Towards Optimising Certified Programs by Proof Rewriting

Kiran Gopinathan
National University of Singapore
gopiandcode.uk

Ilya Sergey
National University of Singapore
ilyasergey.net

Abstract

We present ongoing work on the use of e-graphs for the transformation of certified programs produced by a deductive synthesiser for heap-manipulating programs. We develop a strategy for optimising proofs using the mechanisms of e-graphs, propose a novel e-class analysis over *proof scripts*, and report on its capabilities and limitations.

1 Introduction

The age-old adage goes *"the only constant in life is change"*, and nowhere is this more true than with software maintenance. Unfortunately, this is a rather inconvenient fact for practitioners of formal verification, as the very proofs used to certify the correctness of a program become the obstacles that prevent changing their associated programs: in order to modify such a program, after changing the code, the user must then also update the proof, effectively doubling the maintenance effort. In this ongoing work, we investigate the problem of optimising the source code of certified programs *while* preserving their proofs.

Consider the program produced by the SuSLik program synthesiser [**?**] to free a list alongside the proof script (*i.e.*, a sequence of logic rule applications) of its correctness:

```
// {lseg(x,S)}  listfree(x)  {emp}
void listfree(loc x) {
  if (x == 0) {          Open(x, lseg)
    return;              – Emp
  } else {
    let h = *x;          – Read(h, x, 0)
    let t = *(x + 1);      Read(t, x, 1)
    listfree(t);           Call(listfree, t, lseg(t, S1))
    free(x);               Free(x)
  }                        Emp
}
```

The program is a fairly straightforward implementation of a function to recursively free a linked list, following the specification in **??**. It starts by checking whether the input argument `x` is null, in which case the list is empty. In the case that `x` is not null, the program reads the head of the list, recurses on the tail, and then finally frees the head itself.

The proof script on the right then describes the deductive argument SuSLik constructed to produce the program, closely following the structure of the program. Prior work has shown that the proof tree produced during this search can be extracted into a proof script in the language of theorem provers such as Coq to serve as a formally checked guarantee of the correctness of the synthesised program [**?**].

predicate lseg (**loc** x, **set** S){
| $x = 0 \Rightarrow \{\ S = \emptyset; \text{emp}\ \}$
| $x \neq 0 \Rightarrow \{\ S = \{v\} \cup S_1;$
$[x, 2] * x \mapsto v * (x + 1) \mapsto nxt * lseg(nxt, S_1)\ \}\ \}$

Figure 1. A Separation Logic predicate for a linked list.

While this program may be correct, the observant reader may have noted that it contains a subtle inefficiency. In particular, the function is not stack-safe: the recursive call is not in a tail-call position, so each recursive invocation of the function *must* allocate a new frame on the stack. For this particular case, an experienced C-developer might notice that the behaviour of the function is not changed if we permute the last two statements, and manually rewrite the else case with these two statements swapped. However, now we run into a problem, in that our proof of correctness no longer holds for this rewritten program. *Is there a way to change the program while maintaining its proofs?*

2 E-Graphs Over Proofs

One way to support these kinds of optimisations of generated programs, while preserving their proofs of correctness, is to *directly optimise* the proofs themselves, and then *later* re-extract the optimised programs from the modified proofs. While transforming proofs rather than programs are a relatively novel domain for optimisations, they give rise to the same challenges that one might face during program transformations such as the phase ordering problem. As such, we choose to leverage the mechanisms of equivalence graphs for this task, and perform our proof optimisations by means of encoding them as rewrite rules for an e-graph.

Following the running example of the list free function presented above, we can apply this methodology by transforming the proof such that the **Call** and **Free** rules are permuted, using the following rewrite rule for proof scripts:

Call($?f, ?H$);	$\Rightarrow$	**Free**($?x$);
Free($?x$);		**Call**($?f, ?H$);
...		...

EGRAPHS 2022, June 2022, San Diego, CA, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

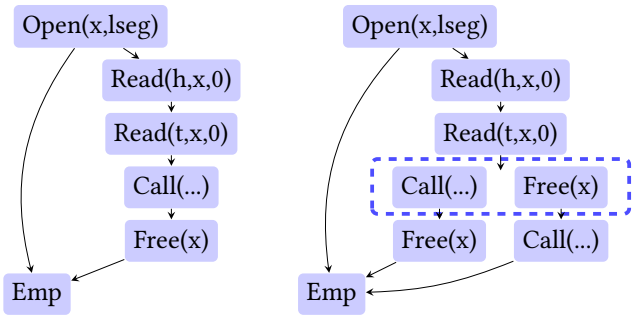

Figure 2. E-graphs for proof script rewriting.

Notice that while such a transformation might not be valid in general, we can ensure its soundness by making the rewrite conditional on the fact that the quantified variable $?x$ does not *syntactically* occur in the heaplets framed in to the function call, which are helpfully included as the second argument to the Call proof rule. That is, it is safe to transpose a call and a `free`, *provided* that the call *does not use* any of the data that will be deallocated by the `free`. As we are operating over the proof script itself rather than the raw program, we are able to encode this *semantic* constraint about the behaviour of the program through a purely *syntactic* check.

Applying the above conditional rewrite rule to an e-graph embedding the list-free proof, we obtain the equivalence classes presented on the **??**. Notice that after applying the transformation, the e-graph has merged the roots of both subtrees for this proof into a single equivalence class that encodes the fact that multiple proof trees may dispatch the same proof obligation. Finally, in order to recover the optimised certified program, we extract a concrete proof tree from the e-graph, tuning the extraction heuristics to prioritise programs with *nice* properties (such as tail call recursion), and then use SuSLik to *extract* the corresponding program from the proof tree using its proof interpreter [**?**].

3 Towards an E-Class Analysis for Proofs

As we have shown, the more explicit nature of proof scripts makes it possible to encode deeper semantic constraints by means of simple syntactic checks, however, this comes with its own drawbacks. For example, to encode the *commutativity* of any two proof rules, as we are reasoning purely syntactically, we must manually develop a bespoke syntactic check each time to enforce the soundness of the transformation, requiring $O(n^2)$ unique rewrite rules to support all possible proof rules.[1] Clearly reasoning in this way can quickly become impractical as the number of proof rules increases, let alone supporting more diverse types of transformations.

To alleviate this problem, we can move part of our reasoning from the purely syntactic to the semantic. To do so, we propose a novel e-class-based analysis to reason about proof trees, building on the e-class analysis framework for e-graphs described in the paper introducing the egg library [**?**].

For each equivalence class in the e-graph, we associate an abstract *proof footprint* that encodes a representation of the *set* of concrete proof goals that any subtree in the class can safely dispatch. Concretely, a proof footprint has the same structure as a SuSLik synthesis goal as before, but also includes a catch-all wildcard expression, to encode the fact that proof trees may be agnostic to the value of particular terms. This proof footprint is then built up in a bottom-up fashion, starting from the leaves of the proof tree, working upwards by defining a *footprint-transformer* for each proof rule that "reverses" the operation of the rule to capture the increasing complexity of proof goals that the extended subtrees can handle. As an example, the footprint transformer for a **Free** rule would modify a footprint by *adding* the freed blocks back into the pre-condition, using wildcards to represent the unknown contents of the block.

With this e-class analysis in hand, we can then modify our rewrite rules to use the footprints of the subtrees that they manipulate, and thereby simplify their implementation. For example, in the case of commutativity, we can say that *any* two proof rules at the root of a given proof tree can commute if the footprints of the resulting proof trees are equal *up to wildcards*. This way, we cut down the implementation cost of defining rewrite rules for commutativity from $O(n^2)$ down to simply the cost of writing $O(n)$ proof transformers for each proof rule, with the added benefit that the abstract footprints may then be used to enable and guide further proof rewritings.

4 Limitations and Future Work

We have implemented our initial experiments on this proposed strategy for optimising certified programs in the OCaml programming language, as part of which, we have developed and published a free software library ego for encoding e-graphs.[2] Over the course of our implementation, we have identified the following unsolved challenges with this approach, which we hope to investigate further in future work:

- **Efficient encoding of reverse-substitutions:** many SuSLik proof rules involve substitution as part of their definition which is a problem when "reversing" them, as they end up introducing significant non-determinism, which is difficult to capture efficiently.
- **Validating footprint transformers:** footprint transformers themselves are rather cumbersome to implement, as they require running the expected behaviours of a rule *in reverse*, which can be easy to implement incorrectly.
- **General utility of proof footprints:** so far, we have only considered the use of proof footprints for the purposes of simplifying our commutativity rewrite rules, so the

[1] For reference, SuSLik has 19 proof rules in total, necessitating 361 bespoke rewrite rules to support just commutativity alone.

[2] Available at https://github.com/verse-lab/ego

question of whether these abstract footprints are useful in a more general setting remains unanswered.