Removing redundant tests by replicating control paths

Irène Durand
Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr

ABSTRACT

We describe a technique for removing redundant tests in intermediate code by replicating the control paths between two identical tests, the second of which is dominated by the first. The two replicas encode different outcomes of the test, making it possible to remove the second of the two. Our technique uses local graph rewriting, making its correctness easy to prove. We also present a proof that the rewriting always terminates. This technique can be used to eliminate multiple tests that occur naturally such as the test for `cons`-ness when both `car` and `cdr` are applied to the same object, but we also show how this technique can be used to automatically create specialized versions of general code, for example in order to create fast specialized versions of sequence functions such as `find` depending on the type of the sequence and the values of the keyword arguments supplied.

CCS Concepts

•Theory of computation → Rewrite systems; •Software and its engineering → Compilers;

Keywords

Intermediate code, compiler optimization, local graph rewriting

1. INTRODUCTION

In a language such as Common Lisp [1], it is hard to avoid redundant tests, even if the programmer goes to great lengths to avoid such redundancies. The reason is that even the lowest-level operators in Common Lisp require *type checks* to determine the exact way to accomplish the operation, so that two or more calls to similar operators may introduce redundant tests that are impossible to eliminate manually.

As an example of such an introduction of redundant tests, consider the basic list operators `car` and `cdr`. We can think of these operators to be defined[1] in a way similar to the code below:

```
(defun car (x)
  (cond ((consp x) (cons-car x))
        ((null x) nil)
        (t (error 'type-error ...))))

(defun cdr (x)
  (cond ((consp x) (cons-cdr x))
        ((null x) nil)
        (t (error 'type-error ...))))
```

where `cons-car` and `cons-cdr` are operations that assume that the argument is of type `cons`. These operations are implementation defined and not available to the application programmer.

Now consider some typical[2] use of `car` and `cdr` such as in the following code:

```
(let ((a (car x))
      (b (some-function)
      (c (cdr x)))
  ...)
```

After the inlining of the `car` and `cdr` operations, the code looks like this:

```
(let ((a (cond ((consp x) (cons-car x))
               ((null x) nil)
               (t (error 'type-error ...)))
      (b (some-function)
      (c (cond ((consp x) (cons-cdr x))
               ((null x) nil)
               (t (error 'type-error ...)))
  ...)
```

We notice that the test for `consp` occurs twice, and that the second occurrence is *dominated by* the first one, i.e., every control path leading to the second occurrence must pass by the first occurrence as well.

[1]For these particular operators, an implementation may use some tricks to avoid some of these tests, but such tricks are not generally usable for other operators. We still prefer to use `car` and `cdr` as examples, because their definitions are easy to understand.

[2]In this case, the programmer might use the standard macro `destructuring-bind`, but for reasons of simplicity, that macro will very likely expand to calls to `car` and `cdr`, rather than to some implementation-specific code that avoids the redundant tests.

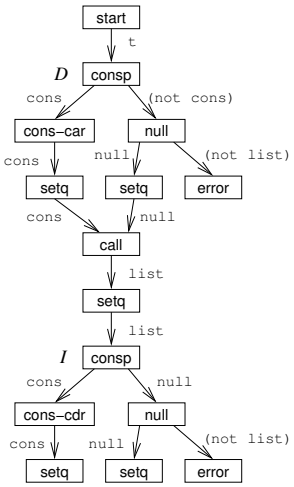

Figure 1: Control flow generated by a typical compiler.

As a consequence, the outcome of the second test for `consp` is always the exact same as the outcome of the first one. Unfortunately, this fact can not be easily exploited. To see why, we need to study the intermediate code generated by a typical compiler as a result of compiling the example program. The result is shown in Figure 1.

As Figure 1, shows, the intermediate code takes the form of a *control-flow graph* in which the nodes are *instructions* and the arcs represent the control flow. When we use the terms *predecessor* and *successor*, they refer to the relationship between two instructions in the control-flow graph, as defined by the control arcs.

In Figure 1, we have omitted references to data so as to simplify the presentation. In this intermediate representation, we have also eliminated scoping constructs, so that liveness of a variable is defined to be between its first assignment and its last use. Each control arc is annotated by a type descriptor, indicating the type of the variable `x` at that point in the execution of the program.

After the first `let` binding has been executed, the control arc with the type `cons` and that with the type `null` both arrive at the instruction that calls `some-function` which is the start of the second `let` binding. As a consequence, after the second `let` binding has been established, the type information available for `x` is `(or cons null)`, which is the same as `list`.

In order to avoid the second test for `consp`, we need to *replicate* the instructions corresponding to the establishment of the second `let` binding. In this paper, we introduce a technique for accomplishing this replication using *local graph rewriting*. The advantage of this technique is that it is very simple to implement, and that its semantic soundness is trivial to prove. We also prove that the technique always terminates, no matter how complicated the intermediate computation between the two tests.

2. PREVIOUS WORK

Mueller and Whalley [4] describe a technique for avoiding conditional branches by path replication. Their work includes heuristics for determining whether such replication is worthwhile. However, their technique for replicating the paths is not based on graph rewriting, and they do not supply a proof that their technique is correct.

3. OUR TECHNIQUE

3.1 General description

Our technique consists of applying *local graph rewriting* to the graph of instructions in intermediate code. Local graph rewriting has the advantage of being simple, both to implement and when it comes to proving correctness.

For the purpose of this paper, we assume that some initial phase has determined that the following conditions are respected:

1. there are two instructions, D and I, in the program that are identical tests,
2. the variable being tested is the same in D and I,
3. D dominates I, and
4. the variable being tested is not assigned to in any path from D to I.

In a real compiler, such a phase probably does not exist. Some conditions are easier to verify if the compiler translates the intermediate code to SSA form [2, 3], and some conditions can be verified during the execution of our technique, avoiding the need to include them in a separate phase.

In Figure 1, the two test instructions labeled D and I respectively verify the conditions listed above, and we will use these two instructions to illustrate our technique.

During the execution of our algorithm, the instruction I will be *replicated*, so that it is part of some set S in which every replica remains dominated by D. Initially, S contains I as its only element.

In our technique, we keep track of the outcome of the test in the *control arcs* of the control-flow graph. We can think of this information as being represented as *labels* associated with control arcs:

- An arc is unlabeled if we have no information concerning the outcome of the test at that point in the program.
- An arc is labeled *true* if the outcome of the test at that point in the program is known to be true.
- An arc is labeled *false* if the outcome of the test at that point in the program is known to be false.

Initially, only the outgoing arcs of D and I have a label.

Our technique involves the repeated application of the first applicable rewrite rule in the following list to some arbitrary element of S, say s, that does not itself have an immediate predecessor in the control-flow graph that is also an element of S:

1. If s has no predecessors, then remove it from S.
2. If s has an incoming arc labeled *true*, then change the head of that arc so that it refers to the successor of s referred to by the outgoing arc of s labeled *true*.

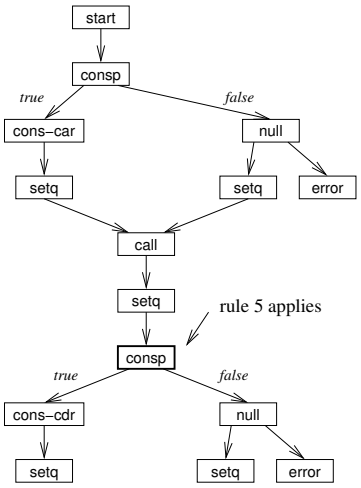

Figure 2: Initial instruction graph.

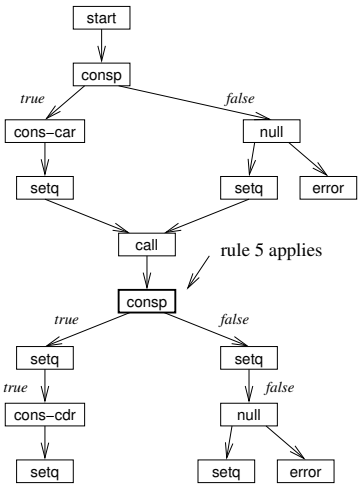

Figure 3: Result after one rewrite.

3. If s has an incoming arc labeled *false*, then change the head of that arc so that it refers to the successor of s referred to by the outgoing arc of s labeled *false*.

4. If s has $n > 1$ predecessors, then replicate s n times; once for each predecessor. Every replica is inserted into S. Labels of outgoing control arcs are preserved in the replicas.

5. Let p be the (unique) predecessor of s. Remove p as a predecessor of s so that existing immediate predecessors of p instead become immediate predecessors of s. Insert a replica of p in each outgoing control arc of s, preserving the label of each arc.

Rewrite rules are applied until the set S is empty, or until each element of S has an immediate predecessor in the control-flow graph that is also a member of S. An element of S could have an immediate predecessor like that if the dominated instruction I were part of a loop. We need to exclude such elements, or else our technique might not terminate in all cases.

3.2 A simple example

Let us see how our technique works on the example in Figure 1. The initial situation is shown in Figure 2. The instructions that are members of S are drawn with a slightly thicker box.

As Figure 2 shows, the second `consp` is dominated by the first, so it becomes the only member of the set S. The last rewrite rule applies to the second `consp` so that the `setq` is replicated as its successors. The result of this first rewrite is shown in Figure 3.

As we can see in Figure 3, the last rewrite rule applies again resulting in the replication of the `call`. The result after the second rewrite is shown in Figure 4.

As we can see in Figure 4, the second `consp` now has two predecessors, and both incoming arcs are unlabeled. Therefore, rewrite rule number 4 applies and the `consp` is replicated. As a result, S now has two members. The result of applying this rule is shown in Figure 5.

We now choose the leftmost replica of the second `consp` to apply our rules to. It has a single predecessor with an

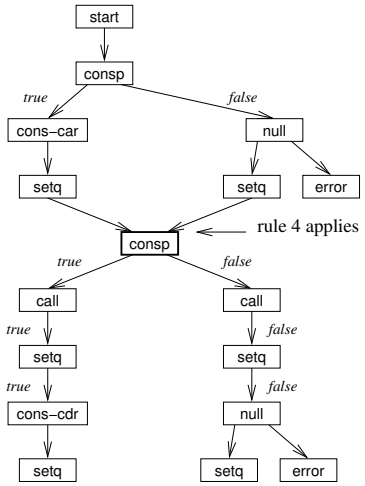

Figure 4: Result after two rewrites.

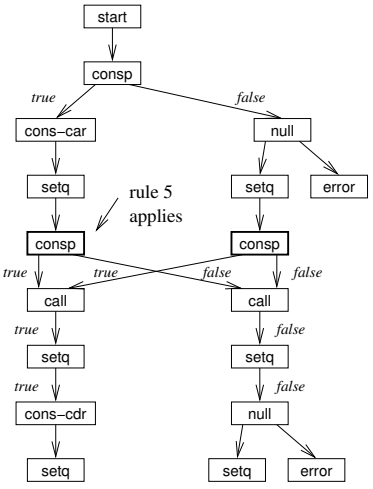

Figure 5: Result after replicating the test.

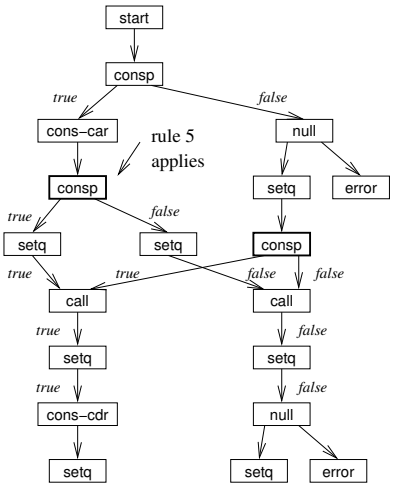

Figure 6: Result after replicating `setq`.

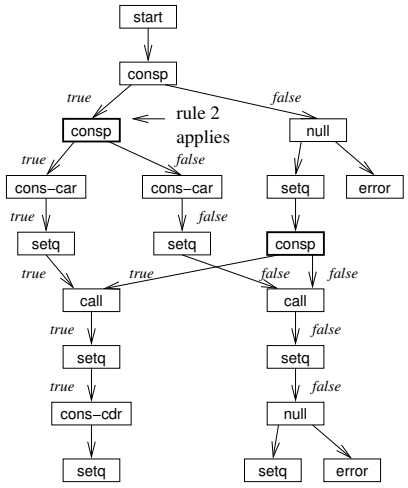

Figure 7: Result after replicating `cons-car`.

unlabeled incoming control arc, so the last rewrite rule applies. We replicate the `setq` in both branches of the test, giving us the result shown in Figure 6.

In Figure 6, the last rewrite rule applies again, and we replicate the `cons-car`, giving us the situation shown in in Figure 7.

As Figure 7 shows, the `consp` instruction now has a single predecessor, but the incoming arc has a known outcome of the test, namely *true*. Therefore, rewrite rule number 2 applies. The left outgoing arc of the first `consp` is redirected to go directly to the `cons-car` instruction. The result of applying this rule is shown in Figure 8.

At this point, the `consp` that we have been processing has no predecessor. Therefore we apply rule number 1 and remove it from S. Removing all instructions that can not be reached from the start instruction gives the situation shown in Figure 9.

Analyzing Figure 9, we can see that if the result of the first `consp` yields *true*, then no second test is performed. Instead, the variable `a` is set to the result of the instruction `cons-car`,

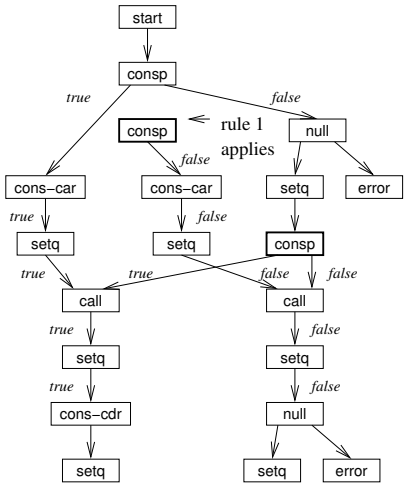

Figure 8: Result after short-circuit `consp`.

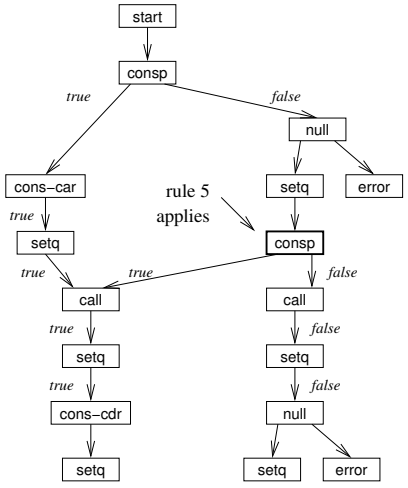

Figure 9: Result after removing unreachable instructions.

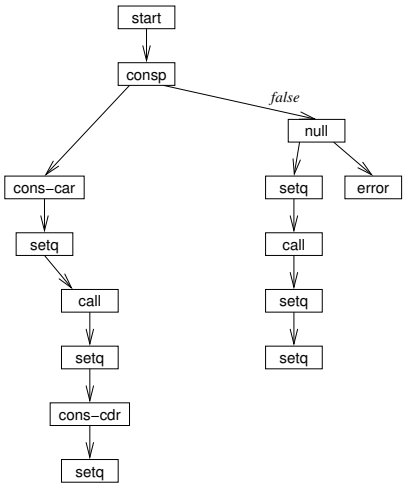

Figure 10: Final result.

the variable `b` is set to the result of the call, and the variable `c` is set to the result of the instruction `cons-cdr`. Applying the same rules to the remaining `consp` instruction in S and then to the second `null` instruction (which is now dominated by the first), yields the final result shown in Figure 10.

This example represents a control graph that is particularly simple, in that there are no loops between the first and the second `consp` instructions. Our technique must obviously work no matter the complexity of the control graph, as long as the first test dominates the second.

3.3 Proof of correctness and termination

The correctness of our technique is easy to prove, simply because each rewrite rule preserves the semantics of the program. The last rewrite rule preserves the semantics only under certain circumstances which are easy to verify:

- The predecessor does not assign to a lexical variable that is read by the test instruction. This condition is respected because we have assumed that the variable being tested is not assigned to in any path between the first and the second occurrences of the test, as condition number 4 in Section 3.1 requires.
- The predecessor must not have any other side effect that may alter the outcome of the test. By restricting the test to lexical variables, this restriction is also respected.

Termination is a bit harder to prove. One way is to find some non-negative *metric* that can be shown to strictly decrease as a result of the application of each rewrite rule. We have not found any such metric. However, this conundrum can be avoided by a simple *grouping* of the rewrite rules. This grouping is not required to be present in the implementation of our technique, only in the termination proof.

To see how the rewrite rules can be grouped, consider a general case where the test instruction has some arbitrary number of labeled or unlabeled incoming control arcs. Rules number 2 and 3 are first applied a finite number of times. What happens next depends on the number n of unlabeled incoming control arcs:

- If $n = 0$ the first rewrite rule applies, in which case the instruction is removed from the set S.
- If $n = 1$, the last rewrite rule is applied. The crucial characteristic of this rewrite rule is that the total number of unlabeled control arcs decreases by one.
- If $n > 1$, rewrite rule number 4 is applied. Notice that the number of unlabeled control arcs is not modified by the application of this rule.

For the purpose of this proof, we assume that the individual rewrite steps in a group happen immediately after each other, so that for a particular instruction, the labeled incoming control arcs are first eliminated, the same instruction is then potentially replicated, and finally, the last rewrite rule is applied to one of the replicas. However, the implementation does not have to work that way in order for termination to be certain.

In other words, we can create groups of rewrite steps, where a group can be formed according to one of the following *group types*:

A. A group in this type has a finite number of applications of rewrite rules number 2 and 3, followed by a single application of rewrite rule number 1.

B. A group in this type has a finite number of applications of rewrite rules number 2 and 3, followed by a single application of rewrite rule number 5.

C. A group in this type has a finite number of applications of rewrite rules number 2 and 3, followed by a single application of rewrite rule number 4, followed by a single application of rewrite rule number 5.

With this information, we can create a metric consisting of a pair (U, N), where U is the total number of unlabeled control arcs of the program and N is the number of elements of the set S. Two pairs can now be compared using a *lexicographic order*, so that for two pairs (U_1, N_1) and (U_2, N_2), (U_1, N_1) is *strictly smaller than* (U_2, N_2), written $(U_1, N_1) < (U_2, N_2)$, if and only if either $U_1 < U_2$ or $U_1 = U_2$ and $N_1 < N_2$.

THEOREM 1. *The rewrite algorithm terminates.*

PROOF. As a result of a rewrite according to a group of type A, U remains the same, but N decreases by 1. As a result of a rewrite according to a group of type B or C, U decreases by 1 (but N may increase). Since U and N are both non-negative integers, we must reach a normal form after a finite number of rewrites. □

4. BENEFITS OF OUR TECHNIQUE

The main benefit of our technique is its simplicity. This simplicity is important both in terms of its implementation and in terms of ensuring termination for all possible control graphs.

As the example in Section 3.2 shows, redundant tests can be avoided in cases where it is not possible to express this redundancy by the use of any portable lower-level constructs. Situations similar to the one in the example occur naturally in many programs:

- If the same variable is used in more than one consecutive numeric operation, then there will be redundant tests to determine the exact numeric subtype of the contents of that variable. An important special case is to determine whether a particular value is of type `fixnum`.

- If a variable holding an array is used in more than one consecutive operation to access some element, then there will be redundant tests to determine various aspects of the array that influence the way the indices and the elements are handled, such as the upgraded element type and whether the array is simple or not.

A particularly interesting special case of these situations occurs when the dominated test is part of a loop. It is particularly interesting, because our technique will then eliminate a test that would otherwise potentially be executed a large number of times. This feature can be taken advantage of in a highly portable version of some of the Common Lisp *sequence functions*. By duplicating a very general loop in every branch of a multiway test for keyword arguments such as `test`, `test-not`, and `end`, each copy of the loop will automatically be simplified differently according to the particular branch it occurs in.

Our technique has some disadvantages as well. First of all, the size of the code may increase, which can have a negative influence on cache performance, especially when different invocations of the code result in different results of the test. In fact, if several variables with overlapping regions of liveness are processed by our technique, the result may be an *exponential blowup* of the size of the code in the overlapping region. It is outside the scope of this paper to discuss heuristics that will determine the conditions for applying our technique, but such conditions are required to avoid such problematic effects.

The increase of the size of the code automatically means longer compilation times as well. Techniques that work on global information about the program can avoid some of these disadvantages, at the cost of increased complexity compared to our simple local rewrite technique.

5. CONCLUSIONS AND FUTURE WORK

We have defined a technique for eliminating redundant tests in intermediate code. The technique relies on replication of code paths between two identical tests. So far, our technique only defines a *mechanism* for achieving the result. It does not yet define a *policy* stating when the technique should be applied.

The question of policy is an important one, because with a large number of redundant tests in the intermediate code, there is a possibility for *exponential blowup* of the code size. Future work involves defining a reasonable policy to avoid such pathological cases.

The technique described in this paper will become available as one of the optimization techniques provided by the Cleavir compiler framework that is currently part of the SICL project.[3] Only then will it be possible to determine the exact characteristics of our technique in terms of applicability, computational cost, performance gain of compiled code, and size increase of typical programs.

[3]See https://github.com/robert-strandh/SICL

6. ACKNOWLEDGMENTS

We would like to thank Philipp Marek for providing valuable feedback on early versions of this paper.

7. REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp*. American National Standards Institute, 1994.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[4] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 56–66, New York, NY, USA, 1995. ACM.