Removing redundant tests by replicating control paths

Irène Durand & Robert Strandh

LaBRI, University of Bordeaux

April, 2017

European Lisp Symposium, Brussels, Belgium ELS2017

Context: The SICL project

https://github.com/robert-strandh/SICL
Several objectives:

- Create high-quality *modules* for implementors of Common Lisp systems.
- Improve existing techniques with respect to algorithms and data structures where possible.
- Improve readability and maintainability of code.
- Improve documentation.
- Ultimately, create a new implementation based on these modules.

Compiler framework

A large part of SICL is a framework (called Cleavir) for creating Common Lisp compilers.

Cleavir is written so that an implementation can adapt the framework to the needs of the implementation.

At the same time, Cleavir provides reasonable default behavior that the implementation can benefit from.

In particular, Cleavir will contain a large number of standard compiler optimization algorithms, and a few of our own.

Purpose of current work

When possible, avoid redundant tests.

A test is redundant if a preceding identical test that dominates this one exists.

We detect redundant tests in *intermediate code*, and we eliminate them using *local graph rewriting*.

Previous work

Frank Mueller and David Whalley presented a paper at PLDI in 1995, titled “Avoiding Conditional Branches by Code Replication”.

That paper uses ad-hoc techniques to detect redundant tests and transform the code to avoid them.

We are unaware of any other work in this domain.

An example

A client implementation might define `car` and `cdr` like this (Clasp and SICL both do):

```
(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 primitive operations that require the argument to be of type `cons`.

An example

Now suppose we have the following code to compile:

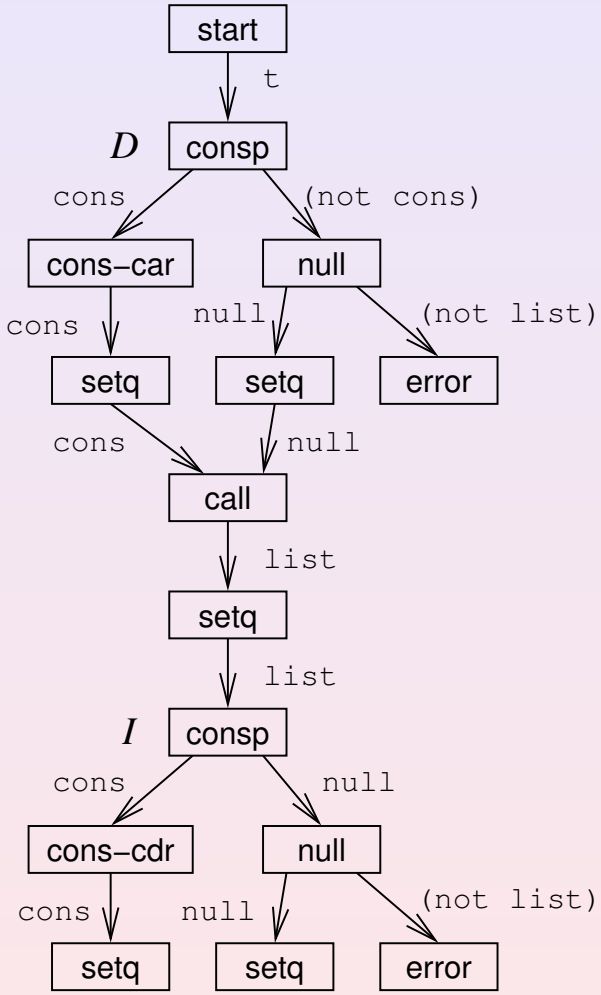
```
(let ((a (car x))
      (b (some-function)
      (c (cdr x)))
  ...)
```

An example

After inlining car and cdr, we get the following code:

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

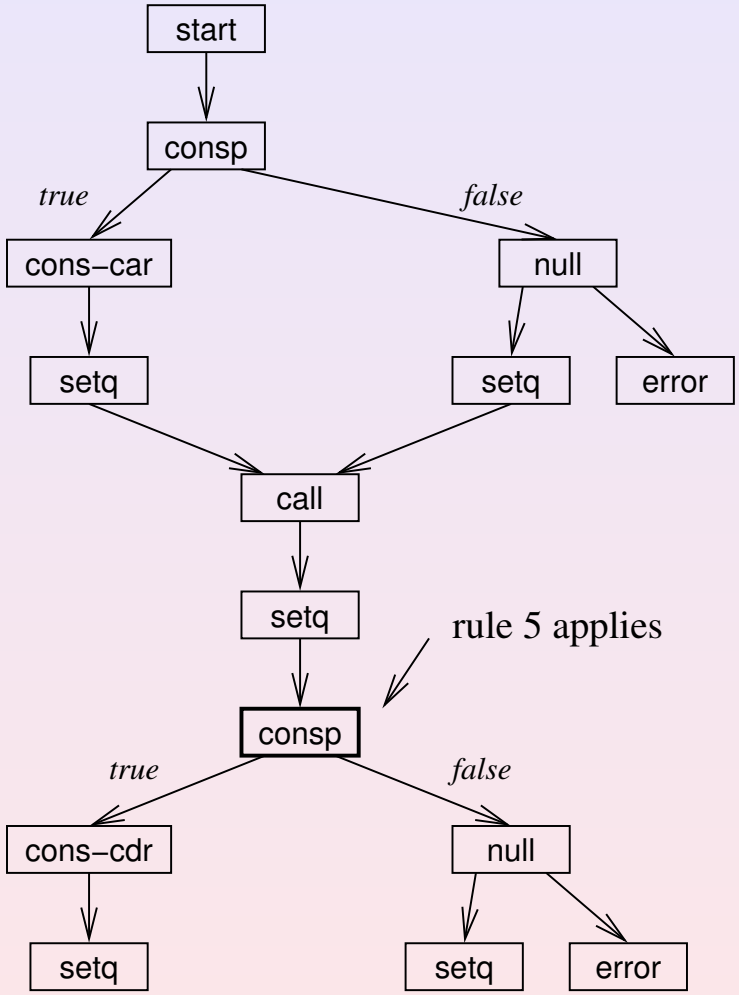
Resulting intermediate code



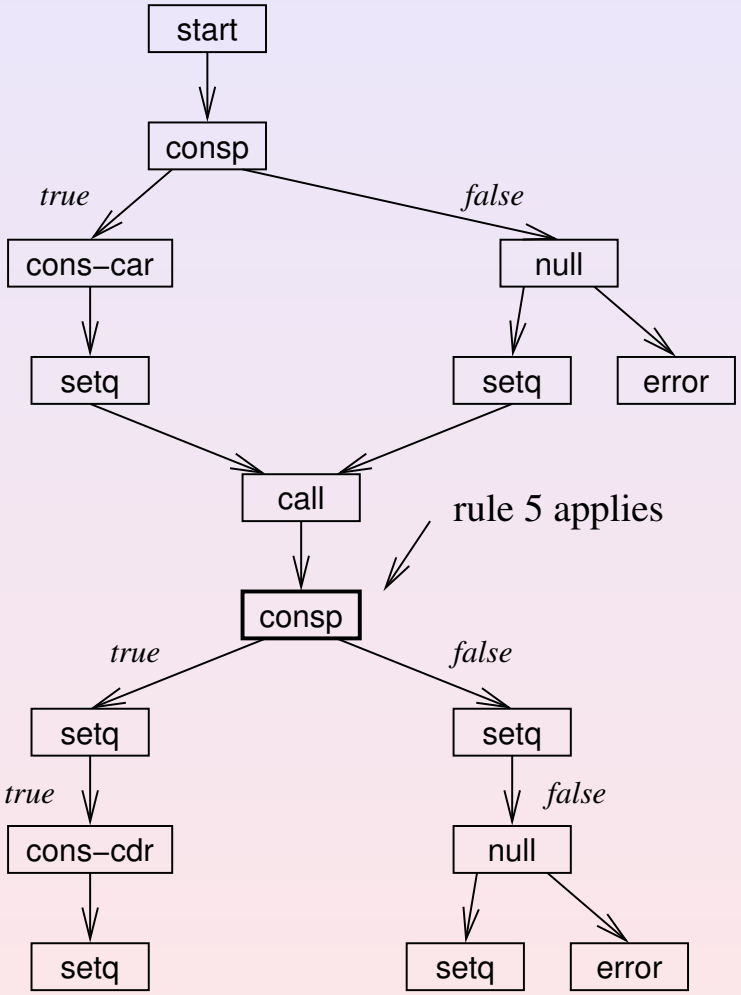
Rewrite rules

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*.
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.

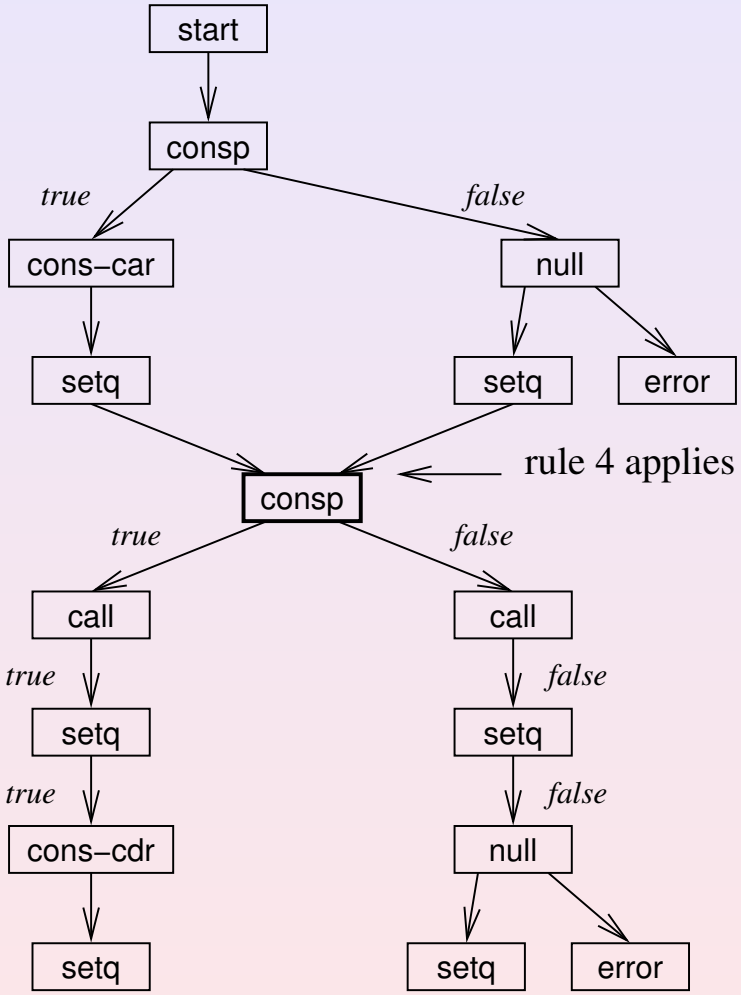
Initial instruction graph



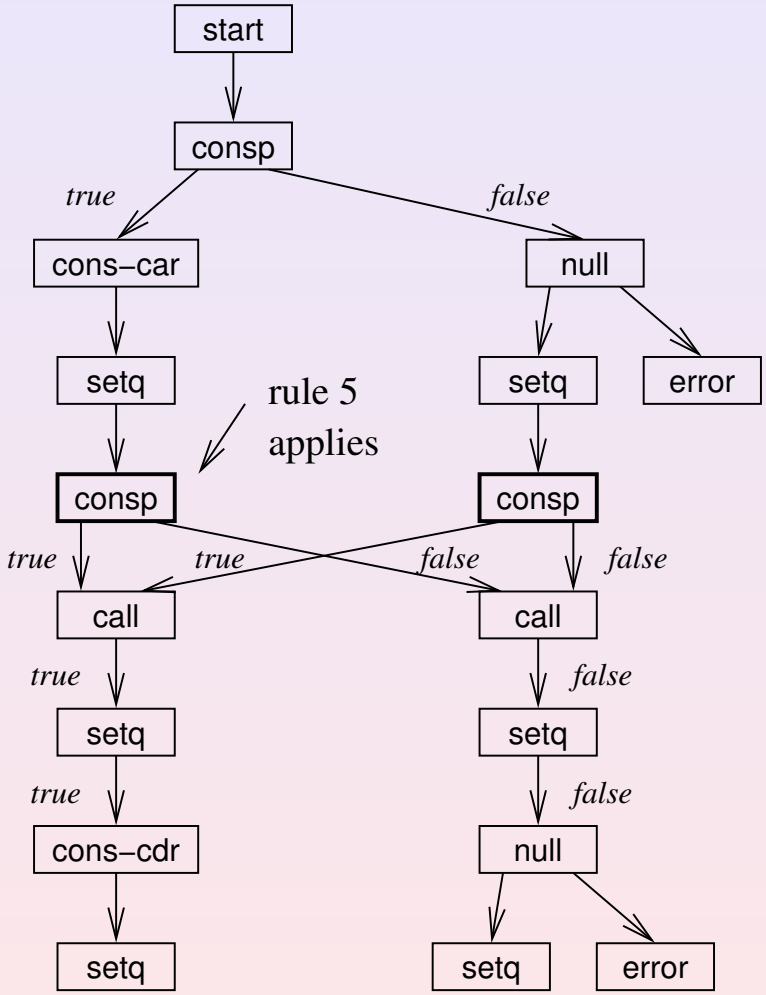
Result after one rewrite



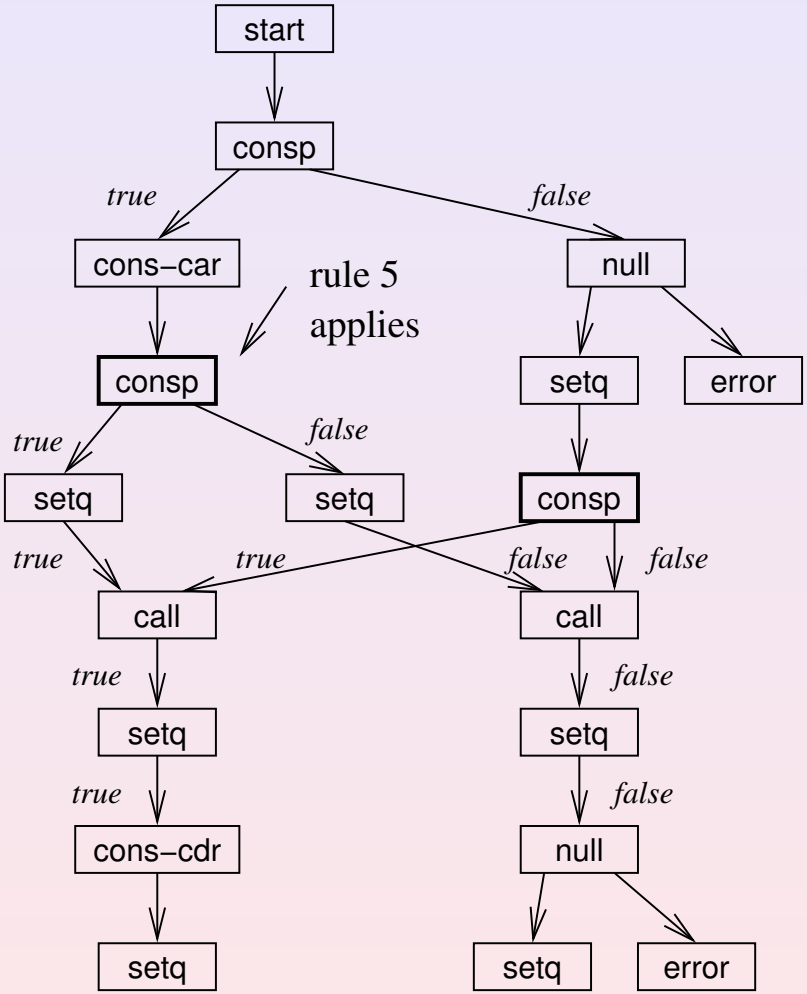
Result after two rewrites



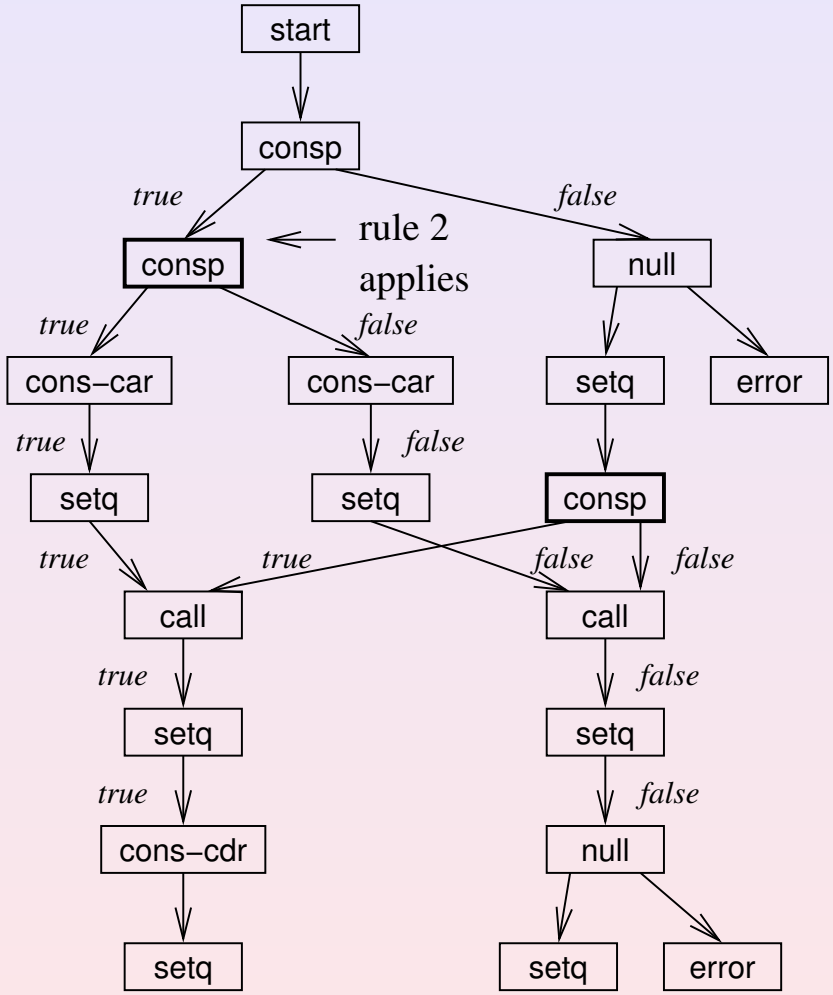
Result after replicating the test



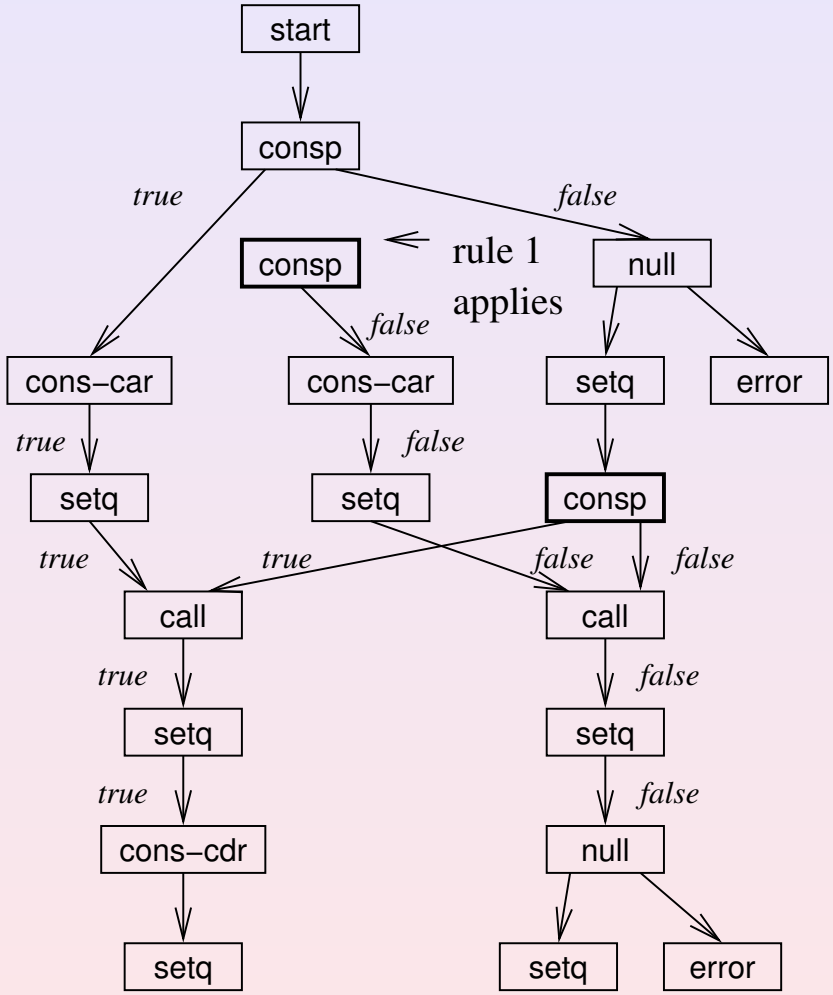
Result after replicating `setq`



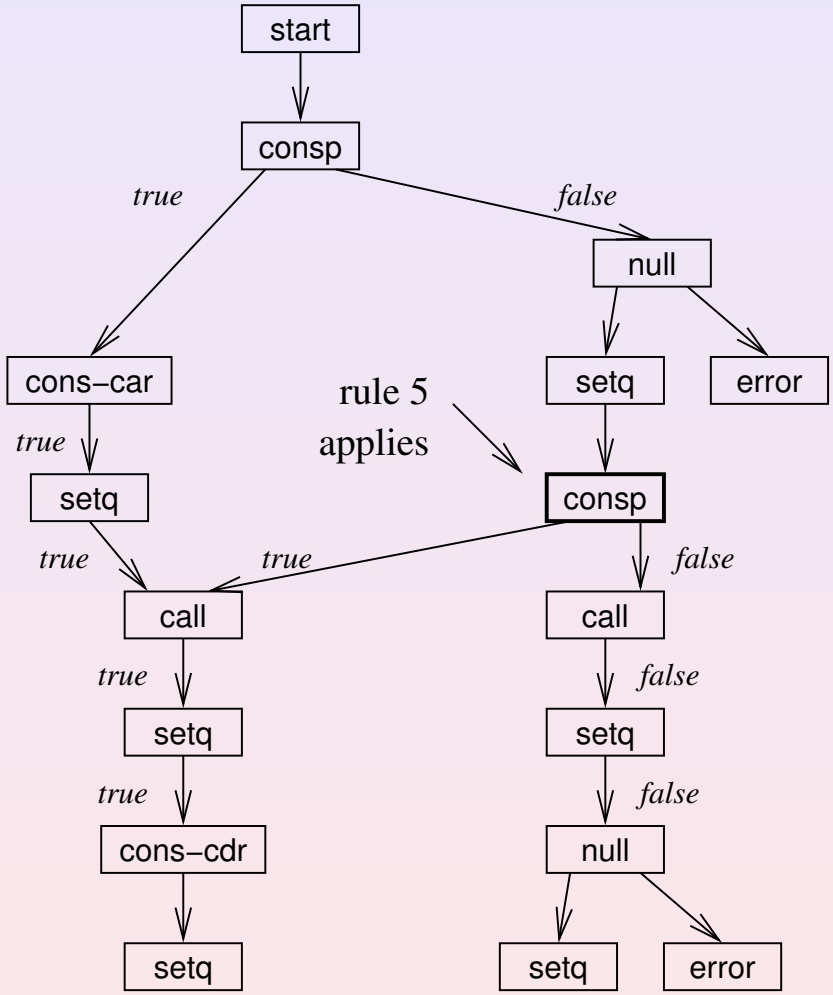
Result after replicating `cons-car`



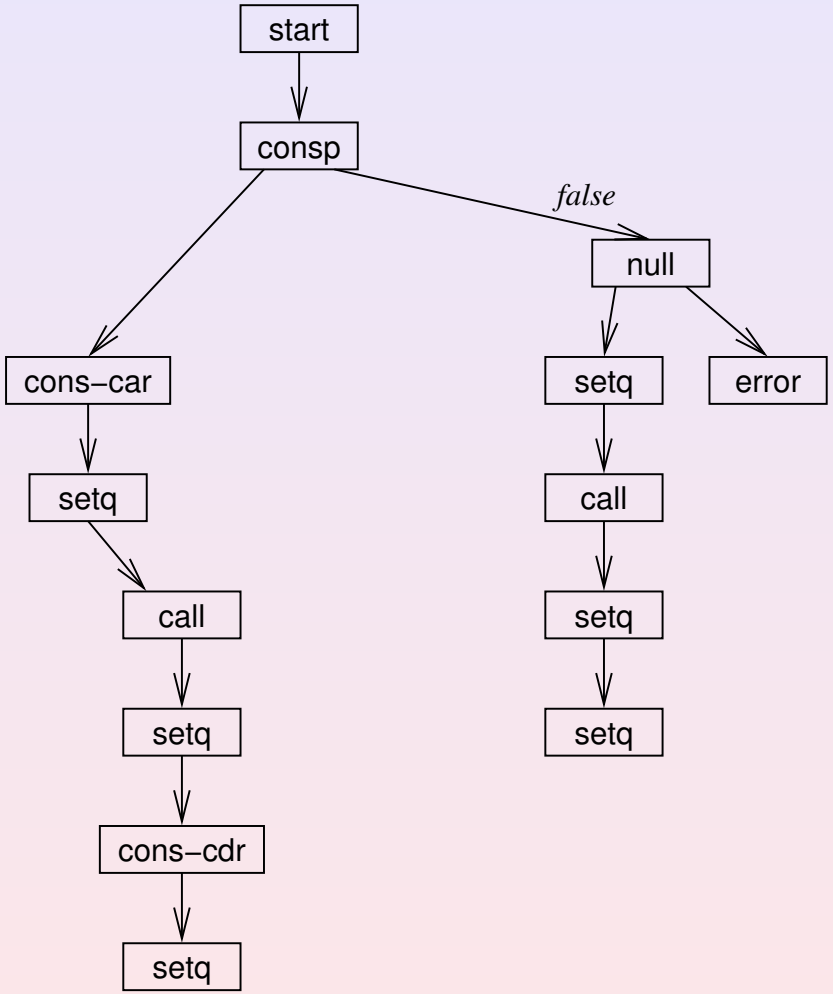
Result after short-circuit consp



Result after removing unreachable instructions



Final result



Advantages to our technique

- It is simple to implement.
- Correctness is obvious, because each rewrite step preserves the semantics of the program.
- In the paper, we give a proof of termination. It works even when loops are replicated.

Disadvantages

- The resulting code is bigger.
- If many, overlapping zones of liveness and redundant tests exist, then code size may increase exponentially. We conjecture that this case is very infrequent.
- Local rewriting is probably not the best way in terms of compile time performance.

Future work

- The current work discusses only mechanism. We must establish a *policy* for when to apply our technique, in particular to avoid huge increases in code size.
- Our technique must be implemented in Cleavir and tested on real programs to determine improvements in performance.

Acknowledgments

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

Thank you

Questions?