ISSN 0361-7688, Programming and Computer Software, 2015, Vol. 41, No. 3, pp. 149–161. © Pleiades Publishing, Ltd., 2015.
Original Russian Text © S.A. Grechanik, 2015, published in Programmirovanie, 2015, Vol. 41, No. 3.

Proving Properties of Functional Programs by Equality Saturation[1]

S. A. Grechanik

Keldysh Institute of Applied Mathematics, 4 Miusskaya sq., Moscow, 125047 Russia

e-mail: sergei.grechanik@gmail.com

Received December 1, 2014

Abstract—The present paper shows how the idea of equality saturation can be used to prove algebraic properties of programs written in a non-total non-strict first-order functional language. We adapt equality saturation approach to a functional language by using transformations borrowed mainly from supercompilation. Proof by induction is performed via a special transformation called merging by bisimilarity. We compare our experimental prover based on this method with a supercompiler HOSC and inductive provers HipSpec and Zeno.

DOI: 10.1134/S0361768815030056

1. INTRODUCTION

Equality saturation [1] is a method of program transformation that uses a compact representation of multiple equivalent programs based on E-graphs. E-graphs are graphs whose nodes are joined into equivalence classes [2, 3]. They allow us to represent a set of equivalent programs, consuming exponentially less memory than representing it as a plain set by sharing their common parts. Equality saturation consists in enlarging this set of programs by applying certain transformations to the E-graph until there's no transformation to apply or the limit of transformation applications is reached. The transformations are usually 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 [4])—in this case program extraction is not needed.

In the original papers by Tate et al. [1] 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 of functions written in a non-strict functional language. We do this mainly by borrowing transformations from supercompilation [5, 6]. 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 [7] and Zeno [8] we don't instantiate the induction scheme, but instead check the correctness of the proof graph similarly to Agda and Foetus [9, 10]. 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 paper is organized as follows. In Section 2 we briefly describe equality saturation and how functional programs and their sets can be represented by E-graphs. Then in Section 3 we discuss basic transformations that 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 present experimental evaluation of our prover. Section 7 discusses related work and Section 8 concludes the paper.

The source code of our experimental prover can be found on GitHub [11].

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 [3], then the set of equalities will

[1] The article was translated by the authors.

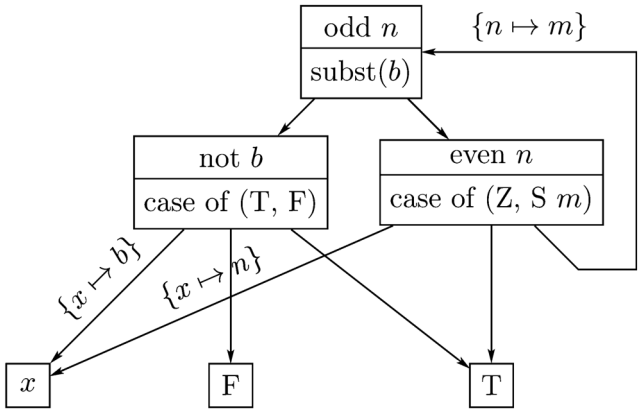

Fig. 1. Graph representation of a program.

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 or transformations describing some properties of 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 transformation application opportunities). This is what equality saturation basically is. So, to solve a problem of function/program equivalence using equality saturation one should first convert both function definitions to E-graphs and put both of them into a single E-graph and then transform the E-graph using some transformations until the target terms are in the same equivalence class or some limit of transformation application is reached—this process is called saturation. In pure equality saturation approach transformations are applied non-destructively and result only in adding new nodes and edges, and merging of equivalence classes. But in practice this is not very efficient, and in our prover we apply some transformations destructively, removing some nodes and edges.

In this paper we will use a 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 following program:

not b = **case** b **of** {T ⟶ F; F ⟶ T}
even n = **case** n **of** {Z ⟶ T; S m ⟶ odd m}
odd n = not (even n)

This program can be naturally represented as a graph, as shown in Fig. 1. 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.

All edges of an E-graph 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 put nodes representing the same function modulo renaming into the same equivalence class, which would increase space consumption. Merging up to renaming will be discussed in Section 2.2.

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 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 a transformation which destructively replaces such explicit substitutions with renamings). Note that we require an explicit substitution to bind *all* variables of the expression being substituted—this simplifies the formulation of some transformations.

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 (Fig. 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 (as "nondeterministic" programs).

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 equivalent 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}

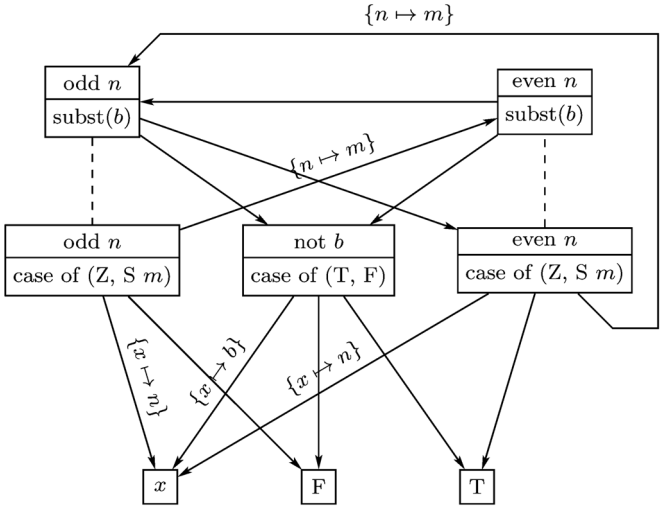

Fig. 2. E-graph representing functions even and odd.

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 describe kinds of node labels and how they correspond to language constructs.

First of all, each node of an E-graph is a member of some equivalence class (which in the simplest case contains only this node). We will use symbols f, g, h, ... 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 from f to g of an E-graph is labeled with a bijective renaming $V(g) \longleftrightarrow V(f)$ (if in the process of transformation it stops being bijective then one of the sets $V(f)$ or $V(g)$ should be automatically decreased).

We will use the notation $f = L \longrightarrow \theta_1 g_1, ..., \theta_n g_n$ to denote a node f with a label L and edges with labels θ_i from f to 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:

• $f = x$. (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

• $f = \text{subst}(x_1, ..., x_n) \longrightarrow \xi h, \theta_1 g_1, ..., \theta_n g_n$. (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 x y} **in** h u v
= (λ u v . h u v) (g1 x) (g2 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 {u = g1 x, v = g2 x y}

• $f = C \longrightarrow \theta_1 g_1, ..., \theta_n g_n$. (Constructor). Code example:

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

• $f = \text{caseof}(C_1 \overline{x_1}, ..., C_n \overline{x_n}) \longrightarrow \xi h, \theta_1 g_1, ..., \theta_n g_n$. (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 $f = L \longrightarrow \theta_1 g_1, ..., \theta_n g_n$ 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 (unless $f = x$ in which case it doesn't have outgoing edges). We will use the following notation for this operation:

$$f' = \xi(L \longrightarrow \theta_1 g_1, ..., \theta_n g_n).$$

Formally the operation is defined as follows:

$$\xi(C \longrightarrow \theta_1 g_1, ..., \theta_n g_n) = C \longrightarrow (\xi \circ \theta_1)g_1, ..., (\xi \circ \theta_n)g_n,$$

$$\xi(\text{subst}(...) \longrightarrow \zeta h, \theta_1 g_1, ..., \theta_n g_n) = \text{subst}(...) \longrightarrow \zeta h, (\xi \circ \theta_1)g_1, ..., (\xi \circ \theta_n)g_n,$$

$$\xi(\text{caseof}(...) \longrightarrow \zeta h, \theta_1 g_1, ..., \theta_n g_n) = \text{caseof}(...) \longrightarrow (\xi \circ \zeta)h, (\xi'_1 \xi \circ \theta_1)g_1, ..., \xi'_n \circ \theta_n)g_n.$$

In the last case each ξ'_i maps the variables bound by 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. Initially new nodes are created in their own separate equivalence classes, and then these classes may be merged as a result of applying some transformations, which corresponds to adding new equalities. 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 $g = x$, and if they both were equal to a variable node then our E-graph would be self-contradictory which would indicate a bug in the transformation system). 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 doesn't work if f and g are in the same equivalence class. In this case the equation actually looks like $f = \theta f$ and should be modelled with an explicit substitution. In practice this case is very rare and corresponds to function commutativity.

3. TRANSFORMATIONS

3.1. Congruence

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

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

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 transformation 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 other transformations.

Also note that to make the search for the appropriate faster, it is beneficial to represent nodes in normal form:

$$f = \zeta(L \longrightarrow \theta_1 g_1, ..., \theta_n g_n),$$

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

3.2. Injectivity

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

$$\frac{f = L \longrightarrow \theta_1 h_1, ..., \theta_n h_n \quad g = L \longrightarrow \zeta_1 k_1, ..., \zeta_n k_n \quad f \cong g \quad L \text{ is injective}}{\forall i \,.\, h_i = \theta_i^{-1} \zeta_i k_i}$$

"L is injective" 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 $\theta_2 h_2, ..., \theta_n h_n, \zeta_2 k_2, ..., \zeta_n k_n$ uses this variable (in other words, positive information is propagated). This transformation is also advantageous to apply as early as possible.

3.3. Semantics of Explicit Substitutions

In this and the next sections we will write transformations 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).

$$
\begin{aligned}
&\text{(subst-id)} && x\ \{x = g\} \ \mapsto\ g \\
&\text{(subst-subst)} && f\ x\ \{x = g\ y\}\ \{y = h\} \ \mapsto\ f\ x\ \{x = g\ y\ \{y = h\}\} \\
&\text{(subst-constr)} && C\ (f\ x)\ (g\ x)\ \{x = h\} \ \mapsto\ C\ (f\ x\ \{x = h\})\ (g\ x\ \{x = h\}) \\
&\text{(subst-case-of)} && (\textbf{case}\ f\ x\ \textbf{of}\ C\ y \to g\ x\ y)\ \{x = h\} \ \mapsto \\
& && \quad \textbf{case}\ f\ x\ \{x = h\}\ \textbf{of}\ C\ y \to g\ x\ y\ \{x = h,\ y = y\}
\end{aligned}
$$

Fig. 3. Transformations of explicit substitutions.

(case-of-constr)
(**case** Ce **of** $C\,y \to fxy$) $\mapsto$
$fxy\{x = x, y = e\}$
(case-of-case-of)
(**case** (**case** e **of** $C_1 y \to g$) **of** $C_2 z \to h$) $\mapsto$
case e **of** $C_1 y \to$ (**case** g **of** $C_2 z \to h$)
(case-of-id)
(**case** e **of** $Cyz \to fxyz$) $\mapsto$
case x **of** $Cyz \to$
$fxyz\{x = (Cyz), y = y, z = z\}$
(case-of-transpose)
case h **of** {
$C_1 x \to$ **case** z **of** $Dv \to fvx$;
$C_2 y \to$ **case** z **of** $Dv \to gvy$;
} $\mapsto$
case z **of** $Dv \to$
case h **of** {
$C_1 x \to fvx$;
$C_2 y \to gvy$;
}

Fig. 4. Transformations of pattern matching.

We use the compact postfix notation to express explicit substitutions. We use letters e, f, g, h, ... 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 Fig. 3 four transformations of explicit substitutions are shown. All of them basically describe how to evaluate a node if it is an explicit substitution. 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)).

Usually in program transformation systems 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 transformations concerning substitution. The first one is elimination of unused variable bindings, (subst-unused):

$$f\,x\,y\,\{x = g, y = h, z = k\} \mapsto f\,x\,y\,\{x = g, y = h\}$$

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

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

$$f\,x\,y\,\{x = y, y = z\} \mapsto f\,y\,z.$$

Note though that application of this transformation results in merging of the equivalence classes cor-

responding to the node representing the substitution and the node *f*, so if they are already in the same class, this transformation is inapplicable. We also apply this transformation destructively.

3.4. Semantics of Pattern Matching

The transformations concerning pattern matching are shown in Fig. 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 transformations came from supercompilation [5, 6]. They tell us what to do when we get stuck during computation because of missing information (i.e. a variable in place of a scrutinized expression). The transformation (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 transformation (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 transformation, (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. Destructive Transformations

We apply some transformations destructively, i.e. remove the original nodes and edges that triggered the transformation. It is essentially a heuristic, which is a deviation from pure equality saturation approach, but without it proving even simple equalities takes too long. Note that in theory if we don't care about performance, deleting information from the E-graph leads to decrease in proving power.

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 congruence transformation can merge two functions into one equivalence class if they have the same tree representation. But if their definitions involve cycles, then the congruence transformation 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 transformations 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 [1] there is such a transformation that merges θ-nodes.

The general idea of this kind of transformation, which we will call merging by bisimilarity, is to find two bisimilar subgraphs growing from two 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:

$$f\ x = C; \quad g\ x = D$$
$$f\ x = f\ (f\ x); \quad g\ x = g\ (g\ x)$$

The functions *f* and *g* are different but they both are idempotent, which is stated by the additional definitions, which can be seen as 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 functions *f* and *g* 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 [10]. These conditions are usually used in total languages to ensure termination and productivity, but they can also be used to guarantee *uniqueness* of the function satisfying a definition in a non-total language with infinite and partial values (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.

```
function BISIMILAR?(c1, c2, history)
    if c1 = c2 then return true
    else if (c1, c2) ∈ history then
        if uniqueness conditions hold then return true
        else return false
    else if c1 and c2 contain incompatible nodes then return false
    else
        for m ∈ c1, n ∈ c2 : 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?(class(m'), class(n'), {(c1, c2)} ∪ history) then
                return true
        return false
```

Fig. 5. Algorithm for searching for two bisimilar subgraphs.

Note that we have refused to interpret functions as least fixed points of their definitions because in the E-graph there often appear definitions that are not equivalent to the original one in this interpretation (like in the indempotence example above). Thus there is a subtle difference between subgraphs that may have multiple fixed points and subgraphs that have a single fixed point equal to $\bot$. Consider the following function "definition":

$$f\,x = f\,x$$

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

f x = eat infinity
infinity = S infinit
eat x = **case** x **of** { S y ⟶ eat y}

The definition of **infinity** is guardedly recursive, and the definition of **eat** is structurally recursive, so both of them have unique fixed points, and consequently the function **f** also have a unique fixed point as their composition. This fixed point is equal to $\bot$.

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 [12] as in two-level supercompilation [13].

4.1. Algorithm Description

In this subsection we'll describe the algorithm that we use to figure out if two equivalence classes have two bisimilar subgraphs growing from them and meeting the uniqueness condition. First of all, the problem of finding two bisimilar subgraphs is a variation of the subgraph bisimulation problem which is NP-complete [14]. In certain places we trade completeness for performance, so sometimes our algorithm fails to find the subgraphs when they exist. Nevertheless, merging by bisimilarity is still one of the biggest performance issues in our experimental implementation.

The algorithm we use is outlined in Fig. 5. It takes two equivalence classes and a history of visited classes (initially empty) and returns true if they are bisimilar (i.e. there are two bisimilar subgraphs growing from them) or false if it cannot find a bisimulation. The algorithm consists in simultaneous depth-first traversal of the E-graph from the given classes. If two classes are the same then we consider them to be bisimilar by definition and return true. If we encounter a previously visited pair of classes, 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, because if we check the uniqueness condition on the second encounter it may hold even if it doesn't on the first one, but for efficiency we stop searching). Note that some kinds of uniqueness conditions have to be checked after the whole bisimilation 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 none of the previous cases is applicable, we try to disprove equivalence of the given classes (the algorithm will work without this but very inefficiently). To do this we check if there are incompatible nodes in the classes, like different constructors or a constructor and a pattern matching on a variable or two pattern

matchings on different variables (this check may be performed to a certain depth). If there are incompatible nodes then their equivalence classes cannot be bisimilar, and we return false. Otherwise we go on and check all pairs of nodes from the given classes to find a pair of nodes such that all their child nodes' classes are bisimilar. If there is such a pair then the original equivalence classes are bisimilar.

Note that the described algorithm can be easily modified to return a lazy list of bisimulations, which is necessary to do in reality because we need them for checking the uniqueness condition. We have also ignored the question of renamings here. Since in practice we want to merge classes up to renamings, we should also search for bisimulations up to renamings, which complicates the real implementation a little bit.

5. ON ORDER OF TRANSFORMATION

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 transformations are complex enough (or at least it will take too long). In particular, the transformations we described above can be applied indefinitely in most cases. To solve this problem we can add monotone preconditions to transformations, similar to whistles in supercompilers, which forbid applying transformations indefinitely (e.g. we can limit the depth of the nodes to which transformations may be applied).

Second, as we have said in Section 3.5, some transformations should be applied destructively, otherwise there will be too many useless nodes and edges in the E-graph. But when we apply transformations destructively, we lose the independence of the result from transformation order (or at least it becomes harder to prove).

Moreover, sometimes it is useful to stop somewhere in the middle, the state in this stop point being deterministic instead of changing from run to run. Such points can be organized using the mentioned preconditions: when we reach fully saturated state, we relax them and run the saturation process again (in this case whistles essentially become heuristic functions). The main problem is to gain sufficiently small time intervals between these points of intermediate saturated states.

Currently we use the following order of transformation, which is mainly based on the breadth-first approach and on partitioning transformations into subsets (inside a subset transformations can be rearranged but the order of applying different subsets is strictly fixed):

(1) Transform the programs and the goal 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 2–4 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. Note that inductive provers and supercompilers usually employ the depth-first approach, which can be used in equality saturation too, which may have some advantages. It is also may be advantageous to use some heuristic functions, but this question is not well-researched yet.

6. EXPERIMENTAL EVALUATION

To evaluate our prover and compare it to similar tools we've used our own set of simple equalities. The main reason of using our own suite was the fact that we work with a non-total language with partial and infinite data. Our suite is not representative, so we don't compare the tools by the number of tests they pass. Moreover, the tools seem to fall into different niches.

We've split this set into two groups: a main group of relatively simple equalities (Table 1) and a group of

Table 1. Comparison of the tools on the main test set

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
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/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
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 2. Comparison of the tools on the additional set of tests

Name	Description	graphsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hipspec (p)	hipspec (t)
Tests requiring nontrivial generalizations								
even-dbl-acclemma	`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
Tests requiring strong induction								
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
Tests requiring coinduction								
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

equalities that require some special features (Table 2), namely complex generalizations, strong induction, and coinduction. The tests can be found in our repository [11] 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 had to make sure that they held 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. The tools we used in our benchmarking were:

• **graphsc.** Graphsc (Graphical SuperCompiler, initially our tool was intended to be a supercompiler) 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. Our prover is written in Scala and can be found on GitHub [11].

• **hosc.** HOSC is a supercompiler designed for program analysis, including the problem of function equivalence [15] (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 [16, 17]. The column labeled **hosc (hl)** corresponds to the higher-level version of HOSC [13, 18]. It can come up with lemmas necessary to prove the main goal and prove them using a single-level HOSC.

• **zeno.** Zeno [8] 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 [7] is an inductive prover for Haskell, which can generate conjectures by testing (using QuickSpec [19]), 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.

Although the test set is not representative, and it is useless to compare the tools by the number of test they pass, 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. However, it is very slow on some tests. It is also much less powerful on tests adjusted with bottoms.

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 tests 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, 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).

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 lead 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).

Now let's look at the additional set of tests (Table 2). We'll start with tests requiring nontrivial generalizations. 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 `odd (doubleAcc x z)` 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, which takes generalizations from discovered lemmas.

Now let's consider tests 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 (they check correctness of proof graphs instead). But Zeno and HipSpec instantiate induction schemes, so these tests are problematic for them. 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.

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 and are represented by the same equivalence class. Currently it is not obvious how this (seemingly technical) problem can be solved.

The last test subset to discuss is the subset of tests requiring coinduction. 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.

7. RELATED WORK

Our work is based on the method of equality saturation, originally proposed by Tate et al. [1], which in turn is inspired by E-graph-based theorem provers like Simplify [2]. 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 have a similar (but simpler) transformation that can merge θ-nodes.

Initially our work arose from analyzing differences between overgraph supercompilation [20] and equality saturation, overgraph supercompilation being a variety of multi-result supercompilation with a flavor of

equality saturation. Its main difference from the present work is that equivalence classes in overgraphs (which are called nodes in overgraph terminology) correspond to expressions with free variables rather than to functions in abstract sense. In overgraph two equivalence classes may be merged only when their corresponding expressions are syntactically equal up to variable renaming, but not when they are only semantically equivalent. We also used to consider the method described in the present paper to be a kind of supercompilation, but although it borrows a lot from supercompilation, it is much closer to equality saturation.

Supercompilation [5] 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 [21]. 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 [22], and two-level supercompilation, proposed by Klyuchnikov and Romanenko [13, 18].

Zeno [8] 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 [7] is another inductive prover for Haskell. It uses theory exploration to discover lemmas. For this purpose it invokes QuickSpec [19], 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 [12].

8. CONCLUSIONS

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 might 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.

Besides mitigating the above problems, another possibility for future work is exploring other applications such as program optimization.

ACKNOWLEDGMENTS

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.

REFERENCES

1. Tate, R., Stepp, M., Tatlock, Z., and Lerner, S., Equality saturation: a new approach to optimization, *SIGPLAN Not.*, 2009, vol. 44, pp. 264–276.
2. Detlefs, D., Nelson, D., and Saxe, J.,, Simplify: a theorem prover for program checking, *JACM*, 2005, vol. 52, no. 3, pp. 365–473.
3. Nelson, G. and Oppen D.C.,, Fast decision procedures based on congruence closure, *JACM*, 1980, vol. 27, no. 2, pp. 356–364.
4. Stepp, M., Tate, R., and Lerner, S., Equality-based translation validator for LLVM, Gopalakrishnan, Ganesh and Qadeer, Shaz, Eds., *Computer Aided Verification—23rd International Conference, CAV 2011* (USA, UT, Snowbird, 2011), Springer, 2011, vol. 6806, pp. 737–742.
5. Turchin, V.F., The concept of a supercompiler, *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 1986, vol. 8, no. 3, pp. 292–325.
6. Sørensen, M.H., Glück, R., and Jones, N.D., A positive supercompiler, *J. Funct. Program.*, 1993, vol. 6, no. 6, pp. 811–838.
7. Claessen, K., Johansson, M., Rosén, D., and Smallbone, N., Automating inductive proofs using theory exploration, Maria Paola Bonacina, Ed., *Automated Deduction—CADE-24—24th International Conference on Automated Deduction* (USA, NY, Lake Placid, 2013), 2013, vol. 7898, pp. 392–406.
8. Sonnex, W., Drossopoulou, S., and Eisenbach, S., Zeno: an automated prover for properties of recursive data structures, in *TACAS, Lecture Notes in Computer Science*, 2012.
9. Abel, A., *Termination Checker for Simple Functional Programs,* 1998.
10. Abel, A. and Altenkrich, T., A predicative analysis of structural recursion, *J. Funct. Program.*, 2002, vol. 12, pp. 1–41.
11. Graphsc source code and the test suite, https://github.com/sergei-grechanik/supercompilation-hypergraph.
12. Sands, D., Total correctness by local improvement in the transformation of functional programs, *ACM Trans. Program. Lang. Syst.*, 1996, vol. 18, no. 2, pp. 175–234.
13. Klyuchnikov, I. and Romanenko, S., Towards higher-level supercompilation, *Second International Workshop on Metacomputation in Russia,* 2010.
14. Dovier, A. and Piazza, C., The subgraph bisimulation problem, *IEEE Transactions on Knowledge and Data Engineering*, USA: IEEE, 2003, vol. 15, no. 4, pp. 1055–1056.
15. Klyuchnikov, I., Supercompiler HOSC 1.0: under the hood, *Preprint of Keldysh Institute of Applied Mathematics*, Moscow, 2009, no. 63.
16. Lisitsa, A.P. and Webster, M., Supercompilation for equivalence testing in metamorphic computer viruses detection, *Proceedings of the First International Workshop on Metacomputation* (Russia, 2008).
17. Klyuchnikov, I. and Romanenko, S., Proving the equivalence of higher-order terms by means of supercompilation, in *Perspectives of Systems Informatics,* 2010, vol. 5947, pp. 193–205.
18. Klyuchnikov, I., Towards effective two-level supercompilation, *Preprint of Keldysh Institute of Applied Mathematics,* Moscow, 2010, no. 81; URL: http://library.keldysh.ru/preprint.asp?id=2010-81&lg=e.
19. Claessen, K., Smallbone, N., and Hughes, J., Quickspec: Guessing formal specifications using testing, Fraser, G. and Gargantini, A., Eds., *Tests and Proofs, 4th International Conference, TAP 2010* (Spain, Mälaga, 2010), Springer, 2010, vol. 6143, pp. 6–21.
20. Grechanik, S.A., Overgraph representation for multi-result supercompilation, Klimov, A.V. and Romanenko, S.A., Eds., *Proceedings of the Third International Valentin Turchin Workshop on Metacomputation* (Russia, Pereslavl-Zalessky, 2012), Pereslavl-Zalessky: Ailamazyan University of Pereslavl, pp. 48–65.
21. Klyuchnikov, I.G. and Romanenko, S.A., Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions, Clarke, Virbitskaite, I. and Voronkov, A., Eds., *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011* (Novosibirsk, Akademgorodok, Russia, 2011), Springer, 2012, vol. 7162, pp. 210–226.
22. Hamilton, G.W., Distillation: extracting the essence of programs, *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, N.Y.: ACM Press, 2007, pp. 61–70.