Inductive Prover Based on Equality Saturation for a Lazy Functional Language (Extended Version)*

Sergei Grechanik

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
sergei.grechanik@gmail.com

Abstract. The present paper shows how the idea of equality saturation can be used to build an inductive prover for a non-total first-order lazy functional language. We adapt equality saturation approach to a functional language by using transformations borrowed from supercompilation. A special transformation called merging by bisimilarity is used to perform proof by induction of equivalence between nodes of the E-graph. Equalities proved this way are just added to the E-graph. We also experimentally compare our prover with HOSC, HipSpec and Zeno.

1 Introduction

Equality saturation [23] is a method of program transformation that uses a compact representation of multiple versions of the program being transformed. This representation is based on E-graphs (graphs whose nodes are joined into equivalence classes [7, 18]) and allows us to represent a set of equivalent programs, consuming exponentially less memory than representing it as a plain set. Equality saturation consists in enlarging this set of programs by applying certain axioms to the E-graph until there's no axiom to apply or the limit of axiom applications is reached. The axioms are applied non-destructively, i.e. they only add information to the E-graph (by adding nodes, edges and equivalences).

Equality saturation has several applications. It can be used for program optimization – in this case after the process of equality saturation is finished, a single program should be extracted from the E-graph. It can also be used for proving program equivalence (e.g. for translation validation [22]) – in this case program extraction is not needed.

In the original papers equality saturation is applied to imperative languages, namely Java bytecode and LLVM (although the E-graph-based intermediate representation used there, called E-PEG, is essentially functional). In this paper we describe how equality saturation can be applied to the task of proving equivalence

* Supported by Russian Foundation for Basic Research grant No. 12-01-00972-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

of functions written in a lazy functional language, which is important for proving algebraic properties like monad laws or some laws concerning natural numbers and lists. We do this mainly by borrowing transformations from supercompilation [21, 24]. Since many properties require proof by induction, we introduce a special transformation called merging by bisimilarity which essentially proves by induction that two terms are equivalent. This transformation may be applied repeatedly, which gives an effect of discovering and proving lemmas needed for the main goal.

Unlike tools such as HipSpec [5] and Zeno [20], we don't instantiate the induction scheme, but instead check the correctness of the proof graph similarly to Agda and Foetus [3, 4]. We also fully support infinite data structures and partial values, and we don't assume totality. As we'll show, proving properties that hold only in total setting is still possible with our tool by enabling some additional transformations, but it's not very efficient.

The main contributions of this paper are: 1) we apply the equality saturation approach to a lazy functional language; 2) we propose to merge classes of the E-graph even if they represent functions equal only up to argument permutation; 3) we articulate the merging by bisimilarity transformation.

The paper is organized as follows. In Section 2 we briefly describe equality saturation and how functional programs and sets of functional programs can be represented by E-graphs. Then in Section 3 we discuss basic transformations which we apply to the E-graph. Section 4 deals with the merging by bisimilarity transformation. Section 5 discusses the order of transformation application. In Section 6 we consider a simple example. In Section 7 we present experimental evaluation of our prover. Section 8 discusses related work and Section 9 concludes the paper.

The source code of our experimental prover can be found on GitHub [1].

2 Programs and E-graphs

An E-graph is a graph enriched with information about equivalence of its nodes by means of splitting them into equivalence classes. In our case, an E-graph essentially represents a set of (possibly recursive) terms and a set of equalities on them, closed under reflexivity, transitivity and symmetry. If we use the congruence closure algorithm [18], then the set of equalities will also be closed under congruence. The E-graph representation is very efficient and often used for solving the problem of term equivalence.

If we have some axioms about our terms, we can also apply them to the E-graph, thus deducing new equalities from the ones already present in E-graph (which in its turn may lead to more axiom application opportunities). This is what equality saturation basically is. So, the process of solving the problem of function/program equivalence using equality saturation is as follows:

1. Convert both function definitions to E-graphs and put both of them into a single E-graph.

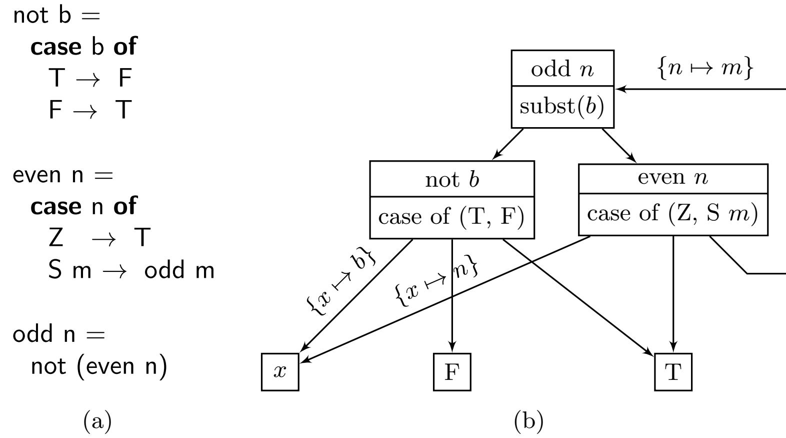

Fig. 1: A program and its graph representation

2. Transform the E-graph using some axioms (transformations) until the target terms are in the same equivalence class or no more axioms are applicable. This process is called saturation.

In pure equality saturation approach axioms are applied non-destructively and result only in adding new nodes and edges, and merging of equivalence classes, but in our prover we apply some axioms destructively, removing some nodes and edges. This makes the result of the saturation dependent on the order of axiom application, so we restrict it to breadth-first order (see Section 5 for more details). This deviation is essential for performance reasons.

In this paper we will use a lazy first-order untyped subset of Haskell (in our implementation higher-order functions are dealt with by defunctionalization). To illustrate how programs are mapped into graphs, let's consider the program in Figure 1a. This program can be naturally represented as a graph, as shown in Figure 1b. Each node represents a basic language construct (pattern matching, constructor, variable, or explicit substitution – we'll explain them in Section 2.1). If a node corresponds to some named function, its name is written in the top part of it. Some nodes are introduced to split complex expressions into basic constructs and don't correspond to any named functions. Recursion is simply represented by cycles. Some nodes are shared (in this example these are the variable x and the constructor T). Sharing is very important since it is one of the things that enable compactness of the representation.

Some of the edges are labeled with renamings. Actually, all edges are labeled with renamings, but identity renamings are not drawn. These renamings are very important – without them we would need a separate node for each variable, and we couldn't merge nodes representing the same function modulo renaming, which would increase space consumption (such functions often appear during transformation). Merging up to renaming will be discussed in Section 2.2.

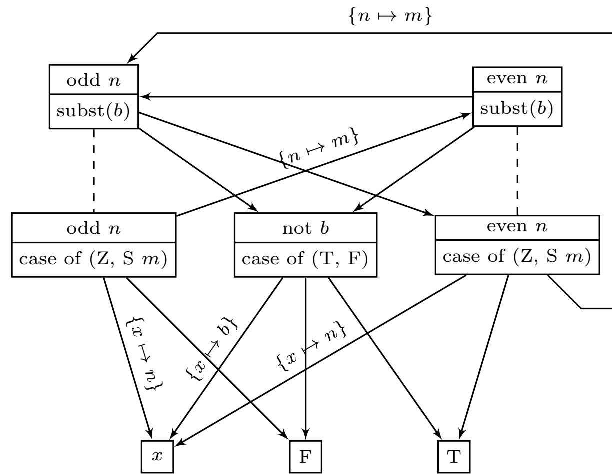

Fig. 2: E-graph representing functions even and odd

Note also that we use two methods of representing function calls. If all the arguments are *distinct* variables, then we can simply use a renaming (the function odd is called this way). If the arguments are more complex, then we use explicit substitution [2], which is very similar to function call but has more fine-grained reduction rules. We can use explicit substitutions even if the arguments are distinct variables, but it's more expensive than using renamings (and actually we have an axiom to transform such explicit substitutions to renamings). Note that we require an explicit substitution to bind *all* variables of the expression being substituted.

The same way graphs naturally correspond to programs, E-graphs naturally correspond to programs with multiple function definitions. Consider the following "nondeterministic" program:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
odd n = case n of { Z → F; S m → even m }

odd n = not (even n)
even n = not (odd n)
```

This program contains multiple definitions of the functions even and odd, but all the definitions are actually equivalent. This program can also be represented as a graph, but there will be multiple nodes corresponding to functions even and odd. If we add the information that nodes corresponding to the same function are in the same equivalence class, we get an E-graph. The E-graph corresponding to the above program is shown in Figure 2. Nodes of equivalent functions are connected with dashed lines, meaning that these nodes are in the same class of

equivalence. As can be seen, the drawing is messy and it's hard to understand what's going on there, so we'll mostly use textual form to describe E-graphs.

E-graphs are also useful for representing compactly sets of equivalent programs. Indeed, we can extract individual programs from an E-graph or a nondeterministic program by choosing a single node for each equivalence class, or in other words, a single definition for each function. However, we cannot pick the definitions arbitrarily. For example, the following program isn't equivalent to the one above:

```
not b = case b of { T → F; F → T }

odd n = not (even n)
even n = not (odd n)
```

This problem should be taken into account not only when performing program extraction, but also during certain complex transformations like merging by bisimilarity which we will discuss in Section 4.

2.1 Node labels

In this section we'll discuss how node labels correspond to language constructs.

First of all, each node of an E-graph is a member of some equivalence class. We will use symbols $f, g, h, \ldots$ to denote nodes as well as corresponding functions. Each node has a label $L(f)$ and a set of input variables $V(f)$ (in the implementation variables are numbered, but in this paper we treat them as named). $V(f)$ may decrease with graph evolution, and it should be kept up to date because we need $V(f)$ to perform some transformations (keeping it up to date is beyond the scope of this paper). Each edge of an E-graph is labeled with an injective renaming, its domain being the set of input variables of the edge's destination node. We will use the notation $f = L \to \theta_1 g_1, \ldots, \theta_n g_n$ to describe a node f with a label L and outgoing edges with renamings θ_i and destinations g_i. We will write $f \cong g$ to denote that f and g are from the same equivalence class.

There are only four kinds of node labels. We give a brief description for each of them and some code examples:

- $f = x$. (Variable / identity function). We use the convention that the identity function always takes the variable x, and if we need some other variable, we adjust it with a renaming. Code example: `f x = x`
- $f = \text{subst}(x_1, \ldots, x_n) \to \xi h, \theta_1 g_1, \ldots, \theta_n g_n$. (Explicit substitution / function call / let expression). An explicit substitution substitutes values $\theta_i g_i$ for the variables x_i in ξh. We require it to bind *all* the variables of ξh. Explicit substitution nodes usually correspond to function calls:

  ```
  f x y = h (g1 x) (g2 y) (g3 x y)
  ```

  They may also correspond to non-recursive let expressions, or lambda abstractions immediately applied to the required number of arguments:

```
f x y = let { u = g1 x; v = g2 y; w = g3 x y } in h u v w
      = (λ u v w . h u v w) (g1 x) (g2 y) (g3 x y)
```

But to describe E-graph transformations we will use the following non-standard (but hopefully more readable) postfix notation:

```
f x y = h u v w { u = g1 x, v = g2 y, w = g3 x y }
```

– $f = C \to \theta_1 g_1, \ldots, \theta_n g_n$. (Constructor). Code example:

```
f x y = C (g1 x) (g2 y) (g3 x y)
```

– $f = \text{caseof}(C_1\overline{x_1}, \ldots, C_n\overline{x_n}) \to \xi h, \theta_1 g_1, \ldots, \theta_n g_n$. (Pattern matching). This label is parametrized with a list of patterns, each pattern is a constructor name and a list of variables. The corresponding case bodies ($\theta_i g_i$) don't have to use all the variables from the pattern. ξh represents the expression being scrutinized. Code example:

```
f x y = case h x of
            S n →  g1 y n
            Z →  g2 x
```

We will also need an operation of adjusting a node with a renaming. Consider a node $f = L \to \theta_1 g_1, \ldots, \theta_n g_n$ and a renaming ξ. Suppose, we want to create a function $f' = \xi f$ (f' is f with parameters renamed). We can do this by adjusting outgoing edges of f with ξ (unless $f = x$ in which case it doesn't have outgoing edges). We will use the following notation for this operation:

$$f' = \xi(L \to \theta_1 g_1, \ldots, \theta_n g_n)$$

The operation is defined as follows:

$$\begin{aligned}
&\xi(C \to \theta_1 g_1, \ldots, \theta_n g_n) = C \to (\xi \circ \theta_1) g_1, \ldots, (\xi \circ \theta_n) g_n \\
&\xi(\text{subst}(\ldots) \to \zeta h, \theta_1 g_1, \ldots, \theta_n g_n) = \\
&\qquad \text{subst}(\ldots) \to \zeta h, (\xi \circ \theta_1) g_1, \ldots, (\xi \circ \theta_n) g_n \\
&\xi(\text{caseof}(\ldots) \to \zeta h, \theta_1 g_1, \ldots, \theta_n g_n) = \\
&\qquad \text{caseof}(\ldots) \to (\xi \circ \zeta) h, (\xi'_1 \circ \theta_1) g_1, \ldots, (\xi'_n \circ \theta_n) g_n
\end{aligned}$$

In the last case each ξ'_i maps the variables bound by the ith pattern to themselves and works as ξ on all the other variables.

2.2 Merging

One of the basic operations of the E-graph is merging of equivalence classes. Usually it is done after applying axioms that result in adding new equalities between nodes. In the case of simple equalities like $f = g$ we should simply merge the corresponding equivalence classes. But we also want to merge functions which are equal only up to some renaming, so should take into account equalities

of the form $f = \theta g$ where θ is some non-identity renaming. In this case we should first adjust renamings on edges so that the equation becomes of the form $f = g$ and then proceed as usual.

Consider the equation $f = \theta g$. Let's assume that g is not a variable node (x) and it's not in the same equivalence class with a variable node (otherwise we can rewrite the equation as $g = \theta^{-1} f$, and if they both were equal to a variable node, then our E-graph would be self-contradictory). Now for each node h in the same equivalence class with g (including g) we should perform the following:

1. Adjust the outgoing edges of h with θ using previously described node adjustment operation.
2. For each edge incoming into h replace its renaming, say, ξ, with a renaming $\xi \circ \theta^{-1}$

After the adjustment the equation becomes $f = g$ and we can merge the equivalence classes.

Note that this procedure works if f and g aren't in the same equivalence classes. If they are, then the equation looks like $f = \theta f$ and should be modelled with an explicit substitution.

3 Axioms

3.1 Congruence

The most common cause of equivalence class merging is equivalence by congruence, that is if we know that $a = f(b)$, $c = f(d)$ and $b = d$, then we can infer that $a = c$. Note that usually this kind of merging is not explicitly formulated as an axiom, but we prefer to do it explicitly for uniformity. Also, in our case the axiom should take into account that we want to detect equivalences up to some renaming. Here is the axiom written as an inference rule, we will later refer to it as (cong):

$$\frac{f = L \to \theta_1 h_1, \dots, \theta_n h_n \qquad \exists \xi : g = \xi(L \to \theta_1 k_1, \dots, \theta_n k_n) \qquad \forall i \; h_i \cong k_i}{g = \xi f}$$

It says that if we have a node f and a node g that is equivalent to f adjusted with some renaming ξ, then we can add the equality $g = \xi f$ to the E-graph. This axiom is advantageous to apply as early as possible since it results in merging of equivalence classes, which reduces duplication and gives more opportunities for applying axioms. Also note that to make the search for the appropriate ξ faster, it is beneficial to represent nodes in normal form:

$$f = \zeta(L \to \theta_1 g_1, \dots, \theta_n g_n)$$

Where θ_i are as close to identity renamings as possible, so to find ξ we should just compare the ζ's.

3.2 Injectivity

This axiom may be seen as something like "inverse congruence". If we know that $a = f(b)$, $c = f(d)$ and $a = c$ and f is injective, then $b = d$. Of course, we could achieve the same effect by adding the equalities $b = f^{-1}(a)$ and $d = f^{-1}(c)$ to the E-graph and then using congruence, but we prefer a separate axiom for performance reasons. We will call it (inj):

$$\frac{f = L \to \xi(\theta_1 h_1, \dots \theta_n h_n) \qquad g = L \to \xi(\zeta_1 k_1, \dots \zeta_n k_n) \qquad f \cong g \qquad L \text{ is inj}}{\forall i \;.\; h_i = \theta_i^{-1} \zeta_i k_i}$$

"L is inj" means that L is either a constructor, or a case-of that scrutinizes a variable (i.e. $\theta_1 = \zeta_1$ and $h_1 = k_1 = x$) such that none of the $\theta_2 h_2, \dots, \theta_n h_n$, $\zeta_2 k_2, \dots, \zeta_n k_n$ uses this variable (in other words, positive information is propagated). This axiom is also advantageous to apply as early as possible.

3.3 Semantics of explicit substitutions

In this and the next sections we will write axioms in a less strict but more human-readable form. A rewriting rule $E_1 \mapsto E_2$ means that if we have a node f_1 representing the expression E_1, then we can add an equality $f_1 = f_2$ to the E-graph where f_2 is the node representing E_2 (which should also be added to the E-graph unless it's already there). We use the compact postfix notation to express explicit substitutions. We use letters $e, f, g, h, \dots$ to represent nodes whose structure doesn't matter. We sometimes write them applied to variables they use ($f\ x\ y$), but if variables don't really matter, we omit them. Note that the presented rules can be generalized to the case when pattern matchings have arbitrary number of branches and functions take arbitrary number of arguments, we just use minimal illustrative examples for the sake of readability.

In Figure 3 four axioms of explicit substitutions [2] are shown. All of them basically describe how to evaluate a node if it is an explicit substitution (using call-by-name strategy). The answer is to push the substitution down (the last three rules) until we reach a variable where we can just perform the actual substitution (the first rule, (subst-id)). The appropriate rule depends on the node we choose as the leftmost child of our substitution node – there are four kinds of nodes, so there are four rules.

(subst-id)	$x\ \{x = g\} \ \mapsto\ g$
(subst-subst)	$f\ x\ \{x = g\ y\}\ \{y = h\} \ \mapsto\ f\ x\ \{x = g\ y\ \{y = h\}\}$
(subst-constr)	$C\ (f\ x)\ (g\ x)\ \{x = h\} \ \mapsto\ C\ (f\ x\ \{x = h\})\ (g\ x\ \{x = h\})$
(subst-case-of)	$(\textbf{case}\ f\ x\ \textbf{of}\ C\ y \to g\ x\ y)\ \{x = h\} \ \mapsto$ $\textbf{case}\ f\ x\ \{x = h\}\ \textbf{of}\ C\ y \to g\ x\ y\ \{x = h,\ y = y\}$

Fig. 3: Axioms of explicit substitutions

$$
\begin{array}{rl}
\text{(case-of-constr)} & (\textbf{case}\ \ C\ e\ \ \textbf{of}\ C\ y \to f\ x\ y) \ \mapsto \\
& \quad f\ x\ y\ \{x = x,\ y = e\} \\
\text{(case-of-case-of)} & (\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ C_1\ y \to g)\ \textbf{of}\ C_2\ z \to h) \ \mapsto \\
& \quad \textbf{case}\ e\ \textbf{of}\ C_1\ y \to (\textbf{case}\ g\ \textbf{of}\ C_2\ z \to h) \\
\text{(case-of-id)} & (\textbf{case}\ x\ \textbf{of}\ C\ y\ z \to f\ x\ y\ z) \ \mapsto \\
& \quad \textbf{case}\ x\ \textbf{of}\ C\ y\ z \to f\ x\ y\ z\ \{x = (C\ y\ z),\ y = y,\ z = z\} \\
\text{(case-of-transpose)} & \textbf{case}\ h\ \textbf{of}\ \{ \\
& \quad C_1\ x \to \textbf{case}\ z\ \textbf{of}\ D\ v \to f\ v\ x; \\
& \quad C_2\ y \to \textbf{case}\ z\ \textbf{of}\ D\ v \to g\ v\ y; \\
& \} \ \mapsto \\
& \quad \textbf{case}\ z\ \textbf{of}\ D\ v \to \\
& \qquad \textbf{case}\ h\ \textbf{of}\ \{ \\
& \qquad\quad C_1\ x \to f\ v\ x; \\
& \qquad\quad C_2\ y \to g\ v\ y; \\
& \qquad \}
\end{array}
$$

Fig. 4: Axioms of pattern matching

Usually substitution in the body of a function is performed as an indivisible operation, but this kind of transformation would be too global for an E-graph, so we use explicit substitutions to break it down.

There are two more rather technical but nonetheless important axioms concerning substitution. The first one is elimination of unused variable bindings:

$$\text{(subst-unused)} \qquad f\ x\ y\ \{x = g,\ y = h,\ z = k\} \ \mapsto\ f\ x\ y\ \{x = g,\ y = h\}$$

When this axiom is applied *destructively* (i.e. the original node is removed), it considerably simplifies the E-graph. This axiom is the reason why we need the information about used variables in every node.

The second axiom is conversion from a substitution that substitutes variables for variables to a renaming:

$$\text{(subst-to-renaming)} \qquad f\ x\ y\ \{x = y,\ y = z\} \ \mapsto\ f\ y\ z$$

This axiom requires the original substitution to be injective. Note also that application of this axiom results in merging of the equivalence classes corresponding to the node representing the substitution and the node f, so if they are already in the same class, this axiom is inapplicable. We also apply this axiom destructively.

3.4 Semantics of pattern matching

The axioms concerning pattern matching are shown in Figure 4. The first of them, (case-of-constr), is essentially a reduction rule: if the scrutinee is an expression starting with a constructor, then we just substitute appropriate subexpressions into the corresponding case branch.

The next two axioms came from supercompilation [21,24]. They tell us what to do when we get stuck during computation because of missing information (i.e. a variable). The axiom (case-of-case-of) says that if we have a pattern matching that scrutinizes the result of another pattern matching, then we can pull the inner pattern matching out. The axiom (case-of-id) is responsible for positive information propagation: if a case branch uses the variable being scrutinized, then it can be replaced with its reconstruction in terms of the pattern variables.

The last axiom, (case-of-transpose), says that we can swap two consecutive pattern matchings. This transformation is not performed by supercompilers and is actually rarely useful in a non-total language.

3.5 Totality

If we assume that our language is total, then we can use all the axioms mentioned above and also some more axioms that don't hold in the presence of bottoms. Although proving equivalence of total functions is not our main goal, our implementation has a totality mode which enables three additional axioms from Figure 5.

(case-of-constr-total)

$$
\begin{array}{l}
\textbf{case}\ h\ \textbf{of}\ \{ \\
\quad C_1\ x \to D\ (f\ x); \\
\quad C_2\ y \to D\ (f\ y); \\
\}\ \mapsto \\
\quad D\ (\textbf{case}\ h\ \textbf{of}\ \{ \\
\quad\quad C_1\ x \to f\ x; \\
\quad\quad C_2\ y \to f\ y; \\
\quad \})
\end{array}
$$

(case-of-transpose-total)

$$
\begin{array}{l}
\textbf{case}\ h\ \textbf{of}\ \{ \\
\quad C_1\ x \to \textbf{case}\ z\ \textbf{of}\ D\ v \to f\ v\ x\ z; \\
\quad C_2\ y \to g\ y; \\
\}\ \mapsto \\
\quad \textbf{case}\ z\ \textbf{of}\ D\ v \to \\
\quad\quad \textbf{case}\ h\ \textbf{of}\ \{ \\
\quad\quad\quad C_1\ x \to f\ v\ x\ z; \\
\quad\quad\quad C_2\ y \to g\ y; \\
\quad\quad \}
\end{array}
$$

(useless-case-of-total)

$$
\begin{array}{l}
\textbf{case}\ h\ \textbf{of}\ \{ \\
\quad C_1\ x \to f; \\
\quad C_2\ y \to f; \\
\}\ \mapsto \\
\quad f
\end{array}
$$

Fig. 5: Additional axioms of pattern matching in total setting

The axiom (case-of-constr-total) lifts equal constructors from case branches. If E could be bottom, then it wouldn't be correct to do that (actually the axiom makes the function lazier). Note that constructors may have arbitrary arity.

The axiom (case-of-transpose-total) is a variation of the axiom (case-of-transpose). It may swap the pattern matchings even if inner pattern matching is not performed in some branches of the outer one.

The axiom (useless-case-of-total) removes an unnecessary pattern matching when all of its branches are equal (they can't use pattern variables (x and y in this case) though).

3.6 On correctness and completeness

Correctness of the mentioned axioms can be easily proved if we fix an appropriate semantics for the language.

Since the problem of function equivalence is undecidable, no finite set of axioms can be complete, but we can compare our set of axioms with other transformers. If we take all the axioms from Figure 3 and the axiom (case-of-constr) from Figure 4, we will be able to perform interpretation. If we also add axioms (case-of-case-of) and (case-of-id) from Figure 4, then we will be able to perform driving (interpretation with incomplete information, i.e. with free variables). Given infinite time, driving allows us to build a perfect tree for a function (which is something like an infinite tabular representation of a function). Perfect trees consist of constructors, variables and pattern matchings on variables with positive information fully propagated. Perfect trees aren't unique, some functions may have multiple perfect trees, and the (case-of-transpose) axiom is used to mitigate this problem (although not eliminate it completely). The totality axioms (Figure 5) equate even more perfect trees by rearranging their nodes.

Of course, we could add more axioms. For example, in the total case we could use an axiom to lift pattern matchings through explicit substitutions, not only other pattern matchings. Or we could add generalizations which are used in supercompilers. All of this would make our equality saturator more powerful but at the cost of lower performance. So this is all about balance. As for the generalization, in the case of equality saturation expressions are already in generalized state, and we can transform any subexpression of any expression. It's not a complete solution to the problem of generalization since it's only equal to peeling the outer function call from an expression, but it still allows us to solve many examples that can't be solved with driving alone.

Another issue is proof by induction or coinduction. In supercompilers coinduction is implicitly applied when we build a residual program. Higher level supercompilers and inductive provers are able to apply (co)induction several times, thus proving the lemmas needed to prove the target proposition. In our equality saturator (co)induction is implemented as a special transformation called merging by bisimilarity which is discussed in Section 4.

3.7 Axioms applied destructively

We apply some transformations destructively, i.e. remove the original nodes and edges that triggered the transformation. It is a deviation from pure equality saturation approach, but it is a necessary one. Currently the transformations we apply destructively are (subst-id), (subst-unused), (subst-to-renaming), and (case-of-constr). We have tried to switch on and off their destructivity. Turned out that non-destructive (case-of-constr) leads to a lot of failures on our test suite (due to timeouts), but helps to pass one of the tests that cannot be passed when it's destructive (which is expected: non-destructive transformations are strictly more powerful when there is no time limit). Non-destructive (subst-unused) has a similar effect: it helps to pass two different tests, but at the price of failing several other tests. At last, non-destructivity of (subst-id) and (subst-to-renaming) doesn't impede the ability of our tool to pass tests from our test suite but when either of them is applied non-destructively, our tool becomes about 15% slower. We also tried to make *all* the mentioned transformations non-destructive, which rendered our tool completely unusable because of combinatorial explosion of the E-graph, which substantiates the importance of at least some destructivity.

4 Merging by bisimilarity

The axiom of congruence can merge two functions into one equivalence class if they have the same tree representation. But if their definitions involve separate (but equal) cycles, then the congruence axiom becomes useless. Consider the following two functions:

```
f = S f
g = S g
```

If they aren't in the same equivalence class in the first place, none of the already mentioned axioms can help us equate them. Here we need some transformation that is aware of recursion. Note that in the original implementation of equality saturation called Peggy [23] there is such a transformation that merges θ-nodes, but it doesn't seem to be published anywhere and it is much less powerful than the one described here.

The general idea of this kind of transformation is to find two bisimilar subgraphs growing from the two given nodes from different equivalence classes and merge these equivalence classes if the subgraphs have been found. Note though that not every subgraph is suitable. Consider the following nondeterministic program:

```
f x = C;       g x = D
f x = f (f x); g x = g (g x)
```

The functions f and g are different but they both are idempotent, which is stated by the additional definitions, so we have two equal closed subgraphs "defining" the functions:

```
f x = f (f x)
g x = g (g x)
```

Of course, we cannot use subgraphs like these to decide whether two functions are equal, because they don't really define the functions, they just state that they have the property of idempotence. So we need a condition that guarantees that there is (semantically) only one function satisfying the subgraph.

In our implementation we employ the algorithm used in Agda and Foetus to check if a recursive function definition is structural or guarded [4]. These conditions are usually used in total languages to ensure termination and productivity, but we believe that they can be used to guarantee *uniqueness* of the function satisfying a definition in a non-total language with infinite and partial values, although a proof of this claim is left for future work. Informally speaking, in this case guarded recursion guarantees that there is data output between two consecutive recursive function calls, and structural recursion guarantees that there is data input between them (i.e. a pattern matching on a variable that hasn't been scrutinized before). It's not enough for function totality since the input data may be infinite, but it defines the behaviour of the function on each input, thus guaranteeing it to be unique.

Note that there is a subtle difference between subgraphs that may have multiple fixed points and subgraphs that have a single fixed point equal to bottom. Consider the following function "definition":

```
f x = f x
```

The least fixed point interpretation of this function is bottom. But there are other fixed points (actually, any one-argument function is a fixed point of this definition). Now consider the following function:

```
f x = eat infinity
infinity = S infinity
eat x = case x of { S y → eat y }
```

The definition of infinity is guardedly recursive, and the definition of eat is structurally recursive. The least fixed point interpretation of the function f is still bottom but now it is guaranteed to be the only interpretation.

Of course, this method of ensuring uniqueness may reject some subgraphs having a single fixed point, because the problem is undecidable in general. Note also that this is not the only possible method of ensuring uniqueness. For example, we could use ticks [19] as in two-level supercompilation [15]. Ticks are similar to constructors but have slightly different properties, in particular they cannot be detected by pattern matching. Tick transformations could be encoded as axioms for equality saturation.

4.1 Algorithm description

In this subsection we'll describe the algorithm that we use to figure out if two nodes have two bisimilar subgraphs growing from them and meeting the uniqueness condition. First of all, the problem of finding two bisimilar subgraphs is a

```
function MERGE-BY-BISIMILARITY(m, n)
    if BISIMILAR?(m, n, ∅) then MERGE(m, n)

function BISIMILAR?(m, n, history)
    if m ≅ n then return true
    else if ∃(m′, n′) ∈ history : m′ ≅ m ∧ n′ ≅ n then
        if the uniqueness condition is met then
            return true
        else
            return false
    else if m and n are incompatible then return false
    else
        for (m′, n′) : m′ ≅ m ∧ n′ ≅ n ∧ label(m′) = label(n′) do
            children_pairs = zip(children(m′), children(n′))
            if length(children(m′)) = length(children(n′))
               and ∀ (m″, n″) ∈ children_pairs
                       BISIMILAR?(m″, n″, {(m′, n′)} ∪ history)  then
                return true
        return false
```

Fig. 6: Merging by bisimilarity

variation of the subgraph bisimulation problem which is NP-complete [8]. In certain places we trade completeness for performance (so sometimes our algorithm fails to find the subgraphs when they exist), but merging by bisimilarity is still one of the biggest performance issues in our experimental implementation. The merging by bisimilarity algorithm that we use is outlined in Figure 6. It checks (using the function BISIMILAR?) if there are two bisimilar subgraphs meeting the uniqueness condition, and if there are, merges the equivalence classes of the nodes. Checking for bisimilarity essentially consists in simultaneous depth-first traversal of the E-graph from the given nodes. This process resembles supercompilation.

The function BISIMILAR? works as follows. If the two nodes are equal, then they are bisimilar and we return true. If we encounter a previously visited pair of nodes (up to $\cong$), we check if the uniqueness condition holds, and if it does, we return true (this case corresponds to folding in supercompilation), and otherwise we stop trying and return false (this case doesn't guarantee that there's no bisimulation, but we do it for efficiency). This case also ensures termination of the algorithm since the number of nodes in the E-graph is finite, and they are bound to repeat at some point. Note that some kinds of uniqueness conditions have to be checked after the whole bisimulation is known (and the guardedness and structurality checker is of this kind since it needs to know all the recursive call sites). In this case it is still advantageous to check some prerequisite condition while folding, which may be not enough to guarantee correctness, but enough to filter out obviously incorrect graphs.

If neither of the two previous cases is applicable, we check if the two nodes are at least compatible (again, for efficiency reasons, we could do without it in theory). That means that there are no nodes equal to them that have incompatible labels, like different constructors or a constructor and a pattern matching on a variable. If the nodes are compatible, we go on and check all pairs of nodes equivalent to the original ones. If there is a pair of nodes such that their children are bisimilar, then the original pair is bisimilar.

We can extract the actual bisimulation by collecting all the node pairs on which we return true (we will call this relation R). We can also extract the two bisimilar subgraphs (actually, E-subgraphs) by taking the corresponding elements of these pairs (either left or right) and outgoing edges for those nodes that occurred bisimilar to some other nodes because their children were bisimilar. Indeed, the roots of these two subgraphs are in R (up to $\cong$) since the function BISIMILAR? returned true, and each pair of nodes from R is either a pair of equivalent nodes (in which case their outgoing edges are not included in the subgraph) or a pair of nodes with equal labels such that their pairs of children are in R (up to $\cong$). This substantiates the name of this transformation. And again we emphasize that the existence of two bisimilar subgraphs proves that the nodes are equivalent only if they meet the uniqueness condition.

Note that in our description of the algorithm we ignored the question of renamings. We did it for the sake of brevity, and actually (since we want to merge nodes even if they are equal only up to some renaming) we should take them into account which complicates the real implementation a little bit.

5 On order of transformation

Our experimental implementation of an equivalence prover for a first-order lazy language based on equality saturation is written in Scala and can be found on GitHub [1].

In our implementation we deviate from pure equality saturation for practical reasons. Pure equality saturation approach requires all transformations to be monotone with respect to the ordering $\sqsubseteq$ on E-graphs where $g_1 \sqsubseteq g_2$ means that the set of equalities encoded by g_1 is a subset of the corresponding set for g_2. Moreover, it requires them to be applied non-destructively, i.e. $g \sqsubseteq t(g)$ for each transformation t (in other words, we cannot remove nodes and edges, and split equivalence classes). But in return we are granted with a nice property: if we reach the fully saturated state (no transformation can change the E-graph further), then the resulting E-graph will be the same no matter in what order we have applied the transformations.

Unfortunately, in reality this is not very practical. First of all, saturation can never be reached if our axioms are complex enough (or at least it will take too long). In particular, the axioms we described above can be applied indefinitely in most cases. To solve this problem we should limit the axiom application. We can do this either by sacrificing randomness of the order of axiom application and simply limiting the total number of applications, or by using some limiting

monotone preconditions (similar to whistles in supercompilers), e.g. limiting the depth of the nodes to which axioms may be applied.

Second, if we always apply axioms non-destructively, we may find ourselves with an E-graph littered with useless garbage. But applying axioms destructively makes the randomness of axiom applications questionable, to say the least. Of course, the system may preserve the nice property of ordering independence, but it may be much harder to prove.

All in all, more or less deterministic order of transformation seems very desirable in practice. In our implementation we use the following order:

1. Transform the programs into an E-graph.
2. Apply all possible non-destructive transformations except merging by bisimilarity, congruence and injectivity to the equations that are already in E-graph but not to the equations that are added by the transformations performed in this step. This can be done by postponing the effects of transformations: first, we collect the effects of applicable transformations (nodes and edges to add, and classes to merge), then we apply all these effects at once.
3. Perform E-graph simplification: apply congruence, injectivity and destructive transformations to the E-graph until the saturation w.r.t. these transformations is reached. It is quite safe since all these transformations are normalizing in nature (i.e. they simplify the E-graph).
4. Perform merging by bisimilarity over each pair of equivalence classes. Pairs of equivalence classes are sorted according to resemblance of their components, and then the merging by bisimilarity algorithm is applied to them sequentially. After each successful merge perform E-graph simplification exactly as in the previous step.
5. Repeat steps 2–5 until the goal is reached.

This way E-graph is being built in a breadth-first manner, generation by generation, each generation of nodes and edges results from applying transformations to the nodes and edges of the previous generations. An exception from this general rule is a set of small auxiliary (but very important) transformations consisting of congruence, injectivity and all the destructive transformations which are applied until the saturation because they always simplify the E-graph.

6 Example

In this section we'll discuss a simple example to illustrate how the transformations described earlier work in practice. Consider the following program:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
odd n = case n of { Z → F; S m → even m }

evenSlow n = case n of { Z → T; S m → oddSlow m }
oddSlow n = not (evenSlow n)
```

It defines functions that check if a natural number is odd or even. The functions even and odd are defined efficiently using tail recursion, but the functions evenSlow and oddSlow will need linear amount of memory during the execution. Still, they are semantically equivalent, so we want to prove that even = evenSlow.

We will follow the scheme from the previous section. The program above correspond to the initial state of the E-graph and constitutes the zeroth generation. Now we should apply all applicable transformations to it. The only application that produces something new after simplification is of the transformation (subst-case-of) to the nodes oddSlow n = not (evenSlow n) and not b = **case** b **of** {...}. Indeed, it produces

```
oddSlow n = case (n { n = evenSlow n }) of { T → F; F → T }
```

which is immediately simplified by destructive application of (subst-id) to

```
oddSlow n = case (evenSlow n) of { T → F; F → T }
```

Actually this sequence of transformation is just expansion of the function not. This new definition of oddSlow appears in the E-graph alongside with the old definition of oddSlow. The current state of the E-graph is the first generation.

Now it is possible to apply the transformation (case-of-case-of) to the nodes oddSlow n = **case** (evenSlow n) **of** {...} and evenSlow n = **case** n **of** {...} which after simplification with (case-of-constr) gives the following definition:

```
oddSlow n = case n of { Z → F; S m → evenSlow2 m }
evenSlow2 m = case (oddSlow m) of { T → F; F → T }
```

Here evenSlow2 is an auxiliary function which is actually equal to evenSlow, but we don't know that yet. The current state of the E-graph is the second generation.

Now we apply (case-of-case-of) to these last two definitions which give us the following:

```
evenSlow2 n = case n of { Z → T; S m → oddSlow2 m }
oddSlow2 m = case (evenSlow2 m) of { T → F; F → T }
```

Again, we had to introduce a new function oddSlow2 which will turn out to be equal to oddSlow.

We should also apply (case-of-case-of) to the same definition of evenSlow2 and a different definition of oddSlow, namely oddSlow n = **case** (evenSlow n) **of** {...}, which gives us

```
evenSlow2 m = case (evenSlow m) of { T → T; F → F }
```

Although from this definition it is quite obvious that evenSlow2 = evenSlow, it is of no use to us: since our internal representation is untyped, evenSlow may return something different from T and F, and the fact that it can't should be proved by induction. Instead, other definitions will be used to show by induction that this equivalence holds.

First of all, let's see what the E-graph currently looks like:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
odd n = case n of { Z → F; S m → even m }

evenSlow n = case n of { Z → T; S m → oddSlow m }

oddSlow n = not (evenSlow n)
oddSlow n = case (evenSlow n) of { T → F; F → T }
oddSlow n = case n of { Z → F; S m → evenSlow2 m }

evenSlow2 n = case n of { Z → T; S m → oddSlow2 m }
evenSlow2 m = case (oddSlow m) of { T → F; F → T }
evenSlow2 m = case (evenSlow m) of { T → T; F → F }

oddSlow2 m = case (evenSlow2 m) of { T → F; F → T }
```

Now we can extract two equal definitions for function pairs evenSlow, oddSlow, and evenSlow2, oddSlow2, the corresponding nodes are highlighted. Here is the definitions for the first pair:

```
evenSlow n = case n of { Z → T; S m → oddSlow m }
oddSlow n = case (evenSlow n) of { T → F; F → T }
```

The definitions for the second pair of functions is the same up to function names and names of bound variables. As it can be seen, all the recursive calls here are preformed on structurally smaller arguments, so there may be no more than one fixed point of each subgraph, and since the subgraphs are bisimilar, we come to a conclusion that evenSlow = evenSlow2 and oddSlow = oddSlow2. Let's add this information to the E-graph, thus performing merging by bisimilarity:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
odd n = case n of { Z → F; S m → even m }

evenSlow n = case n of { Z → T; S m → oddSlow m }
evenSlow n = case (oddSlow n) of { T → F; F → T }
evenSlow n = case (evenSlow n) of { T → T; F → F }

oddSlow n = not (evenSlow n)
oddSlow n = case (evenSlow n) of { T → F; F → T }
oddSlow n = case n of { Z → F; S m → evenSlow m }
```

Now we can perform another merging by bisimilarity to equate the functions even and evenSlow. The needed bisimilar subgraphs conists of the nodes highlighted in the above program. The resulting E-graph is the third (and last, since we've reached the goal) generation and looks like this:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
even n = case (odd n) of { T → F; F → T }
even n = case (even n) of { T → T; F → F }

odd n = case n of { Z → F; S m → even m }
odd n = not (even n)
odd n = case (even n) of { T → F; F → T }
```

It is interesting to point out that although we proved the goal statement using two mergings by bisimilarity (i.e. we used a lemma), we could have managed with only one if we had waited till the fourth generation without using induction. So sometimes lemmas aren't really required but may lead to earlier proof completion. Still, it doesn't mean that the proof will be found faster, even more, usually our tool takes slightly less time if we restrict application of merging by bisimilarity to the nodes from the goal statement since it doesn't have to check all equivalence class pairs from the E-graph in this case – but this is achieved at the cost of failures on tasks that really require lemmas.

7 Experimental evaluation

We've used a set of simple equations to evaluate our prover and compare it to similar tools. We've split this set into four groups: a main group of relatively simple equalities (Table 1) which don't seem to need any special features, a group of equalities that require nontrivial generalizations (Table 2), a group of equalities that need strong induction (Table 3), and a group of equalities that require coinduction (Table 4). The tests can be found in our repository [1] under the directory `samples`. For some of the tests we gave human-readable equations in the second column – note though that real equations are often a bit more complex because we have to make sure that they hold in a non-total untyped language.

The tables show average wall-clock time in seconds that the tools we've tested spent on the tests. We used a time limit of 5 minutes, runs exceeding the time limit counted as failures. We ran our benchmark on an Intel(R) Core(TM) i7 930 @ 2.80 GHz machine with Ubuntu 12.04. The tools we used in our benchmarking were:

- **graphsc**. Graphsc is our experimental prover based on the methods described in this paper. Note that although it internally works only with first-order functions, and there are many equalities in our sets involving higher-order functions, it can still prove them, because we perform defunctionalization before conversion to E-graph.
- **hosc**. HOSC is a supercompiler designed for program analysis, including the problem of function equivalence [12] (but it's not perfectly specialized for this task). It uses the following technique: first supercompile left hand side

and right hand side separately and then syntactically compare the residual programs [14,17]. The column labeled **hosc (hl)** corresponds to the higher-level version of HOSC [13, 15]. It can come up with lemmas necessary to prove the main goal and prove them using a single-level HOSC.

- **zeno**. Zeno [20] is an inductive prover for Haskell. Internally it is quite similar to supercompilers. Zeno assumes totality, so it is not fair to compare tools that don't (our tool and HOSC) to pure Zeno, so we used a trick to encode programs operating on data with bottoms as total programs by adding additional bottom constructor to each data type. The results of Zeno on the adjusted samples are shown in the column **zeno (p)**. The results of pure Zeno (assuming totality) are shown for reference in the column **zeno (t)**.
- **hipspec**. HipSpec [5] is an inductive prover for Haskell which can generate conjectures by testing (using QuickSpec [6]), prove them using an SMT-solver, and then use them as lemmas to prove the goal and other conjectures. Like Zeno, HipSpec assumes totality, so we use the same transformation to model partiality. The results are shown in columns **hipspec (p)** and **hipspec (t)**. Note also that the results of HipSpec are sensitive to the `Arbitrary` type class instances for data types. We generated these instances automatically and ran HipSpec with `--quick-check-size=10` to maximize the number of tests passed given these instances. We also used the maximal induction depth of 2 (`-d2`) to make HipSpec pass two tests requiring strong induction.

Since the test set is not representative, it is useless to compare the tools by the number of test they pass. Moreover, the tools seem to fall into different niches. Still, some conclusions may be drawn from the results.

First of all, HipSpec is a very powerful tool, in total mode it proves most of the equalities from the main set (Table 1) and all of the equalities that require complex generalizations (Table 2). However, it is very slow on some tests. It is also much less powerful on tests adjusted with bottoms. Indeed, partial equalities are often a bit trickier to prove than their total counterparts. It is also possible that this particular approach of reducing a partial problem to a total one and then using a total prover is not very efficient.

Zeno and HOSC are very fast which seems to be due to their depth-first nature. Zeno is also quite powerful and can successfully compete with the slower HipSpec, especially in the partial case. HOSC fails many test from the main set presumably due to the fact that it is not specialized enough for the task of proving equivalences. For example, the equivalence `idle-simple` is much easier to prove when transforming both sides simultaneously. Also HOSC can't prove `bool-eq` and `sadd-comm` because they need the transformation (case-of-transpose) which supercompilers usually lack. Interestingly, higher-level HOSC does prove some additional equalities, but not in the case of tests that really need lemmas (except `even-double-acc` from Table 2), which are the last four tests in the main set (they need lemmas in a sense that neither Graphsc, nor HipSpec can prove them without lemmas).

Name	Description	graphsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hipspec (p)	hipspec (t)
add-assoc	x + (y + z) = (x + y) + z	1.6	0.5	0.6	0.3	0.3	8.8	0.9
append-assoc	x ++ (y ++ z) = (x ++ y) ++ z	1.8	0.5	0.6	0.3	1.1	4.9	3.2
double-add	double (x+y) = double x + double y	2.2	0.6	0.6	0.3	0.3	21.5	2.0
even-double	even (double x) = true	1.7	0.6	0.6	0.3	0.3	82.5	97.9
ho/concat-concat	concat (concat xs) = concat (map concat xs)	3.2	0.6	0.7	0.4	0.4	80.0	47.0
ho/filter-append	filter p (xs ++ ys) = filter p xs ++ filter p ys	3.0	0.6	0.8	0.3	0.3	9.6	4.8
ho/map-append	map f (xs ++ ys) = map f xs ++ map f ys	2.1	0.6	0.7	0.3	0.3	9.8	4.5
ho/map-comp	map (f . g) xs = (map f . map g) xs	3.4	0.6	0.6	0.3	0.3	4.7	4.7
ho/map-concat	map f (concat x) = concat (map (map f) x)	2.8	0.6	0.7	0.3	0.3	91.4	47.9
ho/map-filter	filter p (map f xs) = map f (filter (p . f) xs)	3.6	0.7	0.7	0.3	0.3	6.3	5.8
idnat-idemp	idNat (idNat x) = idNat x	1.5	0.5	0.5	0.3	0.3	0.8	0.7
take-drop	drop n (take n x) = []	2.4	0.6	0.7	0.3	0.3	47.8	9.6
take-length	take (length x) x = x	2.3	0.6	0.6	0.3	0.3	6.8	7.9
length-concat	length (concat x) = sum (map length x)	2.8	0.7	0.8	0.3	0.3	fail	8.5
append-take-drop	take n x ++ drop n x = x	3.6	fail	1.1	0.5	0.3	113.0	11.9
deepseq-idemp	deepseq x (deepseq x y) = deepseq x y	1.8	fail	0.9	0.3	0.3	4.7	1.6
deepseq-s	deepseq x (S y) = deepseq x (S (deepseq x y))	2.1	fail	0.7	0.3	0.3	10.1	0.7
mul-assoc	(x * y) * z = x * (y * z)	11.6	0.8	fail	0.3	0.4	176.2	30.4
mul-distrib	(x*y) + (z*y) = (x + z)*y	3.9	0.7	fail	0.3	0.3	151.8	92.1
mul-double	x * double y = double (x*y)	5.1	0.6	fail	0.3	0.3	165.6	142.1
ho/fold-append	foldr f (foldr f a ys) xs = foldr f a (xs ++ ys)	2.1	0.6	0.7	fail	0.3	176.8	4.6
ho/church-id	unchurch (church x) = x	6.1	0.6	0.6	0.3	0.3	fail	fail
ho/church-pred		fail	0.7	0.8	fail	fail	fail	fail
ho/church-add		fail	0.7	0.7	0.3	0.3	fail	fail
idle-simple	idle x = idle (idle x)	1.4	fail	fail	0.3	0.3	0.8	0.7
bool-eq		1.3	fail	fail	0.3	0.3	1.1	0.8
sadd-comm		2.1	fail	fail	0.3	0.3	3.3	16.7
ho/filter-idemp	filter p (filter p xs) = filter p xs	fail	fail	fail	0.3	0.3	1.3	0.9
even-slow-fast	even x = evenSlow x	1.8	fail	fail	fail	fail	2.6	1.1
or-even-odd	even x || odd x = true	3.9	fail	fail	fail	0.3	128.9	1.0
dummy		1.6	fail	fail	0.3	0.3	2.4	0.8
idle	idle x = deepseq x 0	1.5	fail	fail	0.3	0.3	1.8	0.7
quad-idle		1.9	fail	fail	0.3	0.3	fail	0.7
exp-idle		3.4	fail	fail	0.3	fail	fail	1.7

Table 1: Comparison of different tools on the main test subset

Our tool, Graphsc, seems to be in the middle: it's slower than HOSC and Zeno (and it should be since it's breadth-first in nature) but rarely needs more than 10 seconds. It's interesting to analyze the failures of our tool. It fails three tests from the main set. The tests `ho/church-pred` and `ho/church-add` need deeper driving, our tool can pass them in the experimental supercompilation mode which change the order of transformation to resemble that of traditional supercompilers. Unfortunately, this mode is quite slow and leads to many other failures when enabled by default. The test `ho/filter-idemp` is interesting: it needs more information to be propagated, namely that the expression p x evaluates to True. Since this expression is not a variable, we don't propagate this information (and neither does HOSC, however there is an experimental mode for HOSC that does this and helps pass this test).

Name	Description	graphsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hipspec (p)	hipspec (t)
even-dbl-acc-lemma	even (doubleAcc x (S y)) = odd (doubleAcc x y)	fail	0.7	0.6	0.3	0.3	38.8	37.4
nrev-idemp-nat		fail	fail	fail	0.3	0.3	21.9	2.0
deepseq-add-comm		fail	fail	fail	fail	0.3	fail	2.1
even-double-acc	even (doubleAcc x 0) = true	fail	fail	0.8	fail	fail	fail	38.4
nrev-list	naiveReverse = reverse	fail	fail	fail	fail	fail	185.5	19.7
nrev-nat		fail	fail	fail	fail	fail	fail	1.1

Table 2: Comparison of the tools on the tests that require nontrivial generalization

Now let's look at the tests requiring nontrivial generalizations (Table 2). Here we call a generalization trivial if it's just peeling of the outer function call, e.g. f (g a) (h b c) trivially generalizes to f x y with x = g a and y = h b c. Our tool supports only trivial generalizations, and they are enough for a large number of examples. But in some cases more complex generalizations are needed, e.g. to prove the equality `even-dbl-acc-lemma` one need to generalize the expression odd (doubleAcc x (S (S y))) to odd (doubleAcc x z) with z = S (S y). It's not super sophisticated, but the expression left after taking out the S (S y) is a composition of two functions, which makes this generalization nontrivial. Our tool is useless on these examples. Supercompilers like HOSC usually use most specific generalizations which helps in some cases. But the best tool to prove equalities like these is HipSpec (and still it doesn't work that well in the partial case).

In Table 3 the tests are shown that require strong induction, i.e. induction schemes that peel more than one constructor at a time. This is not a problem for Graphsc and HOSC since they don't explicitly instantiate induction schemes. But Zeno and HipSpec do. In the case of HipSpec the maximum induction depth can be increased, so we specified the depth of 2, which helped HipSpec to pass two of these tests at the price of increased running times for other tests.

Name	Description	gra-phsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hip-spec (p)	hip-spec (t)
add-assoc-bot		2.1	0.6	0.6	fail	fail	fail	fail
double-half	double (half x) + mod2 x = x	4.6	fail	1.2	fail	fail	81.7	6.6
length-intersperse	length (intersperse x xs) = length (intersperse y xs)	fail	0.6	0.7	fail	fail	1.6	0.9
kmp-eq		fail	1.2	1.7	fail	fail	fail	fail

Table 3: Comparison of the tools on the tests that require strong induction

Our tool doesn't pass the KMP-test because it requires deep driving (and again, our experimental supercompilation mode helps pass it). In the case of `length-intersperse` it has trouble with recognizing the goal as something worth proving because both sides are equal up to renaming. Currently it is not obvious how this (seemingly technical) problem can be solved.

Name	Description	gra-phsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hip-spec (p)	hip-spec (t)
inf	fix S = fix S	1.2	0.4	0.5	fail	fail	fail	fail
shuffled-let		1.5	0.5	0.5	fail	fail	fail	fail
shifted-cycle	cycle [A,B] = A : cycle [B,A]	3.6	fail	fail	fail	fail	fail	fail
ho/map-iterate	map f (iterate f a) = iterate f (f a)	fail	0.6	0.6	fail	fail	fail	fail

Table 4: Comparison of the tools on the tests that require coinduction

The last test subset to discuss is the subset of tests requiring coinduction (Table 4). Coinduction is not currently supported by Zeno and HipSpec, although there are no obstacles to implement it in the future. The equality `ho/map-iterate` can't be proved by our tool because besides coinduction it needs a nontrivial generalization.

8 Related work

Our work is based on the method of equality saturation, originally proposed by Tate et al [23], which in turn is inspired by E-graph-based theorem provers like Simplify [7]. Their implementation, named Peggy, was designed to transform programs in low-level imperative languages (Java bytecode and LLVM), although internally Peggy uses a functional representation. In our work we transform lazy functional programs, so we don't have to deal with encoding imperative operations in functional representation, which makes everything much easier. Another difference is that in our representation nodes correspond to functions, not just first-order values, which allows more general recursion to be used, moreover we merge equivalence classes corresponding to functions equal up to parameter permutation, which considerably reduces the E-graph complexity. We also articulate the merging by bisimilarity transformation which plays a

very important role, making our tool essentially an inductive prover. Note that Peggy has a similar (but simpler) transformation that can merge θ-nodes, but it doesn't seem to be published anywhere.

Initially our work arose from analyzing differences between overgraph supercompilation [9] and equality saturation, overgraph supercompilation being a variety of multi-result supercompilation with a flavor of equality saturation. The present paper is loosely based on the preprint [10] which used a different terminology (hypergraph instead of E-graph, hence the name of our GitHub repository). We also used to consider the method to be a kind of supercompilation, but although it borrows a lot from supercompilation, it is much closer to equality saturation.

Supercompilation [24] is a program transformation technique that consists in building a process tree (perhaps implicitly) by applying driving and generalization to its leaves, and then folding the tree, essentially producing a finite program, equivalent to the original one. Although supercompilation is usually considered a source-to-source program transformation, it can be used to prove program equivalence by syntactically comparing the resulting programs, due to the normalizing effect of supercompilation.

Traditional supercompilers always return a single program, but for some tasks, like program analysis, it is beneficial to produce a set of programs for further processing. This leads to the idea of multi-result supercompilation, which was put forward by Klyuchnikov and Romanenko [16]. Since there are many points of decision making during the process of supercompilation (mainly when and how to generalize), a single-result supercompiler may be transformed into a multi-result one quite easily by taking multiple paths in each such point. The mentioned motivation behind multi-result supercompilation is essentially the same as that behind equality saturation.

Another important enhancement of traditional supercompilation is higher-level supercompilation. Higher-level supercompilation is a broad term denoting systems that use supercompilation as a primitive operation, in particular supercompilers that can invent lemmas, prove them with another (lower-level) supercompiler, and use them in the process of supercompilation. Examples of higher-level supercompilation are distillation, proposed by Hamilton [11], and two-level supercompilation, proposed by Klyuchnikov and Romanenko [13, 15].

Zeno [20] is an inductive prover for Haskell which works quite similarly to multi-result supercompilation. Indeed, Zeno performs case analysis and applies induction (both correspond to driving in supercompilation) until it heuristically decides to generalize or apply a lemma (in supercompilation this heuristic is called a whistle). That is, both methods are depth-first in nature unlike the equality saturation approach, which explores possible program transformations in breadth-first manner.

HipSpec [5] is another inductive prover for Haskell. It uses theory exploration to discover lemmas. For this purpose it invokes QuickSpec [6], which generates all terms up to some depth, splits them into equivalence classes by random testing, and then transforms these classes into a set of conjectures. After that

these conjectures are proved one by one and then used as lemmas to prove other conjectures and the main goal. To prove conjectures HipSpec uses external SMT-solvers. This bottom-up approach is contrasted to the top-down approach of most inductive provers, including Zeno and supercompilers, which invent lemmas when the main proof gets stuck. HipSpec discovers lemmas speculatively which is good for finding useful generalizations but may take much more time.

As to our tool, we do something similar to the bottom-up approach, but instead of using arbitrary terms, we use the terms represented by equivalence classes of the E-graph (i.e. generated by transforming initial term) and then try to prove them equal pairwise, discarding unfruitful pairs by comparing perfect tree prefixes that have been built in the E-graph so far, instead of testing. Since we use only terms from the E-graph, we can't discover complex generalizations this way, although we can still find useful auxiliary lemmas sometimes (but usually for quite artificial examples).

Both Zeno and HipSpec instantiate induction schemes while performing proof by induction. We use a different technique, consisting in checking the correctness of a proof graph, similarly to productivity and termination checking in languages like Agda. This approach has some advantages, for example we don't have to know the induction depth in advance. Supercompilers usually don't even check the correctness because for single-level supercompilation it is ensured automatically. It is not the case for higher-level supercompilation, and for example, HOSC checks that every lemma used is an improvement lemma in the terminology of Sand's theory [19].

9 Conclusion and future work

In this paper we have shown how an inductive prover for a non-total first-order lazy functional language can be constructed on top of the ideas of equality saturation. The key ingredient is merging by bisimilarity which enables proof by induction. Another feature that we consider extremely important is the ability to merge equivalence classes even if they represent functions equal only up to some renaming. This idea can be extended, for example if we had ticks, we could merge classes representing functions which differ by a finite number of ticks, but we haven't investigated into it yet.

Of course our prover has some deficiencies:

- Our prover lacks proper generalizations. This is a huge issue since many real-world examples require them. We have an experimental flag that enables arbitrary generalizations, but it usually leads to combinatorial explosion of the E-graph. There are two plausible ways to fix this issue. The first one is to use some heuristics to find generalizations from failed proof attempts, like it's done in supercompilers and many inductive provers. The other one is to rely on some external generalization and lemma discovery tools. In this case a mechanism of applying externally specified lemmas and generalizations may be very useful. In the case of E-graphs it is usually done with E-matching,

and we have an experimental implementation, although it doesn't work very well yet.
– Although it is possible to prove some propositions that hold only in total setting by adding some transformations, our prover is not very effective on this task. It may not seem to be a big problem if we only work with non-total languages like Haskell, but actually even in this case the ability to work with total values is important since such values may appear even in partial setting, e.g. when using the function `deepseq`.
– Our internal representation is untyped, and for this reason we cannot prove some natural equalities.
– We don't support higher-order functions internally and need to perform defunctionalization if the input program contains them. This issue is especially important if we want to produce a residual program.
– Our prover is limited to propositions about function equivalence, and it is not obvious how to add support for implications.

Besides mitigating the above problems, another possibility for future work is exploring other applications. Equality saturation is a program transformation technique which is not limited to proving function equivalence. Initially it was successfully applied to imperative program optimization, so some results in the functional field are to be expected. Even merging by bisimilarity may be of some use since it is known that using lemmas may lead to superlinear performance improvement. Another possible area is program analysis.

Acknowledgements

The author would like to express his gratitude to Sergei Romanenko, Andrei Klimov, Ilya Klyuchnikov, and other participants of the Refal seminar at Keldysh Institute.

References

1. Graphsc source code and the test suite. `https://github.com/sergei-grechanik/supercompilation-hypergraph`.
2. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 1990.
3. A. Abel. Foetus – termination checker for simple functional programs, July 16 1998.
4. A. Abel and T. Altenkrich. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002.
5. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In M. P. Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2013.

6. K. Claessen, N. Smallbone, and J. Hughes. Quickspec: Guessing formal specifications using testing. In G. Fraser and A. Gargantini, editors, *Tests and Proofs, 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010. Proceedings*, volume 6143 of *Lecture Notes in Computer Science*, pages 6–21. Springer, 2010.
7. Detlefs, Nelson, and Saxe. Simplify: A theorem prover for program checking. *JACM: Journal of the ACM*, 52, 2005.
8. A. Dovier and C. Piazza. The subgraph bisimulation problem. *IEEE Transactions on Knowledge & Data Engineering*, 15(4):1055–6, 2003. Publisher: IEEE, USA.
9. S. A. Grechanik. Overgraph representation for multi-result supercompilation. In A. Klimov and S. Romanenko, editors, *Proceedings of the Third International Valentin Turchin Workshop on Metacomputation*, pages 48–65, Pereslavl-Zalessky, Russia, July 2012. Pereslavl-Zalessky: Ailamazyan University of Pereslavl.
10. S. A. Grechanik. Supercompilation by hypergraph transformation. Preprint 26, Keldysh Institute of Applied Mathematics, 2013. URL: `http://library.keldysh.ru/preprint.asp?id=2013-26&lg=e`.
11. G. W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70. ACM Press New York, NY, USA, 2007.
12. I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.
13. I. Klyuchnikov. Towards effective two-level supercompilation. Preprint 81, Keldysh Institute of Applied Mathematics, 2010. URL: `http://library.keldysh.ru/preprint.asp?id=2010-81&lg=e`.
14. I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.
15. I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.
16. I. G. Klyuchnikov and S. A. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In E. Clarke, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011*, volume 7162 of *Lecture Notes in Computer Science*, pages 210–226. Springer, 2012.
17. A. Lisitsa and M. Webster. Supercompilation for equivalence testing in metamorphic computer viruses detection. In *Proceedings of the First International Workshop on Metacomputation in Russia*, 2008.
18. Nelson and Oppen. Fast decision procedures based on congruence closure. *JACM: Journal of the ACM*, 27, 1980.
19. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.
20. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, Lecture Notes in Computer Science, March 2012.
21. M. Sørensen, R. Glück, and N. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1993.
22. M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 737–742. Springer, 2011.

23. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.
24. V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.