Contents

1 Dios + Ruler paper

1.1 Rough sketch of the argument

Equality saturation with e-graphs are a promising way to easily build specialized high-performance compilers at the cost of compile time. However, their performance depends entirely on the *quality* of rewrite rules that are fed to the equality saturation engine. Bad rules can make the performance of the compiler unacceptably bad. Writing good rules is difficult process that requires careful and tedious tuning of the rule set.

Ruler showed that rewrite rules could be *automatically* generated using equality saturation equipped with an interpreter to give terms meaning. In theory, an automatic rule-set and an equality saturation could automatically produce a high-performance compiler from an interpreter.

However, in practice, *Ruler* generates far too many rules. In this paper, we present techniques to make it feasible to automatically learn rules suitable for use in an equality saturation based compiler.

The techniques are:

- The key observation is that equality saturation for compilers has the special property that it takes some `src` subset to a `target` subset (rather than being circular rules). Equality saturation naturally splits into 2 phases, 1) moving from `src -> target` and 2) optimizing `target`. By explicitly separating rules into these two phases, we can support orders of magnitude more rules for our equality saturation engine.
- **there is a quality of rules**
- you can get this for free, from the cost function.

equality sat is not enough for just compiling
Some RQs:

1.1.1 Do *phases* naturally arise when using equality saturation for compilation?

1.1.2 Does enforcing *phases* improve the performance of the compiler and the quality of the output?

1.1.3 Does equality saturation for compilation behave differently from equality saturation for optimization?

I need to expand upon this question. *What does it mean to behave differently*?

1.1.4 How does ruler run time affect the quality and performance of Dios?

1.1.5 How does this approach compare to other approaches?

Compare compiler performance and compiler output quality:

- Baseline diospyros
- Raw ruler + diospyros
- others?
- Other ways of handling large ruleset? Look into other approaches for AC matching.

1.1.6 How does *cost function* splitting compare to *syntax based* splitting?

1.1.7 Does this approach generalize to other languages?

Caddy would be interesting (also just to play with). I wonder how equivalence is defined. Hmm it probably doesn't really have an interpreter in the same sense. Unless there is some "core" language.

Not sure if there is this nice source, target distinction. Maybe it would be interesting to see that this approach doesn't work in that scenario? Not sure about that.

So I think it does have a source (Core Caddy) and target (Caddy). Not sure what equivalence is yet. Ok, equivalence is defined by some semantics that they don't mention in that paper. It's unclear what CVECs would be though hmm. Well I suppose the CVECs would be meshes and equivalence is measured with this Hausdorf equivalence. I'm not sure how hard this would be to extend Ruler to support?

1.1.8 How many of rules out of the ruleset are actually used?

1.1.9 How does this approach scale with increasing vector size?

1.1.10 Is it possible to devise a test to see if some language has phases and thus if Diospyros would apply?

2 Outline: Push-button Compiler Generation using Equality Saturation

2.1 Introduction

2.2 Background

2.2.1 E-graphs and equality saturation

2.2.2 Rewrite rule generation

2.3 Approach

2.3.1 phase splitting

2.4 Implementation Hacks

2.4.1 The right kinds of exponential growth

Exponential growth in *depth* (deeper trees explored) is ok. Exponential growth in *width* (nodes with lots of children) are unacceptable.

2.5 Evaluation

2.6 Related Works

2.7 Conclusion