
# *Introduction to Rocq* — Arthur Azevedo de Amorim

Lecture 4 - *June 24, 2026*

## 1 Optimizations

In this lecture we look into the *src/opt.v* file. In this file we are concerned with optimizing the execution of programs written on the Imp language defined in the previous lecture. First, let's look at *cf_add*. This optimization adds constants together, and removes zero sums.

To assert that this optimization is correct we define the following lemma:

```
Lemma cf_add_correct : forall e1 e2 s,
  eval_expr (cf_add e1 e2) s
  = eval_expr (Binop Add e1 e2) s.
```

Its proof follows directly from case analysis. As a result, we can safely optimize additions knowing that the result is equivalent to the original result. This is a clear display of the advantages of the use of proof assistants for the development of programming languages.

Similarly, we can optimize multiplications. For instance, all multiplications by zero can be greatly simplified to zero, and all multiplications by 1 can be simplified to the non-1 term. To assert that this is correct we state the following lemma:

```
Lemma cf_mul_correct : forall e1 e2 s,
  eval_expr (cf_mul e1 e2) s
  = eval_expr (Binop Mul e1 e2) s.
```

But the proof is not possible! Because we are not considering cases where an error might happen on the discarded terms. [1].

[1] For a more detailed explanation please refer to line 75 of src/opt.v

Compiled By:
Bas Laarakker, Songlin Jia, Ulises Torrella

# 2 Improvements

One could argue that cf_mul is actually an improvement of the original semantics, by reducing the numbers of erroneous outputs. To define an improvement on an execution we define the following relation:

```
Definition improves {T} (x y : result T) : Prop :=
  y = Error \/ x = y.
```

In simple words, a result $x$ improves a result $y$ if they are either the same, or $y$ is an error. We quickly assert reflexivity, transitivity and compositionality.

Now we can show that the optimization of mult is an improvement of the original semantics.

```
Lemma cf_mul_correct : forall e1 e2 s,
  improves
    (eval_expr (cf_mul e1 e2) s)
    (eval_expr (Binop Mul e1 e2) s).
```

## 2.1 Improving the proof

The proof of cf_mul_correct is possible, but requires tedious manual case analysis. One option is to resort to proof automation, but actually we can do something else. In this lecture we see how to define a case analysis principle. This principle is customly made to match our optimization specification, and thus we can apply it to destruct the proof of *cf_mul_correct* into solely the interesting cases. First, we need to prove that cf_mul respects this specification. Thankfully, Rocq is smart enough to do this for us with case analysis and *eauto*. Now, the proof of correctness for cf_mul goes much smoother.

## 2.2 More optimizations

We can continue optimizing other binary operations, such as Sub and Leq, and compose them together into one optimization function: cf_binop. Similarly as done for mutiplication, we can show that cf_binop improves the result of the original semantics:

```
Lemma cf_binop_correct : forall b e1 e2 s,
  improves (eval_expr (cf_binop b e1 e2) s)
    (eval_expr (Binop b e1 e2) s).
```

Finally, we can apply these optimizations to the binary operations of an expression. To do this, we need a fix-point to recursively optimize binary operations if expressions and sub-expressions:

```
Fixpoint cf_expr (e : expr) : expr :=
(* <exercise-only>(* Replace this *) Num 0.</exercise-only> *)
(* <solution> *)
  match e with
  | Binop b e1 e2 => cf_binop b (cf_expr e1) (cf_expr e2)
  | _ => e
  end.
```

## 3 More study material

This is as far as this lecture goes. However, in the repository, we can find a similar approach to optimize commands. In contrast with expressions, these optimizations have to handle state. We refer the reader to line 384 of src/opt.v to see how it is done.