Overview of “Proving Properties of Functional Programs by Equality Saturation”

Sergei Grechanik

2023

What is this talk about?

It’s about my dissertation “Proving Properties of Functional Programs by Equality Saturation” that was defended in 2018.

Main idea: Let’s implement Supercompilation via Equality Saturation.

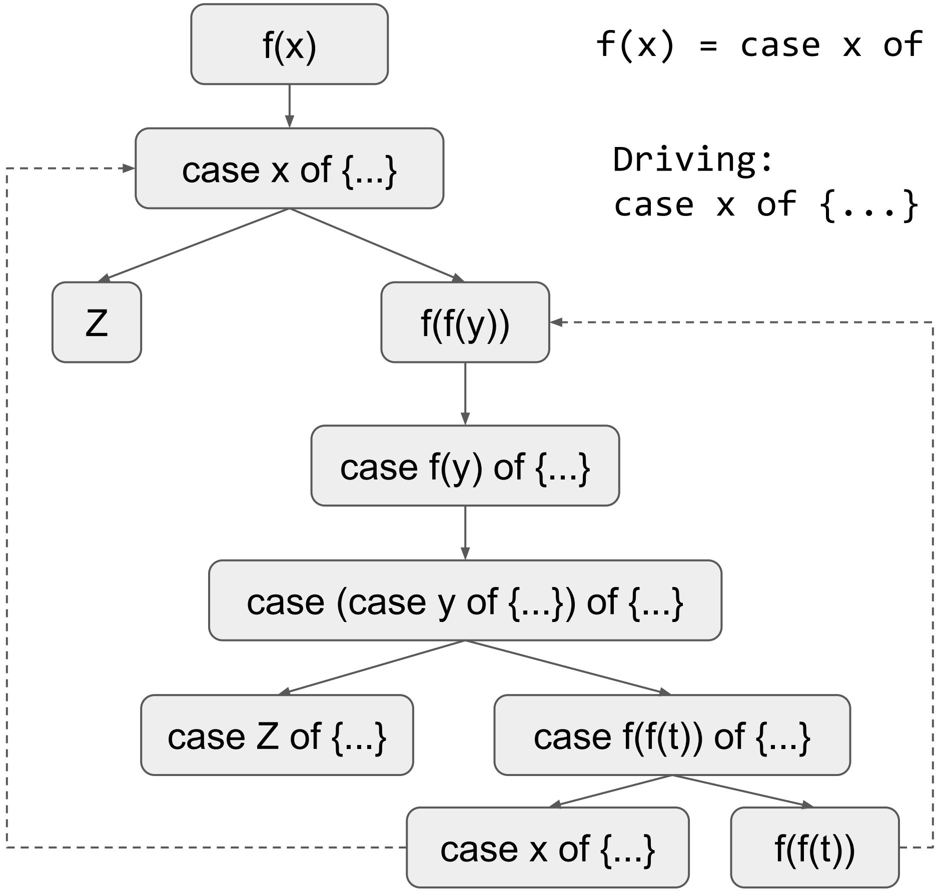
Input language: a first-order lazy functional Haskell-like language.

Task: proving equalities.

```
    x ++ (y ++ z) = (x ++ y) ++ z
```

(Originally I started this work to speed up multi-result supercompilation by sharing common subexpressions).

Supercompilation (briefly)



```
f(x) = case x of { Z -> Z; S(y) -> f(f(y)) }
```

```
Driving:
case x of {...} = case x of { Z -> Z ; S(y) -> f(f(y)) }
```

```
Driving (unfolding):
f(f(y)) = case f(y) of {...}
```

```
Driving:
case (case …) =
  case y of {Z -> …;
             S(t) -> case f(f(t)) of …}
```

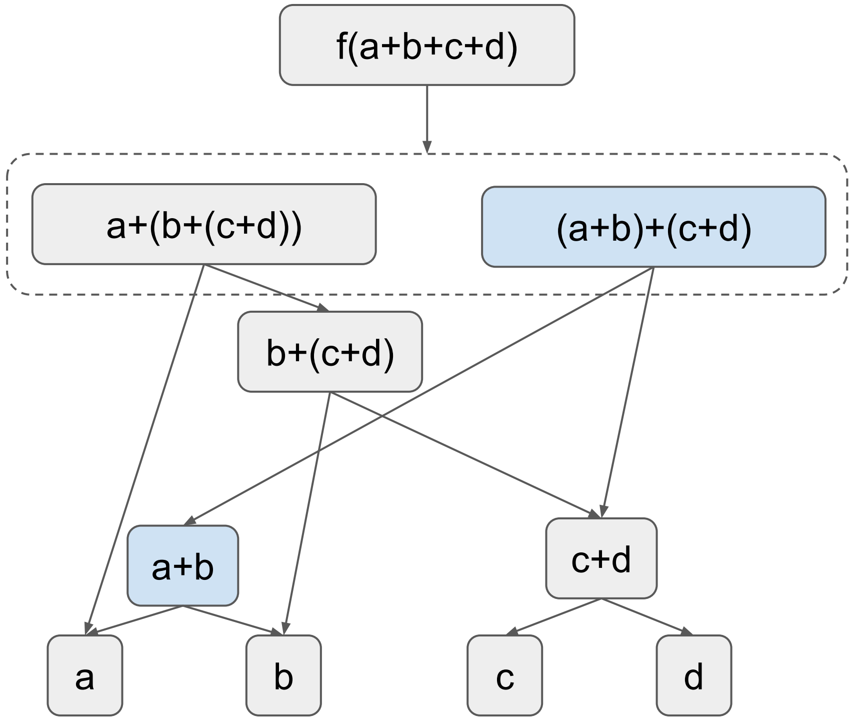
```
Folding: E1 = E2<E1>
then E1 = g(x) where g(x) = E2<g>
```

Supercompilation (briefly)

- We build a graph of configurations using driving and generalization.
- Each node of the graph is an expression with free variables, edges (hyperedges actually) can be seen as equalities between expressions.
- We can stop when we encounter an expression we have seen before (up to variable renaming). This is called folding.
- In the end we can transform the graph into a program equivalent to the original one.
- What we do is controlled by heuristics (whistle, lgg). We can also explore multiple paths (multi-result supercompilation).
- The correctness of folding is tricky. It’s usually automatically correct, unless we do something fancy like applying lemmas. The correctness is usually proved using Sands’s theory of improvement or structural recursion.

Equality Saturation (briefly)

Main idea: represent expressions and equalities using E-graphs: graphs with an equivalence relation over nodes.



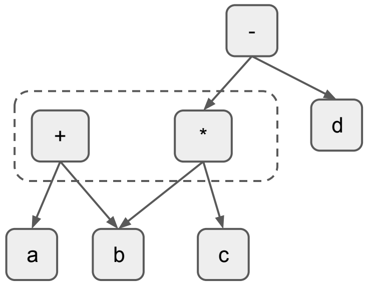
- We start with an expression, and then apply rewriting rules that add new expressions and equalities.
- We keep the relation congruently closed.
- In the end we can extract a residual program, but correctness is not automatic (it is a very hard part).

How to combine them?

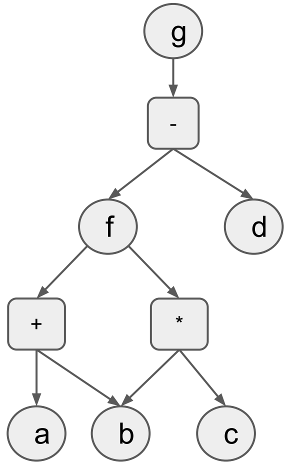
Supercompilation	Equality Saturation
Expressions (configurations)	Expressions (nodes of the E-graph)
Edges of the graph of configurations	Equalities
Driving	Rewriting rules
Generalization	Rewriting rules
Folding	Congruence closure
Multi-result supercompilation	Comes naturally
Higher-level supercompilation	Merging by bisimulation

E-graphs, Hypergraphs and Polyprograms

E-graph (equivalence classes, nodes, edges)



Hypergraph (nodes, hyperedges)



Polyprogram (functions, definitions)

```
g = f - d
f = a + b
f = b * c
```

Decomposed polyprograms

In decomposed polyprograms, each definition has the simplest form: a single language construct with trivial function calls as subexpressions.

Original:

```
  f(x) = case x of { Z -> Z; S(y) -> f(f(x)) }
```

Decomposed:

```
  f(x) = case v(x) of { Z -> z(); S(y) -> g(x) }
  v(x) = x
  z()  = Z
  g(x) = f(f(x))
```

This is to make them closer to E-graphs and hypergraphs.

It’s also convenient to consider definitions like this one decomposed:

```
  g(x, y) = f(y, x)
```

Equivalence up to variable permutation

I think it’s a very important trick

```
f(x, y) = a(x) / b(y)
g(x, y) = a(y) / b(x)
```

We want to merge them, we don’t want a separate function for every possible permutation.

Rename variables canonically in the rhs:

```
f(x, y) = a(x) / b(y)
g(y, x) = a(x) / b(y)
```

Now we can replace f(A, B) with g(B, A) everywhere (for any subexpressions A and B)

Function calls as explicit substitutions

In decomposed polyprograms function calls work similarly to explicit substitutions.

Before expansion of f:

```
e(x) = f(g(x), h(x))
f(x, y) = r(s(x), t(y))
```

After expansion of f:

```
e(x) = f(g(x), h(x))
f(x, y) = r(s(x), t(y))
e(x) = r(s1(x), t1(x))
s1(x) = s(g(x))
t1(y) = t(h(x))
```

Destructive rules

In naive Equality Saturation rewriting rules are applied non-destructively (cannot delete definitions/hyperedges). Non-destructivity automatically gives us confluence.

However, the E-graph grows too quickly without destructive rules. In my dissertation I have a group of rules that are applied destructively (simplification rules, see page 92).

Destructive rules make the question of confluence harder, and unfortunately I didn’t have enough time for it.

```
f(x) = case g(x) of { Z -> …; S(y) -> h(y) }
g(x) = S(x)
```

$\Longrightarrow$

```
f(x) = h(x)
g(x) = S(x)
```

The list of simplification rules (1)

When a rule proves equivalence of two functions, it’s convenient to express this equivalence with an explicit definition and then merge the functions separately using the congruence rule.

Congruence:

$$\begin{array}{l} \mathtt{f}(x_1, \ldots, x_n) = \mathtt{g}(x_{p(1)}, \ldots, x_{p(n)}) \\ \mathtt{f(...) = ...} \\ \mathtt{e(...) = E<f>} \end{array} \Longrightarrow \begin{array}{l} \mathtt{g = g} \\ \mathtt{g(...) = ...} \\ \mathtt{e(...) = E<g>} \end{array}$$

Example:

$$\begin{array}{l} \mathtt{f(x,\ y) = g(y,\ x)} \\ \mathtt{e(x,\ y) = f(f(x,\ y),\ x)} \\ \mathtt{f(x,\ y) = E} \end{array} \Longrightarrow \begin{array}{l} \mathtt{g(y,\ x) = g(y,\ x)} \\ \mathtt{e(x,\ y) = g(x,\ g(y,\ x))} \\ \mathtt{g(y,\ x) = E} \end{array}$$

The list of simplification rules (2)

Transitivity (with symmetry):

$$\begin{array}{l} \texttt{f}(x_1, \ldots, x_n) = E \\ \texttt{g}(x_{p(1)}, \ldots, x_{p(n)}) = E \end{array} \Longrightarrow \begin{array}{l} \texttt{f}(x_1, \ldots, x_n) = E \\ \texttt{g}(x_{p(1)}, \ldots, x_{p(n)}) = E \\ \mathbf{f(x_1, \ldots, x_n) = g(x_{p(1)}, \ldots, x_{p(n)})} \end{array}$$

Arity reduction:

$$\texttt{f}(x, y) = E\langle x\rangle \Longrightarrow \texttt{f}'(x) = E\langle x\rangle$$

f(x, y) is replaced with f'(x) everywhere

Commutativity:

$$\begin{array}{l} \texttt{f}(x_1, \ldots, x_n) = \texttt{f}(x_{p(1)}, \ldots, x_{p(n)}) \\ \texttt{e}(\ldots) = E\langle \texttt{f}(x_1, \ldots, x_n)\rangle \end{array} \Longrightarrow \begin{array}{l} \texttt{f}(x_1, \ldots, x_n) = \texttt{f}(x_{p(1)}, \ldots, x_{p(n)}) \\ \texttt{e}(\ldots) = E\langle \texttt{f}(x_1, \ldots, x_n)\rangle \\ \mathbf{e(\ldots) = E\langle f(x_{p(1)}, \ldots, x_{p(n)})\rangle} \end{array}$$

We need a better theory instead of this rule

The list of simplification rules (3)

Call to permutation:

```
f(x, y) = g(v(y), v(x))
v(x) = x
```

$\Rightarrow$

```
f(x, y) = g(y, x)
v(x) = x
```

Call of a variable reduction:

```
f(x, y) = v(g(x, y))
v(x) = x
```

$\Rightarrow$

```
f(x, y) = g(x, y)
v(x) = x
```

Case-of reduction:

```
f(x) = case c(x) of {C(y) -> h(x, y)}
c(x) = C(g(x))
```

$\Rightarrow$

```
f(x) = h(x, g(x))
c(x) = C(g(x))
```

The list of simplification rules (4)

Injectivity of constructors:

```
f() = C(g(), h())
f() = C(g'(), h'())
```

$\Longrightarrow$

```
f() = C(g(), h())
f() = C(g'(), h'())
g() = g'()
h() = h'()
```

Injectivity of case-of:

```
f(x) = case v(x) of {C(y) -> h(x, y)}
f(x) = case v(x) of {C(y) -> h'(x, y)}
v(x) = x
```

$\Longrightarrow$

```
...
h(x, y) = h'(x, y)
```

On the order of rule application

```
Repeat until timeout or the equivalence is proved:
    * Apply "saturating" rules non-destructively, but only to the existing definitions
    * Repeat until convergence (simplification):
        - Apply until convergence: transitivity, arity reduction, call to permutation, call of a variable, case-of reduction, injectivities
        - Apply until convergence: congruence and commutativity
    * Merging by bisimulation
```

“Saturating” rules are non-destructive driving-like rules:

- The remaining call reductions (call-call, call-caseof, call-cons)
- Positive information propagation
- Pushing outer case-of into the inner one
- Case-of reordering

Saturating rules

Call reduction (several rules):

```
f(x) = g(h(x), k(x))
g(x, y) = e(s(x, y), t(x, y))
=> f(x) = e(s’(x), t’(x))
=> s’(x) = s(h(x), k(x))
=> t’(x) = t(h(x), k(x))
```

Case-of pushing:

```
f(x) = case g(x) of {C(y) -> …}
g(x) = case h(x) of {D(y) -> s(x, y)}
=> f(x) = case h(x) of {D(y) -> s’(x, y)}
=> s’(x, z) = case s(x, z) of {C(y) -> …}
```

Positive information propagation:

```
f(x) = case v(x) of {C(y) -> g(x, y)}
v(x) = x
=> f(x) = case v(x) of {C(y) -> g’(y)}
=> g’(y) = g(c(y), y)
=> c(y) = C(y)
```

Case-of reordering:

```
f(x) = case e(x) of {C(y) -> g(x, y)}
g(x, y) = case h(x) of {D(z) -> …}
=> f(x) = case h(x) of {D(z) -> s(x, z)}
=> s(x, z) = case e(x) of {C(y) -> …}
```

Merging by bisimulation

Merging by bisimulation is like a recursive congruence

Congruence: $$\frac{a = E \quad b = E}{a = b}$$

“Recursive congruence”: $$\frac{a = E(a) \quad b = E(b) \quad E \text{ is “good”}}{a = b}$$

Merging by bisimulation: $$\frac{a = E(a) \quad b = F(b) \quad E \text{ and } F \text{ are bisimilar and “good”}}{a = b}$$

The term bisimulation here means a polyprogram bisimulation (much simpler than an LTS bisimulation).

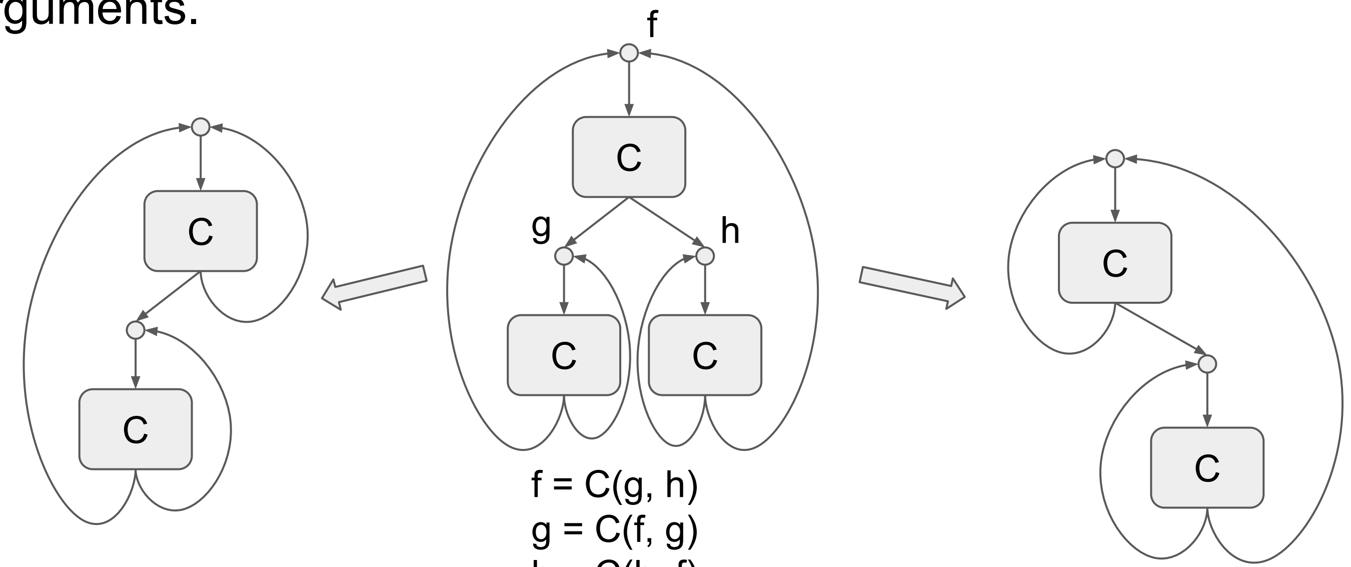
“Good” means there is a single model, which can be guaranteed using structural and guarded recursion.

Polyprogram bisimulation

It’s just a span of polyprogram morphisms:

φ, ψ : B → P

Polyprogram morphism are like hypergraph morphisms, but can also rearrange function arguments.



f = C(g, f)
g = C(f, g)

f = C(g, h)
g = C(f, g)
h = C(h, f)

f = C(f, h)
h = C(h, f)

Conclusion

What was done:

- Driving was expressed via polyprogram rewriting rules.
- Termination of rewriting rules was proved (not pretty, there are restrictions on the application order).
- Merging by bisimulation was introduced, with guarded and structural recursion as correctness criteria (correctness was proved).
- Equivalence up to variable permutation.

What wasn’t done:

- I couldn’t develop a pretty theory of polyprogram rewriting.
- Couldn’t prove confluence.
- Generalizations are very limited (only trivial generalizations that don’t need rewriting + some weird side effects of one of the driving rules).
- It’s not very good in practice, comparing to HipSpec.

Possible directions for future work

- Ticks (Sands's theory).
- Building residual programs.
- Execution on E-graphs/polyprograms.
- Proper support for higher-order functions.
- Generalizations (no idea how).
- Heuristics, ML to guide rewriting.
- User-specified rewriting rules/lemmas.

Some references

- The dissertation itself (in Russian): Grechanik S. **“Proving Properties of Functional Programs by Equality Saturation”**
- Grechanik S. **“Polyprograms and Polyprogram Bisimulation”** (introduction to polyprograms, in English)
- Tate R. et al **“Equality saturation: a new approach to optimization”**
- Ticks and the improvement theory:
  Sands D. **“Total correctness by local improvement in the transformation of functional programs”**
  Sands D. **“Proving the correctness of recursion-based automatic program transformations”**
- Multi-result supercompilation: Klyuchnikov I. G., Romanenko S. A. **“Multi-Result Supercompilation as Branching Growth of the Penultimate Level in Metasystem Transitions”**
- Higher-level supercompilation:
  Klyuchnikov I., Romanenko S. **“Towards Higher-Level Supercompilation”**
  Klyuchnikov I. **“Towards effective two-level supercompilation”**
- Distillation: Hamilton G. W., Jones N. D. **“Distillation with labelled transition systems”**
- Hypergraph rewriting: Plump D. **“Evaluation of Functional Expressions by Hypergraph Rewriting”**: PhD thesis