describing a way to judge which of two orders is better, and with that comparison function, we can visit every permutation and choose the best.

One might also wonder why we suffer the pain of establishing heuristics and visiting all permutations of the mutually disjoint types in order to find the best order. One might ask, why not just put the clauses in the best order to begin with. The reason is because in the general case, it is not possible to predict what the best order is. As is discussed in Section 4, ordering the Boolean variables to produce the smallest binary decision diagram is an NP-hard problem. The only solution in general is to visit every permutation. The problem of ordering a set of type tests for optimal reduction must also be NP-hard because if we had a better solution, we would be able to solve the BDD NP-hard problem as a consequence.

2.5 Comparing heuristically

Given a set of disjoint and thus reorderable clauses, we can now consider finding a good order. We can examine a type specifier, typically after having been reduced, and heuristically assign a *cost*. A high cost is assigned to a `satisfies` type, a medium cost to `AND`, `OR`, and `NOT` types which takes into account the cost of the types specified therein, and a low cost to atomic names.

To estimate the relative *goodness* of two given semantically identical `typecase` invocations, we can heuristically estimate the complexity of each by using a weighted sum of the costs of the individual clauses. The weight of the first clause is higher because the type specified therein will be always checked. Each type specifier thereafter will only be checked if all the preceding checks fail. Thus the heuristic weights assigned to subsequent checks is chosen successively smaller as each subsequent check has a smaller probability of being reached at run-time.

2.6 Reduction with automatic reordering

Now that we have a way to heuristically measure the complexity of a given invocation of `typecase` we can therewith compare two semantically equivalent invocations and choose the better one. If the number of clauses is small enough, we can visit all possible permutations. If the number of clauses is large, we can sample the space randomly for some specified amount of time or specified number of samples, and choose the best ordering we find.

We introduce the macro, `auto-permute-typecase`. It accepts the same arguments as `typecase` and expands to a `typecase` form. It does so by transforming the specified types into mutually disjoint types as explained in Section 2.4, then iterating through all permutations of the clauses. For each permutation of the clauses, it reduces the types, eliminating redundant checks where possible using forward-substitution as explained in Section 2.2, and assigns a cost heuristic to each permutation as explained in Section 2.5. The `auto-permute-typecase` macro then expands to the `typecase` form with the clauses in the order which minimizes the heuristic cost.

Example 9 shows an invocation and expansion of `auto-permute-typecase`. In this example `auto-permute-typecase` does a good job of eliminating redundant type checks.

Example 9 (Invocation and expansion of `auto-permute-typecase`)**.**

```
(auto-permute-typecase obj
  ((and unsigned-byte (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...))

(typecase obj
  ((eql 42) body-forms-2...)
  (unsigned-byte body-forms-1...)
  (fixnum body-forms-4...)
  (number body-forms-3...))
```

As mentioned in Section 1, a particular optimization can be made in the situation where the type checks in the `typecase` are exhaustive; in particular the final type check may be replaced with `t`/`otherwise`. Example 10 illustrates such an expansion in the case that the types are exhaustive. Notice that the final type test in the expansion is `t`.

Example 10 (Invocation and expansion of `auto-permute-typecase` with exhaustive type checks)**.**

```
(auto-permute-typecase OBJ
  ((or bignum unsigned-byte) body-forms-1...)
  (string body-forms-2...)
  (fixnum body-forms-3...)
  ((or (not string) (not number)) body-forms-4...))

(typecase obj
  (string body-forms-2...)
  ((or bignum unsigned-byte) body-forms-1...)
  (fixnum body-forms-3...)
  (t body-forms-4...))
```

3. DECISION DIAGRAM APPROACH

In Section 2.6 we looked at a technique for reducing `typecase` based solely on programmatic manipulation of type specifiers. Now we explore a different technique based on a data structure known as Reduced Ordered Binary Decision Diagram (ROBDD).

Example 7 illustrates that redundant type checks cannot always be reduced via reordering. Example 11 is, however, semantically equivalent to Example 7. Successfully mapping the code from of a `typecase` to an ROBDD will guarantee that redundant type checks are eliminated. In the following sections we automate this code transformation.

Example 11 (Suggested expansion of Example 7)**.**

```
(if (typep OBJ 'unsigned-byte)
    (if (typep obj 'bignum)
        nil
        (progn body-forms-1...))
    (if (typep obj 'bignum)
        (progn body-forms-2...)
        nil))
```

The code in Example 11 also illustrates a concern of code size explosion. With the two type checks `(typep OBJ 'unsigned-byte)` and `(typep obj 'bignum)`, the code expands to 7 lines of code. If this code transform be done naïvely, the risk is that each `if`/`then`/`else` effectively doubles the code size. In such an undesirable case, a `typecase` having N unique type tests among its clauses, would expand to $2^{N+1} - 1$ lines of code, even if such code has many congruent code paths. The use of ROBDD related techniques allows us to limit the code size to something much more manageable. Some discussion of this is presented in Section 4.

ROBDDs (Section 3.1) represent the semantics of Boolean equations but do not maintain the original evaluation order encoded in the actual code. In this sense the reordering of the type checks, which is explicit and of combinatorical complexity in the previous approach, is automatic in this approach. A complication is that normally ROBDDs express Boolean functions, so the mapping from `typecase` to ROBDD is not immediate, as a `typecase` may contain arbitrary side-effecting expressions which are not restricted to Boolean expressions. We employ an encapsulation technique which allows the ROBDDs to operate opaquely on these problematic expressions (Section 3.1). Finally, we are able to serialize an arbitrary `typecase` invocation into an efficient `if`/`then`/`else` tree (Section 3.3).

ROBDDs inherently eliminate duplicate checks. However, ROBDDs cannot easily guarantee removing all unnecessary checks as that would involve visiting every possible ordering of the leaf level types involved.

3.1 An ROBDD compatible type specifier

An ROBDD is a data structure used for performing many types of operations related to Boolean algebra. When we use the term ROBDD we mean, as the name implies, a decision diagram (directed cyclic graph, DAG) whose vertices represent Boolean tests and whose branches represent the consequence and alternative actions. An ROBDD has its variables **O**rdered, meaning that there is some ordering of the variables $\{v_1, v_2, ..., v_N\}$ such that whenever there is an arrow from v_i to v_j then $i < j$. An ROBDD is deterministically **R**educed so that all common sub-graphs are shared rather than duplicated. The reader is advised to read the lecture nodes of Andersen [3] for a detailed understanding of the reduction rules. It is worth noting that there is variation in the terminology used by different authors. For example, Knuth [18] uses the unadorned term BDD for what we are calling an ROBDD.

A unique ROBDD is associated with a canonical form representing a Boolean function, or otherwise stated, with an equivalence class of expressions within the Boolean algebra. In particular, intersection, union, and complement operations as well as subset and equivalence calculations on elements from the underlying space of sets or types can be computed by straightforward algorithms. We omit detailed explanations of those algorithms here, but instead we refer the reader to work by Andersen [3] and Castagna [12].

We employ ROBDDs to convert a `typecase` into an `if`/`then`/`else` diagram as shown in Figure 2. In the figure, we see a decision diagram which is similar to an ROBDD, at least in all the internal nodes of the diagram. Green arrows lead to the consequent if a specified type check succeeds. Red arrows lead to the alternative. However, the leaf nodes are not Boolean values as we expect for an ROBDD.

We want to transform the clauses of a `typecase` as shown in Figure 1 into a binary decision diagram. To do so, we associate a distinct `satisfies` type with each clause of the `typecase`. Each such `satisfies` type has a unique function associated with it, such as `P1`, `P2`, etc, allowing us to represent the diagram shown in Figure 2 as an actual ROBDD as shown in Figure 3.

In order for certain Common Lisp functions to behave properly (such as `subtypep`) the functions `P1`, `P2`, etc. must be real functions, as opposed to place-holder functions types as Baker[7] suggests, so that `(satisfies P1)` etc, have type

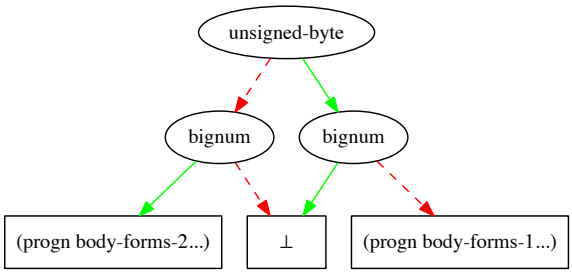

Figure 2: Decision Diagram representing irreducible `typecase`. This is similar to an ROBDD, but does not fulfill the definition thereof, because the leaf nodes are not simple Boolean values.

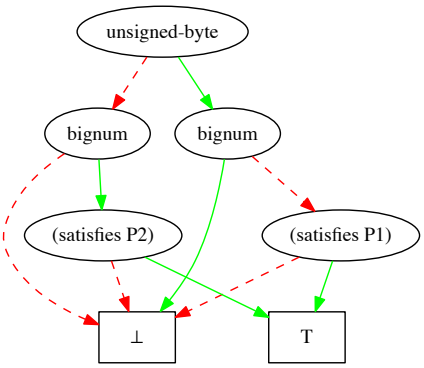

Figure 3: ROBDD with temporary valid `satisfies` types

specifier semantics. `P1`, `P2`, etc, must be defined in a way which preserves the semantics of the `typecase`.

Ideally we would like to create type specifiers such as the following:

```
(satisfies (lambda (obj)
             (typep obj '(and (not unsigned-byte)
                              bignum))))
```

Unfortunately, the specification of `satisfies` explicitly forbids this, and requires that the operand of `satisfies` be a symbol representing a globally callable function, even if the type specifier is only used in a particular dynamic extent. Because of this limitation in Common Lisp, we create the type specifiers as follows. Given a type specifier, we create such a functions at run-time using the technique shown in the function `define-type-predicate` defined in Implementation 1, which programmatically defines function with semantics similar to those shown in Example 12.

Implementation 1 (define-type-predicate)**.**

```
(defun define-type-predicate (type-specifier)
  (let ((function-name (gensym "P")))
    (setf (symbol-function function-name)
          #'(lambda (obj)
              (typep obj type-specifier)))
    function-name))
```

Example 12 (Semantics of `satisfies` predicates)**.**

```
(defun P1 (obj)
  (typep obj '(and (not unsigned-byte) bignum)))
(defun P2 (obj)
  (typep obj '(and (not bignum) unsigned-byte)))
```

The `define-type-predicate` function returns the name of a named closure which the calling function can use to construct a type specifier. The name and function binding are generated in a way which has dynamic extent and is thus friendly with the garbage collector.

To generate the ROBDD shown in Figure 3 we must construct a type specifier equivalent to the entire invocation of `typecase`. From the code in Figure 1 we have to assemble a type specifier such as in Example 13. This example is provided simply to illustrate the pattern of such a type specifier.

Example 13 (Type specifier equivalent to Figure 1)**.**

```
(let ((P1 (define-type-predicate 'Type.1))
      (P2 (define-type-predicate
            '(and Type.2 (not Type.1))))
      (P3 (define-type-predicate
            '(and Type.3 (not (or Type.1 Type.2)))))
      ...
      (Pn (define-type-predicate
           '(and Type.n (not (or Type.1 Type.2
                                 ... Type.n-1))))))
  `(or (and Type.1
            (satisfies ,P1))
       (and Type.2
            (not Type.1)
            (satisfies ,P2))
       (and Type.3
            (not (or Type.1 Type.2))
            (satisfies ,P3))
       ...
       (and Type.n
            (not (or Type.1 Type.2
                     ... Type.n-1))
            (satisfies ,Pn))))
```

3.2 BDD construction from type specifier

Functions which construct an ROBDD need to understand a complete, deterministic ordering of the set of type specifiers via a *compare* function. To maintain semantic correctness the corresponding *compare* function must be deterministic. It would be ideal if the function were able to give high priority to type specifiers which are likely to be seen at run time. We might consider, for example, taking clues from the order specified in the `typecase` clauses. We do not attempt to implement such decision making. Rather we choose to give high priority to type specifiers which are *easy* to check at run-time, even if they are less likely to occur.

We use a heuristic similar to that mentioned in Section 2.5 except that type specifiers involving `AND`, `OR`, and `NOT` never occur, rather such types correspond to algebraic operations among the ROBDDs themselves such that only non-algebraic types remain. More precisely, the heuristic we use is that atomic types such as `number` are considered fast to check, and `satisfies` types are considered slow. We recognize the limitation that the user might have used `deftype` to define a type whose name is an atom, but which is slow to type check. Ideally, we should fully *expand* user defined types into Common Lisp types. Unfortunately this is not possible in a portable way, and we make no attempts to implement such expansion in implementation specific ways. It is not even clear whether the various Common Lisp implementations have public APIs for the operations necessary.

A crucial exception in our heuristic estimation algorithm is that to maintain the correctness of our technique, we must assure that the `satisfies` predicates emanating from `define-type-predicate` have the lowest possible priority. *I.e.*, as is shown in Figure 3, we must avoid that any type check appear below such a `satisfies` type in the ROBDD.

There are well known techniques for converting an ROBDD which represents a pure Boolean expression into an `if`/`then`/`else` expression which evaluates to `true` or `false`. However, in our case we are interested in more than simply the Boolean value. In particular, we require that the resulting expression evaluate to the same value as corresponding `typecase`. In Figure 1, these are the values returned from `body-forms-1...`, `body-forms-2...`, ... `body-forms-n...`. In addition we want to assure that any side effects of those expressions are realized as well when appropriate, and never realized more than once.

We introduce the macro `bdd-typecase` which expands to a `typecase` form using the ROBDD technique. When the macro invocation in Example 14 is expanded, the list of `typecase` clauses is converted to a type specifier similar to what is illustrated in Example 13. That type specifier is used to create an ROBDD as illustrated in Figure 4. As shown in the figure, temporary `satisfies` type predicates are created corresponding to the potentially side-effecting expressions `body-forms-1`, `body-forms-2`, `body-forms-3`, and `body-forms-4`. In reality these temporary predicates are named by machine generated symbols; however, in Figure 4 they are denoted `P1`, `P2`, `P3`, and `P4`.

Example 14 (Invocation of `bdd-typecase` with intersecting types)**.**

```
(bdd-typecase obj
  ((and unsigned-byte (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...))
```

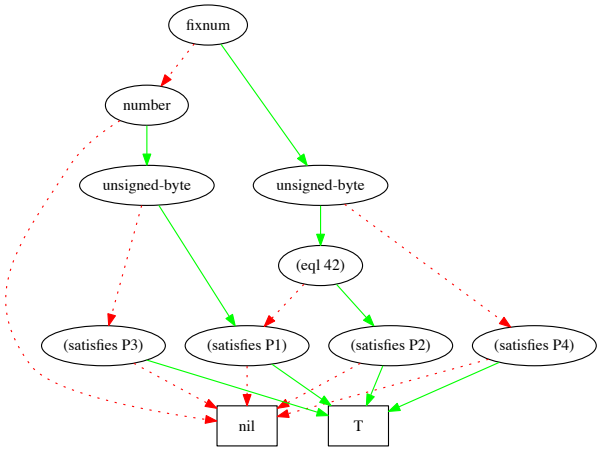

Figure 4: ROBDD generated from `typecase` clauses in Example 14

3.3 Serializing the BDD into code

The macro `bdd-typecase` emits code as in Example 15, but just as easily may output code as in Example 16 based on `tagbody`/`go`. In both example expansions we have substituted more readable labels such as `L1` and `block-1` rather than the more cryptic machine generated uninterned symbols `#:l1070` and `#:|block1066|`.

Example 15 (Macro expansion of Example 14 using `labels`)**.**

```
((lambda (obj-1)
   (labels ((L1 () (if (typep obj-1 'fixnum)
                       (L2)
                       (L7)))
            (L2 () (if (typep obj-1 'unsigned-byte)
                       (L3)
                       (L6)))
            (L3 () (if (typep obj-1 '(eql 42))
                       (L4)
                       (L5)))
            (L4 () body-forms-2...)
            (L5 () body-forms-1...)
            (L6 () body-forms-4...)
            (L7 () (if (typep obj-1 'number)
                       (L8)
                       nil))
            (L8 () (if (typep obj-1 'unsigned-byte)
                       (L5)
                       (L9)))
            (L9 () body-forms-3...))
     (L1)))
 obj)
```

The `bdd-typecase` macro walks the ROBDD, such as the one illustrated in Figure 4, visiting each non-leaf node therein. Each node corresponding to a named closure type predicate is serialized as a tail call to the clauses from the `typecase`. Each node corresponding to a normal type test is serialized as left and right branches, either as a label and two calls to `go` as in Example 16, or a local function definition with two tail calls to other local functions as in Example 15.

Example 16 (Alternate expansion of Example 14 using `tagbody/go`).

```
((lambda (obj-1)
   (block block-1
     (tagbody
      L1 (if (typep obj-1 'fixnum)
             (go L2)
             (go L7))
      L2 (if (typep obj-1 'unsigned-byte)
             (go L3)
             (go L6))
      L3 (if (typep obj-1 '(eql 42))
             (go L4)
             (go L5))
      L4 (return-from block-1
           (progn body-forms-2...))
      L5 (return-from block-1
           (progn body-forms-1...))
      L6 (return-from block-1
           (progn body-forms-4...))
      L7 (if (typep obj-1 'number)
             (go L8)
             (return-from block-1 nil))
      L8 (if (typep obj-1 'unsigned-byte)
             (go L5)
             (go L9))
      L9 (return-from block-1
           (progn body-forms-3...)))))
 obj)
```

3.4 Emitting compiler warnings

The ROBDD, as shown in Figure 4, can be used to generate the Common Lisp code semantically equivalent to the corresponding `typecase` as already explained in Section 3.3, but we can do even better. There are two situations where we might wish to emit warnings: (1) if certain code is unreachable, and (2) if the clauses are not exhaustive. Unfortunately, there is no standard way to incorporate these warnings into the standard compiler output. One might tempted to simply emit a warning of type `style-warning` as is suggested by the `typecase` specification. However, this would be undesirable since there is no guarantee that the corresponding code was human-generated—ideally we would only like to see such style warnings corresponding to human generated code.

The list of unreachable clauses can be easily calculated as a function of which of the `P1`, `P2` ... predicates are missing from the serialized output. As seen in Figure 4, each of `body-forms-1`, `body-forms-2`, `body-forms-3`, and `body-forms-4` is represented as `P1`, `P2`, `P3`, and `P4`, so no such code is unreachable in this case.

We also see in Figure 4 that there is a path from the root node to the `nil` leaf node which does not pass through `P1`, `P2`, `P3`, or `P4`. This means that the original `typecase` is not exhaustive. The type of any such value can be calculated as the particular path leading to `nil`. In the case of Figure 4, `(and (not fixnum) (not number))`, which corresponds simply to `(not number)`, is such a type. *I.e.*, the original `bdd-typecase`, shown in Example 14, does not have a clause for non numbers.

4. RELATED WORK

This article references the functions `make-bdd` and `bdd-cmp` whose implementation is not shown herein. The code is available via GitLab from the EPITA/LRDE public web page. `https://gitlab.lrde.epita.fr/jnewton/regular-type-expression.git`. That repository contains several things. Most interesting for the context of BDDs is the Common Lisp package, `LISP-TYPES`.

As there are many individual styles of programming, and each programmer of Common Lisp adopts his own style, it is unknown how widespread the use of `typecase` is in practice, and consequently whether optimizing it is effort well spent. A casual look at the code in the current public Quicklisp[6] repository reveals a rule of thumb. 1 out of 100 files, and 1 out of 1000 lines of code use or make reference to `typecase`. When looking at the Common Lisp code of SBCL itself, we found about 1.6 uses of `typecase` per 1000 lines of code. We have made no attempt to determine which of the occurrences are comments, trivial uses, or test cases, and which ones are used in critical execution paths; however, we do loosely interpret these results to suggest that an optimized `typecase` either built into the `cl:typecase` or as an auxiliary macro may be of little use to most currently maintained projects. On the contrary, we suggest that having such an optimized `typecase` implementation, may serve as motivation to some programmers to make use of it, at least in machine generated code such as Newton *et al.* [20] explain. Since generic function dispatch conceptually bases branching choices on Boolean combinations of type checks, one naturally wonders whether our optimizations might be of useful within the implementation of CLOS[17].

Newton *et al.* [20] present a mechanism to characterize the type of an arbitrary sequence in Common Lisp in terms of a rational language of the types of the sequence elements. The article explains how to build a finite state machine and from that construct Common Lisp code for recognizing such a sequence. The code associates the set of transitions existing from each state as a `typecase`. The article notes that such a machine generated `typecase` could greatly benefit from an optimizing `typecase`.

The `map-permutations` function (Section 2.6) works well for small lists, but requires a large amount of stack space to visit all the permutations of large lists. Knuth[18] explores several iterative (not recursive) algorithms using various techniques, in particular by plain changes[18, Algorithm P, page 42], by cyclic shifts[18, Algorithm C, page 56], and by Erlich swaps[18, Algorithm E, page 57]. A survey of these three algorithms can also be found in the Cadence SKILL Blog[7] which discussions an implementation in SKILL[8], another lisp dialect.

There is a large amount of literature about Binary Decision Diagrams of many varieties [9, 10, 2, 14, 3]. In particular Knuth [18, Section 7.1.4] discusses worst-case and average sizes, which we alluded to in Section 3. Newton *et al.* [21] discuss how the Reduced Ordered Binary Decision Diagram (ROBDD) can be used to manipulate type specifiers, especially in the presence of subtypes. Castagna [12] discusses the use of ROBDDs (he calls them BDDs in that article) to perform type algebra in type systems which treat types as sets [16, 13, 4].

BDDs have been used in electronic circuit generation[15], verification, symbolic model checking[11], and type system models such as in XDuce [16]. None of these sources discusses how to extend the BDD representation to support subtypes.

Common Lisp does not provide explicit pattern matching [5] capabilities, although several systems have been proposed

[6] `https://www.quicklisp.org/`

[7] `https://community.cadence.com/tags/Team-SKILL`, SKILL for the Skilled, *Visiting all Permutations*

such as Optima[8] and Trivia[9]. Pierce [23, 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.

Decision tree techniques are useful in the efficient compilation of pattern matching constructs in functional languages[19]. An important concern in pattern matching compilation is finding the best ordering of the variables which is known to be NP-hard. 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.

In Section 2.2 we mentioned the problem of symbolic algebraic manipulation and simplification. Ableson *et al.* [1, Section 2.4.3] discuss this with an implementation of rational polynomials. Norvig [22, Chapter 8] discusses this in a use case of a symbolic mathematics simplification program. Both the Ableson and Norvig studies explicitly target a lisp-literate audience.

5. CONCLUSION AND FUTURE WORK

As illustrated in Example 9, the exhaustive search approach used in the `auto-permute-typecase` (Section 2.6) can often do a good job removing redundant type checks occurring in a `typecase` invocation. Unfortunately, as shown in Example 7, sometimes such optimization is algebraically impossible because the particular type interdependencies. In addition, an exhaustive search becomes unreasonable when the number of clauses is large. In particular there are $N!$ ways to order N clauses. This means there are $7! = 5040$ orderings of 7 clauses and $10! = 3,628,800$ orderings of 10 clauses.

On the other hand, the `bdd-typecase` macro, using the ROBDD approach (Section 3.2), is always able to remove duplicate checks, guaranteeing that no type check is performed twice. Nevertheless, it may fail to eliminate some unnecessary checks which need not be performed at all.

It is known that the size and *shape* of a reduced BDD depends on the ordering chosen for the variables [9]. Furthermore, it is known that finding the *best* ordering is NP-hard, and in this article we do not address questions of choosing or improving variable orderings. It would be feasible, at least in some cases, to apply the exhaustive search approach with ROBDDs. *I.e.*, we could visit all orders of the type checks to find which gives the smallest ROBDD. In situations where the number of different type tests is large, the development described in Section 3.1 might very well be improved employing some known techniques for improving BDD size though variable ordering choices[6]. In particular, we might attempt to use the order specified in the `typecase` as input to the sorting function, attempting in at least the simple cases to respect the user given order as much as possible.

In Section 2.4, we presented an approach to approximating the cost a set of type tests and commented that the heuristics are simplistic. We leave it as a matter for future research as to how to construct good heuristics, which take into account how compute intensive certain type specifiers are to manipulate.

We believe this research may be useful for two target audiences: application programmers and compiler developers. Even though the currently observed use frequency of `typecase` seems low in the majority of currently supported applications, programmers may find the macros explained in this article (`auto-permute-typecase` and `bdd-typecase`) to be useful in rare optimization cases, but more often for their ability to detect certain dubious code paths. There are, however, limitations to the portable implementation, namely the lack of a portable expander for user defined types, and an ability to distinguish between machine generated and human generated code. These shortcomings may not be significant limitations to the compiler implementer, in which case the compiler may be able to better optimize user types, implement better heuristics regarding costs of certain type checks, and emit useful warnings about unreachable code.

[8] `https://github.com/m2ym/optima`

[9] `https://github.com/guicho271828/trivia`

6. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

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

[3] H. R. Andersen. An introduction to binary decision diagrams. Technical report, Course Notes on the WWW, 1999.

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

[5] L. 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.

[6] A. Aziz, S. Taşiran, and R. K. Brayton. Bdd variable ordering for interacting finite state machines. In *Proceedings of the 31st Annual Design Automation Conference*, DAC '94, pages 283–288, New York, NY, USA, 1994. ACM.

[7] H. G. Baker. A Decision Procedure for Common Lisp's SUBTYPEP Predicate. *Lisp and Symbolic Computation*, 5(3):157–190, 1992.

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

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

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

[11] 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.

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

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

[14] M. 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, Dec. 2013.

[15] O. Coudert, C. Berthet, and J. C. 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.

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

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

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

[19] L. 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.

[20] J. Newton, A. Demaille, and D. Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In *European Lisp Symposium*, Kraków, Poland, May 2016.

[21] J. Newton, D. Verna, and M. Colange. Programmatic manipulation of Common Lisp type specifiers. In *European Lisp Symposium*, Brussels, Belgium, Apr. 2017.

[22] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.

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