Towards Unification of Supercompilation and Equality Saturation

(Extended Abstract)

Sergei Grechanik*

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

Both equality saturation and supercompilation are methods of program transformation. The idea of equality saturation [2] is to infer new equalities about program functions from the initial ones (function definitions). These new equalities can then be used to various ends: a new program may be extracted from these equalities by choosing a representative set of equalities which will constitute the new program's definitions, or properties of the original program may be proved by looking at the set of inferred equalities. Previously we have shown that equality saturation is applicable to functional languages by using transformations (inference rules) borrowed from supercompilation (driving, more precisely), more specifically we used it for the problem of proving equalities [1].

The idea of supercompilation [3] is to build a process tree representing all possible paths of program execution and then transform it into a finite graph which can be easily turned into a new program. Building a process tree is done by using a combination of driving, generalization and folding.

A question might be asked: what is the relationship between equality saturation and supercompilation? Can't they be represented as special cases of something more general? Turns out that they can be (more or less): supercompilation can be seen as the very same equality inference process that underlies equality saturation, the only difference being that in supercompilation this inference is strictly guided by heuristics, whereas in equality saturation transformations are applied simply in breadth-first order. Indeed, a process tree can be represented as a set of equalities between configurations, and process tree building operations just infer new equalities.

Let's consider a simple example of supercompilation in equality inference style. We will use only equalities of the form $f(x_1, ..., x_n) = E$ (i.e. function definitions). Equalities of more general form $E_1 = E_2$ can be represented as multiple equalities of the aforementioned form by introducing auxiliary functions. This representation is actually asymptotically more efficient when we have many equalities.

* Supported by Russian Foundation for Basic Research grant No. 16-01-00813-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

Consider the following function definitions:

$$(1)\ add(x, y) = \textbf{case}\ x\ \textbf{of}\ \{Z \to y; S(x') \to sadd(x', y)\}$$
$$(2)\ sadd(x, y) = S(add(x, y))$$
$$(3)\ f(x, y, z) = add(add(x, y), z)$$

This is a classic addition associativity example. We will supercompile the function f to get a more optimal definition of the three-number sum function. f may be considered as the root node of our process tree with the right hand side of (3) being its configuration. Rules of supercompilation prescribe performing driving first, which in this case is just unfolding of the *add* function using its definition (1):

$$(4)\ f(x, y, z) = \textbf{case}\ add(x, y)\ \textbf{of}\ \{Z \to z; S(x') \to sadd(x', z)\}$$

To continue driving we should unfold *add* in the scrutinee position using (1) and lift the inner case-of up which is done in one inference step:

$$(5)\ f(x, y, z) = \textbf{case}\ x\ \textbf{of}\ \{Z \to f_1(y, z); S(x') \to f_2(x', y, z)\}$$
$$(6)\ f_1(y, z) = \textbf{case}\ y\ \textbf{of}\ \{Z \to z; S(y') \to sadd(y', z)\}$$
$$(7)\ f_2(x', y, z) = \textbf{case}\ sadd(x', y)\ \textbf{of}\ \{Z \to z; S(x'') \to sadd(x'', z)\}$$

(Auxiliary functions f_1 and f_2 were introduced to split up the complex right hand side of (5)). Now (5) has the form of variable analysis, which means that we are done with f and can move on to the branches f_1 and f_2. f_1 is not interesting and driving it won't actually add new equalities, so let's consider only f_2. In (7) we should unfold *sadd* using (2) and reduce the case-of using the appropriate branch (again, one inference step):

$$(8)\ f_2(x', y, z) = sadd(add(x', y), z)$$

Unfolding of *sadd* leads to

$$(9)\ f_2(x', y, z) = S(f_3(x', y, z))$$
$$(10)\ f_3(x', y, z) = add(add(x', y), z)$$

But the right hand side of (10) is the same as the right hand side of (3) (which is a configuration seen earlier), so we should perform folding. In equality saturation setting this is done by removing (10) and replacing f_3 with f in every definition. Here (9) is the only one we need to modify:

$$(9')\ f_2(x', y, z) = S(f(x', y, z))$$

To get a residual program we should traverse the definitions from f choosing one definition for each function. Actually supercompilation requires us to choose

the last ones (i.e. for f we take (5), not (4) or (3)):

$$
\begin{aligned}
&(5)\ f(x,y,z) = \textbf{case}\ x\ \textbf{of}\ \{Z \to f_1(y,z); S(x') \to f_2(x',y,z)\}\\
&(9')\ f_2(x',y,z) = S(f(x',y,z))\\
&(6)\ f_1(y,z) = \textbf{case}\ y\ \textbf{of}\ \{Z \to z; S(y') \to sadd(y',z)\}\\
&(2)\ sadd(x,y) = S(add(x,y))\\
&(1)\ add(x,y) = \textbf{case}\ x\ \textbf{of}\ \{Z \to y; S(x') \to sadd(x',y)\}
\end{aligned}
$$

This example shows that performing supercompilation as process of equality inference is possible in principle.

Equality saturation is guaranteed to eventually perform the same steps as supercompilation since it works in breadth-first manner. This actually makes equality saturation more powerful in theory. Indeed, it already subsumes multi-result supercompilation since it doesn't restrict application of different transformations to either driving or generalization. If we add merging by bisimulation, which is essentially checking equivalence of two expression by residualizing them, then we also get higher-level supercompilation. However, this power comes at a price: without heuristic guidance we risk to be hit by combinatorial explosion. And this actually happens in reality: our experimental prover was unable to pass the KMP-test because of this, and what is worse, generalizing rules had to be disabled which left our prover only with the simplest form of generalization, namely removal of the outermost function call. That's why it would be interesting to make a hybrid between supercompilation and pure equality saturation that would restrict transformation application, but not too much.

Currently our equality saturating prover has an experimental mode that performs driving up to certain depth before performing ordinary equality saturation. It allows our prover to pass the KMP-test as well as a couple of similar examples, but results in regression on some other examples. Although this mode shows that such a combination is possible, it is far from a fully fledged supercompilation/equality saturation hybrid since it lacks most of supercompilation heuristics and works in a sequential way (first driving, and only then breath-first saturation). Therefore, two directions of future research may be named here:

- Developing heuristics to control generalization. Using generalizing transformations without restriction leads to combinatorial explosion. Direct application of mgu from supercompilation doesn't seem to be a good solution because there may be too many pairs of terms due to multiresultness.
- Developing heuristics to control overall rewriting, mainly depth of driving. In supercompilation whistles are used for this purpose, but traditional homeomorphic embedding whistles are also hard to implement in multi-result setting of equality saturation.

Since the main difficulty in using traditional heuristics seems to be with multi-result nature of equality saturation, it is entirely possible that completely new methods should be developed.

References

1. S. Grechanik. Inductive prover based on equality saturation. In A. Klimov and S. Romanenko, editors, *Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation*, Pereslavl-Zalessky, Russia, July 2014. Pereslavl Zalessky: Publishing House "University of Pereslavl".
2. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.
3. V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.