Keldysh Institute • Publication search

Keldysh Institute preprints • Preprint No. 26, 2013

Grechanik S.A.

Supercompilation by hypergraph transformation

Recommended form of bibliographic references: Grechanik S.A. Supercompilation by hypergraph transformation. Keldysh Institute preprints, 2013, No. 26, 24 p. URL: http://library.keldysh.ru/preprint.asp?id=2013-26&lg=e

KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Sergei Grechanik

Supercompilation by Hypergraph Transformation

Moscow
2013

Sergei Grechanik. Supercompilation by Hypergraph Transformation

This paper presents a reformulation of the notion of multi-result supercompilation in terms of graph transformations. For this purpose we use a hypergraph-based representation of the program being transformed. The presented approach bridges the gap between supercompilation and equality saturation. We also show how higher-level supercompilation naturally arises in this setting.

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.

Сергей Гречаник. Суперкомпиляция путём преобразования гиперграфа

В данной работе представлена новая формулировка многорезультатной суперкомпиляции на основе преобразований графа. Для этого используется представление преобразуемой программы, основанное на гиперграфах. Данный подход соединяет суперкомпиляцию и насыщение равенствами. Также в работе показано, что в этих условиях естественным образом возникает многоуровневая суперкомпиляция.

Работа выполнена при поддержке гранта РФФИ № 12-01-00972-а и гранта Президента РФ для ведущих научных школ № НШ-4307.2012.9.

Contents

1 Introduction

Supercompilation is a method of equivalence-preserving source-to-source program transformation invented by Valentin Turchin [21]. Supercompilation is often seen as an optimization technique but it can also be used for program analysis. Here are some program analysis applications of supercompilation:

- Proving some property of a program by supercompiling composition of the property testing function and the program, and checking if the result of the supercompilation cannot return false [15].
- Proving properties by applying program analysis to the supercompiled program rather than the original one. The idea is that a supercompiled program may be easier to analyze [17].
- Proving equivalence of two programs by supercompiling them and then syntactically comparing the resulting programs [11, 16].

Although program analysis was declared as possible application of supercompilation from the very beginning, interest in this application grew quite recently. This interest motivated rethinking of supercompilation, as it turned out that traditional supercompilers adjusted to program optimization were not necessarily well-suited for program analysis. One of the ideas that appeared during this rethinking was the idea of *multi-result supercompilation* [14]. A supercompiler is a

multi-result one if it produces multiple programs, rather than a single program. Since supercompilers have many decision points, a single-result supercompiler can be transformed into a multi-result one by making it follow several possible paths, rather than choosing a single one by using some heuristics. This idea is fruitful for program analysis because we rarely know for sure what heuristics will bring us a more analyzable program.

The idea of multiresultness has also arisen in the fields different from supercompilation. For example, in the paper "Equality Saturation: A New Approach to Optimization" [20] a method of optimizing imperative programs is described which essentially consists of two steps: building a set of transformed programs and choosing the best one among them. A notable thing about this method is that it uses a smart representation of the set of programs which exponentially reduces the amount of space needed to represent it by sharing common subprograms.

Originally multi-result supercompilation missed this smart representation of the set of programs (however MRSC uses a spaghetti stack representation which partially solves the problem [13]). In our previous paper [3] an attempt to employ this kind of representation was made. There the data structure representing a set of programs was called an overgraph (which was essentially a graph of configurations cleared of restrictions which had been imposed on it by traditional single-result supercompilation). Although it was an obvious step toward the equality saturation approach, there were still many differences. This work is the result of analyzing these differences. Here we present supercompilation in a completely different way putting it on a graph transformation basis.

The paper is structured as follows. In Section 2 we discuss sharing-enabling representations of program sets. In Section 3 we show how a program can be represented as a hypergraph. In Section 4 we discuss how hypergraphs can be transformed. In Section 5 we introduce a powerful operation called merging by graph isomorphism. In Section 6 we discuss transformation ordering. Then in Section 7 we consider a simple example, and Section 8 concludes the paper.

The source code of an experimental implementation of a hypergraph supercompiler based on the ideas of this paper can be found at `github.com/sergei-grechanik/supercompilation-hypergraph`

2 E-graphs vs Hypergraphs

Good representation of sets of programs is crucial for multi-result program transformers like multi-result supercompilers. For example, consider the expression $a_1 + a_2 + \ldots + a_n$. Assume that the only thing we know about the $+$ operation is that it is associative. Then the number of all possible representations of this expression as a graph is equal to $(n-1)$-th Catalan number. Since each such graph contains $n-1$ functional elements representing the $+$ operation, the total number of functional elements (or hyperedges[1], see below) to represent the whole

[1]We use the number of hyperedges because it grows faster than the number of nodes.

set would be

$$(n-1)C_{n-1} = \frac{n-1}{n}\binom{2n-2}{n-1} = O\left(\frac{4^n}{\sqrt{n}}\right)$$

But this approach does not take into account common subexpressions. For example, these two representations have a common subexpression $a_1 + a_2$:

$$(a_1 + a_2) + (a_3 + a_4) \qquad ((a_1 + a_2) + a_3) + a_4$$

If we merge all the graphs into a single graph using one of the methods described below so that all common subexpressions are shared, the total number of functional elements representing + will be expressed by $(n-1)$-th tetrahedral number:

$$T_{n-1} = \frac{n(n-1)(n+1)}{6} = O(n^3)$$

This is much better, it's even polynomial! (Indeed, each contiguous subexpression of length $m \leq n$ has $m-1$ ways to be represented as a sum of two smaller subexpressions. By summing up we get exactly T_{n-1}.)

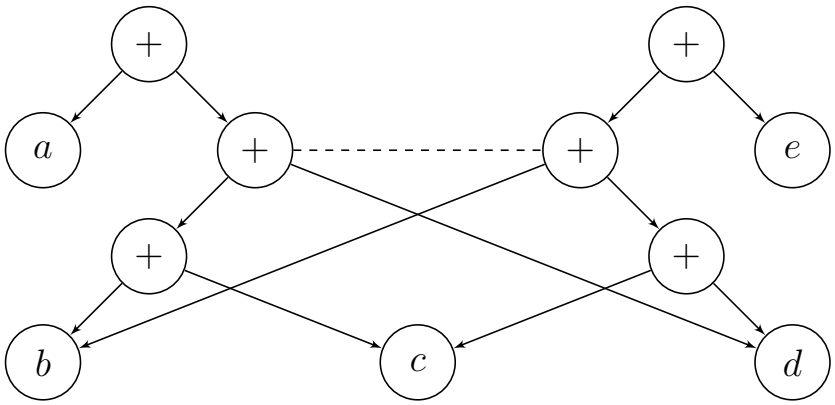

Figure 1: An example of an E-graph representing two expressions, $a + b + c + d$ and $b + c + d + e$, the subexpression $b + c + d$ having two representations

There are several representations of graph sets which enable sharing although all of them are very similar and can be easily transformed to each other. The first one is an *E-graph* [1,8]. An E-graph is a graph together with an equivalence relation defined on the nodes of this graph. Nodes of an E-graph are usually labeled with an operation and connected with edges to nodes representing its arguments. Nodes which are known to represent the same value are put in the same equivalence class. Usually equivalent nodes are drawn to be connected with a dashed line. An example of an E-graph is shown in Figure 1. One of the minor problems of E-graphs can be illustrated with this figure: the subexpression $(b+c)+d$ is part of the expression $a+(b+c+d)$ while the alternative representation of this subexpression $b+(c+d)$ is part of the expression $(b+c+d)+e$ but since the two representations are equal as denoted by the dashed line, either of them could equally be used to form any of the bigger expressions. This makes the E-graph

representation ambiguous which is not a very big problem but is annoying, so we prefer to use other representations.

The cause of this problem is that edges in E-graphs connect functional elements to functional elements while they should really connect them to entire equivalence classes. This leads us to the *bipartite graph* representation. A bipartite graph representing a program or an expression contains nodes of two types: *And-nodes* and *Or-nodes* (and thus graphs of this kind are often called *And-Or-graphs*). Or-nodes represent values and correspond to equivalence classes of E-graphs. And-nodes represent operations and correspond to nodes of E-graphs. Edges going from Or-nodes to And-nodes represent multi-resultness: each such edge corresponds to a representation of a given Or-node. Informally, to compute the value of an Or-node one should choose *any* of the outgoing edges, and hence the name "Or-node". Similarly, an And-node requires *all* of the outgoing edges to compute the value of the node (it's not a precise statement though, as it may be false for lazy languages). Figure 2 depicts a bipartite graph corresponding to the E-graph shown above. Note that although it is useful to label Or-nodes with expressions they represent, they don't really contain any such information.

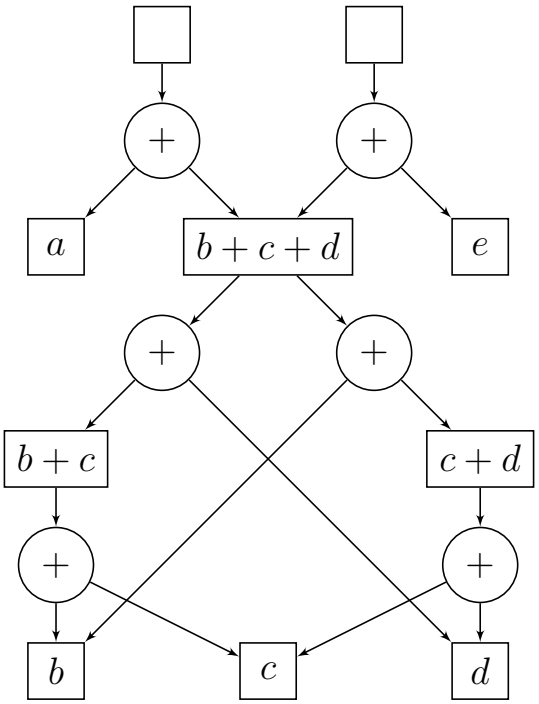

Figure 2: The same expressions represented as a bipartite graph

The last representation we consider is the *hypergraph* representation. A hypergraph is a generalization of a graph with hyperedges in place of edges. Hyperedges differ from edges in that they can connect arbitrary number of nodes (as long as this number is greater than zero). Each hypergraph can be represented as a bipartite graph by replacing every hyperedge with a node with edges connecting it to all the nodes that were connected by the original hyperedge. That is, the difference between the bipartite graph representation and the hypergraph representation is mainly terminological: we now call Or-nodes just *nodes*, and And-nodes together

with incident edges are now *hyperedges*. We prefer this terminology to underline the difference between nodes and hyperedges: nodes behave like metavariables and hyperedges are like statements about these metavariables. Moreover, hyperedges have no identity: if two hyperedges have the same source and destination nodes and the same label then they are indistinguishable. On the other hand, nodes contain no information themselves (except maybe some auxiliary information) but they have identity and we can distinguish among them.

However, we will draw hypergraphs as bipartite graphs as it is easier to draw a hyperedge as a node with edges. Note also that, for the sake of brevity, we will just say *graph* instead of using the term "directed labeled multihypergraph of a certain kind" to describe instances of the data structure we use.

3 Programs as graphs

To describe hypergraph-based supercompilation we will use a lazy first-order subset of Haskell as a source language. All its features are demonstrated by the following piece of code:

$$
\begin{array}{l}
\text{not } b = \\
\quad \textbf{case } b \textbf{ of} \\
\quad\quad T \;\rightarrow\; F \\
\quad\quad F \;\rightarrow\; T \\
\\
\text{even } n = \\
\quad \textbf{case } n \textbf{ of} \\
\quad\quad Z \;\rightarrow\; T \\
\quad\quad S\, m \;\rightarrow\; \text{odd } m \\
\\
\text{odd } n = \text{not (even } m)
\end{array}
$$

Unsurprisingly, we can represent any such program as a graph. Indeed, functions can be represented by nodes, and constructions like "case of" or function calls become hyperedges. Since we work with graphs, recursion is represented simply by cycles. The graph corresponding to the code above is shown in Figure 3.

Several things should be pointed out:

- Intermediate nodes are introduced to split complex expressions into basic constructions.
- Some nodes are shared. In this example only constants and variables are shared but the complexity of shared expressions is not limited.
- Variables are represented by the identity function.

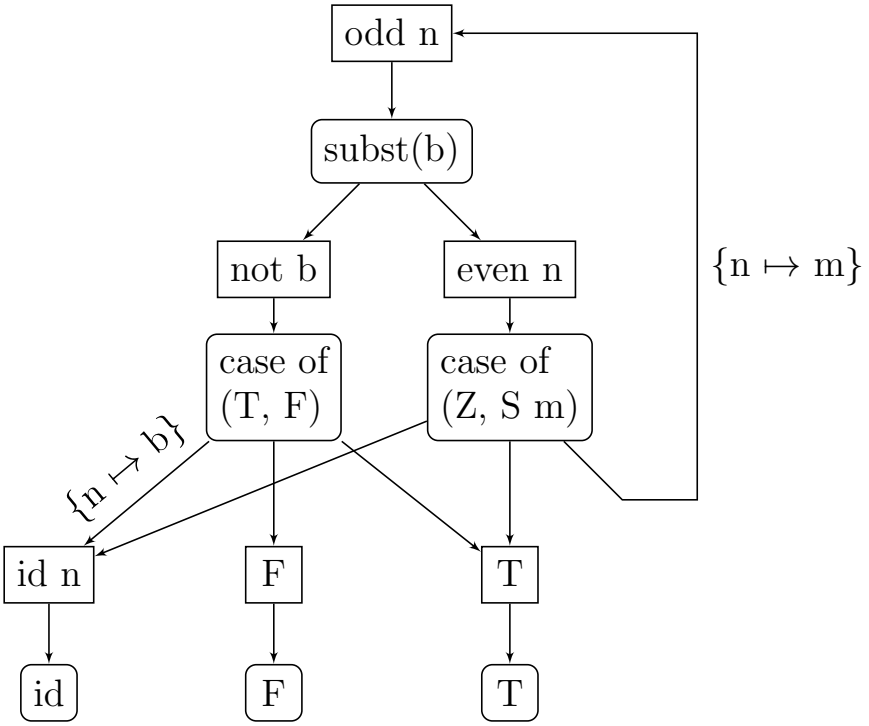

Figure 3: Hypergraph representing the code above

– Each node represents a function of several *named* arguments that returns a single result. Putting it in other words, each node represents an expression with several free variables. It is useful to store names of the arguments inside a node. The actual implementation uses natural numbers instead of names but we will use names for the sake of readability.

– Some edges have renamings on them. Actually, all edges of all hyperedges are labeled with renamings but we don't draw those which happened to be identity. These renamings are important because without them we wouldn't be able to share expressions which differ only in the names of their variables (see Section 4.1).

– There are two possible ways to represent a function call. If all the arguments are just variables then we can simply put a renaming on the edge. But if some of the arguments are complex expressions then we should use an explicit substitution (which is like a let expression but must bind *all* free variables).

– Some hyperedge labels contain parameters. For example, the "subst" label is parameterized by the names of variables being bound. The "case of" label is parameterized by the list of patterns.

– Each node of this graph has a single outgoing hyperedge. This is because it has been converted from a program. It works the other way too: if every node of a graph has exactly one outgoing hyperedge then it can be converted into a program. During the graph transformation phase the graph loses this property and starts to represent a set of programs.

4 Graph evolution

The process of hypergraph-based supercompilation from the bird's eye view consists of three steps:

1. Converting a program being supercompiled into a graph.
2. Transforming the graph.
3. Extracting residual programs from the graph (optional).

The third step is optional because some applications of supercompilation (like proving equivalence of two expressions, see Section 5) don't require residualization. Thus the second step is the most important one.

We mainly follow the equality saturation approach [20]. Graph evolution consists in consecutive application of different transformations. We don't restrict transformations too much: a transformation checks if the graph meets certain conditions and then adds some hyperedges or nodes, removes some hyperedges, and merges some nodes (the operation to be described soon). It is known that if transformations are not allowed to delete hyperedges and their preconditions are monotone then the graph transformation system is confluent, i.e. we can apply transformations in any order and get the same result if we reach saturation (the state when no more transformations can be applied). There are two problems though: first, this approach is too restrictive, sometimes hyperedge deletions are useful, particularly we use transformations which replace hyperedges with their simplified versions; second, it is very hard to reach saturation, at least in the case of a supercompiler. That's why we propose a slightly different solution to the ordering problem which we will describe later in Section 6.

4.1 Node merging

Merging is an operation that glues two nodes into one. It corresponds to merging of equivalence classes of an E-graph. Transformations can perform merging directly but usually it is performed automatically when there are two different nodes with equal outgoing hyperedges (i.e. with the same destination nodes and labels). Merging of two nodes may lead to more hyperedges turning out to be equal thus triggering a chain reaction (Figure 4).

All incident hyperedges are reattached to their renewed source or destination nodes during the process of merging, and since hyperedges don't have identity, equal hyperedges are automatically merged. Note that in terms of E-graphs the result of this chain reaction will be the congruence closure of the associated equivalence relation on nodes [18].

In the domain of supercompilation for functional languages merging has a peculiarity: nodes should be merged modulo renaming (modulo invertible renaming, to be precise). Indeed, an expression with n free variables can be represented in

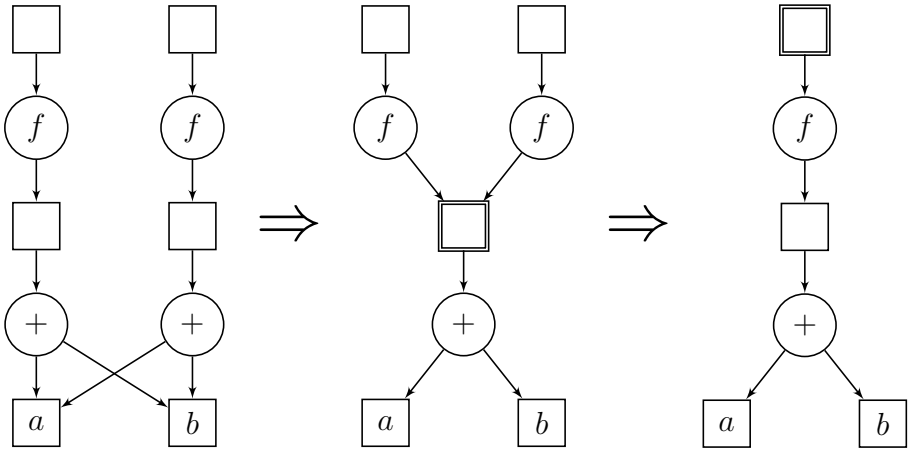

Figure 4: A chain reaction of node merging

$n!$ ways which differ only in the order of variables. Fortunately, merging modulo renaming is easy when hyperedges are labeled with renamings, as we just have to adjust them when reattaching hyperedges as shown in Figure 5.

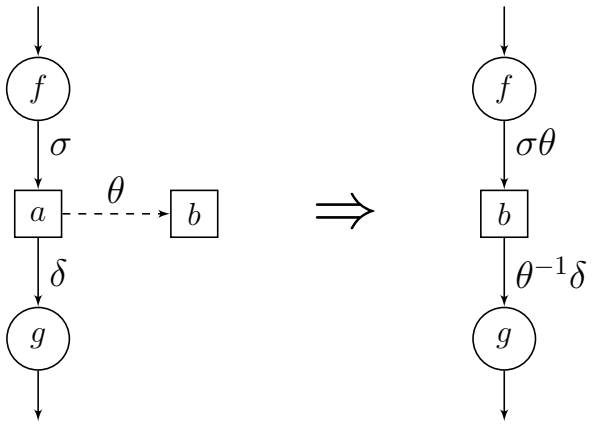

Figure 5: Adjustment of renamings during the reattachment of hyperedges

4.2 Transformations

We split up all transformations into three groups. The first one is a group of transformations that simplify hyperedges by *replacing* them with simpler ones. An example of such transformation is replacing a substitution that substitutes only variables with a simple renaming. These transformations are very useful in practice because they considerably simplify the graph. We won't consider these transformations further because they are usually concerned with a lot of technical details, but it is important to bear in mind their existence as they undermine the simple monotonicity-based solution to the transformation ordering problem.

The second group includes transformations which do most of the work, namely driving and generalization. They have monotone preconditions and only *add* nodes and hyperedges. The third group is a group of a single transformation which we call *merging by graph isomorphism*. It corresponds to residualization

(subst-id)	$x\ \{x = A\} \mapsto A$
(subst-subst)	$E\ \{x = A\}\ \{y = B\} \mapsto E\ \{x = A\ \{y = B\}\}$
(subst-constr)	$(C\ E_1\ E_2)\ \{x = A\} \mapsto C\ (E_1\ \{x = A\})\ (E_2\ \{x = A\})$
(subst-case-of)	$(\mathbf{case}\ E_1\ \mathbf{of}\ C\ y\ z \to E_2)\ \{x = A\} \mapsto$ $(\mathbf{case}\ E_1\ \{x = A\}\ \mathbf{of}\ C\ y\ z \to E_2\ \{x = A\})$
(case-of-constr)	$(\mathbf{case}\ C\ E_1\ E_2\ \mathbf{of}\ C\ y\ z \to E_3) \mapsto$ $E_3\ \{y = E_1,\ z = E_2\}$
(case-of-case-of)	$(\mathbf{case}\ (\mathbf{case}\ E_1\ \mathbf{of}\ C\ y\ z \to E_2)\ \mathbf{of}\ C\ s\ t \to F) \mapsto$ $(\mathbf{case}\ E_1\ \mathbf{of}\ C\ y\ z \to (\mathbf{case}\ E_2\ \mathbf{of}\ C\ s\ t \to F))$
(case-of-id)	$(\mathbf{case}\ x\ \mathbf{of}\ C\ y\ z \to E) \mapsto$ $(\mathbf{case}\ x\ \mathbf{of}\ C\ y\ z \to E\ \{x = C\ y\ z\})$

Figure 6: Transformations which enable driving

and enables higher-level supercompilation. We will discuss merging by graph isomorphism in the next section while devoting the rest of this section to the second group.

In traditional supercompilers operations like driving or generalization work with whole configurations, which are usually just expressions with free variables. In the case of multi-result supercompilation based on hypergraph transformation, operations should be more fine-grained as configurations are dissipated over the graph in a form of individual hyperedges. With explicit substitutions this can be achieved quite easily. Most of the transformations we use to perform driving and generalization require two hyperedges with one common node as a precondition, and their effect is usually swapping of these two hyperedges (the original hyperedges are preserved, of course).

Although transformations work on graphs, we will write them as term rewriting rules in the form *original-term* $\mapsto$ *new-term* for brevity (we also describe them quite informally pretending that we have only single binary constructor (C), no renamings on edges, and the problem of variable name collisions doesn't exist). To describe terms we use the syntax of the simple lazy language presented above with a single exception: instead of function calls we use explicit substitutions, like this:

$$E\ \{x = A,\ y = B\}$$

The term rewriting rules can be interpreted as graphs transformations quite easily. Let's consider the first (and the simplest) transformation subst-id (Figure 6). It states that if we substitute A for x then we get A, or in other words if we call the id function then we get the argument. The effect of applying this transformation to a graph is shown in Figure 7. The effect of this operation is just merging of the nodes N (which represents the left hand side) and A (which is exactly the right hand side). Note that it doesn't delete the original substitu-

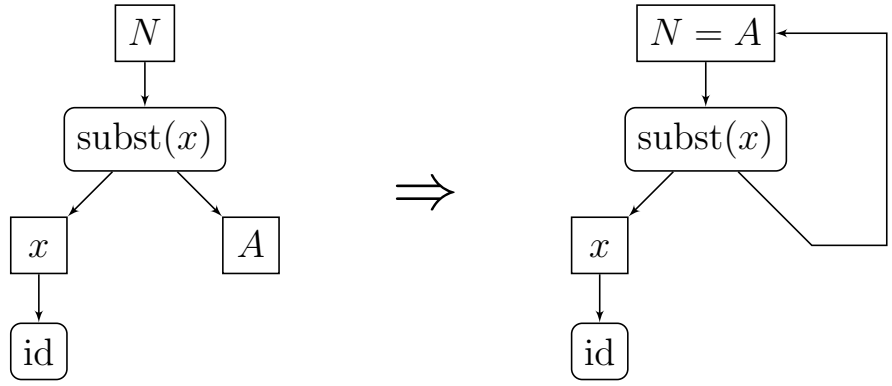

Figure 7: Applying subst-id to a graph

$$
\begin{array}{rl}
\text{(id-subst)} & A \;\mapsto\; x\;\{x = A\} \\
\text{(subst-subst-inv)} & E\;\{x = A\;\{y = B\}\} \;\mapsto\; E\;\{x = A\}\;\{y = B\} \\
\text{(constr-subst)} & C\;(E_1\;\{x = A\})\;E_2 \;\mapsto\; (C\;E_1\;E_2)\;\{x = A\} \\
\text{(case-of-subst-1)} & (\textbf{case}\;E_1\;\{x = A\}\;\textbf{of}\;C\;y\;z \to E_2) \;\mapsto \\
 & \qquad (\textbf{case}\;E_1\;\textbf{of}\;C\;y\;z \to E_2)\;\{x = A\} \\
\text{(case-of-subst-2)} & (\textbf{case}\;E_1\;\textbf{of}\;C\;y\;z \to E_2\;\{x = A\}) \;\mapsto \\
 & \qquad (\textbf{case}\;E_1\;\textbf{of}\;C\;y\;z \to E_2)\;\{x = A\} \\
\text{(share)} & E\;\{x = A,\; y = A\} \;\mapsto\; E\;\{x = z,\; y = z\}\;\{z = A\}
\end{array}
$$

Figure 8: Transformations which enable generalization

tion hyperedge. Other transformations would usually add some new hyperedges growing from N instead of merging.

Among the transformations the most important are those which enable driving (Figure 6). These transformations essentially describe the evaluation rules for our language except the last transformation, case-of-id, which represents propagation of positive information.

Generalization is also important for supercompilation but in the case of hypergraphs configurations are already in a sort of generalized state, i.e. we are allowed to transform any subexpression of a given expression. For example consider a composition of three functions $f \circ (g \circ h)$. Its proper subexpressions are f, g, h and $g \circ h$. Each such subexpression is represented by a node. However, $f \circ g$ is not a subexpression of $f \circ (g \circ h)$, so the task of generalization in a hypergraph supercompiler is to rewrite the original expression into an equivalent but different form $(f \circ g) \circ h$, which does contain $f \circ g$ as a subexpression. This can be done with the help of transformations from Figure 8 (again, we omit some details, especially those concerned with name collisions; moreover in the implementation a slightly different set of transformations is used). Their main idea is to lift substitutions up, in contrast to driving which pushes them down

It is worth mentioning that generalization in the context of multi-result super-

compilation is a very hard problem as it produces a lot of different expressions, and sharing doesn't always help. For this very reason saturation is hard to achieve, and more delicate solutions are needed.

5 Merging by graph isomorphism

Local transformations like those presented in the previous section are necessary, but not sufficient. It's not difficult to see why. For instance, consider the following two zero-arity functions which build an infinite "number":

$$\text{inf1} = S\ \text{inf1}$$
$$\text{inf2} = S\ \text{inf2}$$

The two are obviously equal but they will not be merged by the hypergraph supercompiler if only the techniques described so far are used. The automatic node merging method described in Section 4.1 guarantees that nodes that have equal *non-recursive* definitions will be merged, but each of these two nodes is *recursively* defined in terms of itself (Figure 9) and there is no transformation that can convert a recursive definition into a non-recursive one.

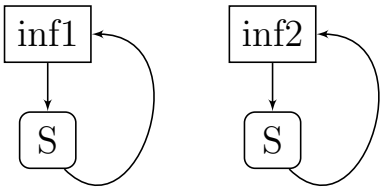

Figure 9: An example of two equal nodes that cannot be easily merged

There are at least two possible ways to solve the problem. The first one is to introduce a transformation that would convert a recursive definition into a non-recursive definition (for example using an explicit fixed-point combinator and higher-order functions). The second one is to add a recursion-aware merging operation. We've followed the second way (partly because we didn't want to complicate our language with higher-order functions yet).

We call this kind of merging operation *merging by graph isomorphism* (or *merging by isomorphism* for short). It takes two nodes and tries to find two isomorphic subgraphs growing from them and meeting certain additional conditions. If it succeeds in finding these subgraphs then the two nodes are being merged. Merging by isomorphism is related to traditional residualization. Indeed, the found subgraphs can be considered residual programs, and the merging by isomorphism algorithm can be easily altered to return a set of residual programs for a single node. It is also possible to build a merging by isomorphism operation using residualization – we just need to build two sets of residual programs for the given nodes and then look if they have a non-empty intersection. It is a well-known trick in the field

```
function MERGE-BY-ISOMORPHISM(m, n)
    if ISOMORPHIC?(m, n, ∅) then
        MERGE(m, n)

function ISOMORPHIC?(m, n, history)
    if m = n then
         return true
    else if (m, n) ∈ history and the safety condition is met then
        return true
    else if m and n have incompatible hyperedges then
        return false
    else
        Check all the pairs of outgoing hyperedges:
        for h1 ← out(m), h2 ← out(n) do
            children = zip(dests(h1), dests(h2))
            if label(h1) = label(h2) and
                    ∀ (m', n') ∈ children
                        ISOMORPHIC?(m', n', {(m, n)} ∪ history)  then
                return true
        return false
```

Figure 10: Merging by graph isomorphism algorithm

of supercompilation but we do not recommend doing this since it is much more efficient to check nodes for equivalence directly.

The merging by isomorphism algorithm is outlined in Figure 10. The merging by isomorphism function calls the function ISOMORPHIC? (which does all the work) and merges the given nodes if it returns true. The function ISOMORPHIC? checks if there are two isomorphic subgraphs growing from the nodes. The idea is to simultaneously traverse the graph from the given nodes by following hyperedges with equal labels. Note though that, since any merging is done modulo renaming, and merging by isomorphism is not an exception, the equality between the corresponding hyperedges should be checked up to renaming, but for simplicity we do not take this into account here.

There are several shortcuts in the presented algorithm. First, if the two nodes are equal non-recursively then we don't have to further check their equivalence. Second, if the nodes have incompatible outgoing hyperedges then they can't be equal and we return false. As an example of incompatible hyperedges consider two hyperedges representing different constructors. Since a function cannot issue two different constructors simultaneously, their source nodes cannot be equal.

Another important point is the safety condition that is checked when we encounter a pair of nodes that we have already seen (this situation somewhat corresponds to folding in traditional supercompilation). We have to do this because there may be vacuous cycles in the graph. For example a hyperedge of the fol-

lowing form can appear during application of transformations described earlier:

$$A = x\ \{x = A\}$$

This is a tautology which is absolutely correct but cannot be used as a definition of A. So, the task of the safety condition is to filter out this kind of cycles. In our current implementation we use guarded and structural recursion as safety conditions. Note that there are other possible ways of ensuring safety, for example, using the tick algebra [12, 19].

Merging by isomorphism is a very important operation because it makes hypergraph-based supercompilation at least as powerful as traditional supercompilation. It also makes it possible to use hypergraph supercompilation for proving program equivalence without performing actual residualization. To do this one should put the functions suspected to be equal in the same graph and then transform it until the nodes representing these functions are merged.

The most amazing thing about merging by isomorphism is that it automatically enables higher-level supercompilation. Indeed, the idea of higher-level supercompilation is to use a lower-level supercompiler to prove some equalities (lemmas) which then can be used by the higher-level supercompiler. But this is exactly what happens if we apply merging by isomorphism several times: earlier merging may facilitate further merging.

6 Transformation ordering

Traditional supercompilers have a very strict order of applying transformations: perform driving until the whistle blows (a termination heuristic), then either fold or generalize the current (or sometimes the upper) configuration and continue driving.

Equality saturation approach, on the other hand, allows applying transformations in any order since the result will be the same if saturation (the state when no transformation is applicable) is reached. However, applying this method for supercompilation directly is complicated by several difficulties:

- Driving can be applied infinite number of times producing infinite number of different nodes. This problem can be overcome by using termination heuristics (whistles). For example, a simple heuristic is to perform driving only if the depth of the node is less than a certain number.

- Not all of our transformations meet the condition required for ordering-independence to hold. Actually it doesn't mean that our system is not ordering-independent, it means that this property is now much harder to substantiate and it's better not to rely on it.

- Even if we use whistles, saturation is very hard to achieve in practice. This is mostly due to generalizations as they produce a huge (but finite) number of different nodes.

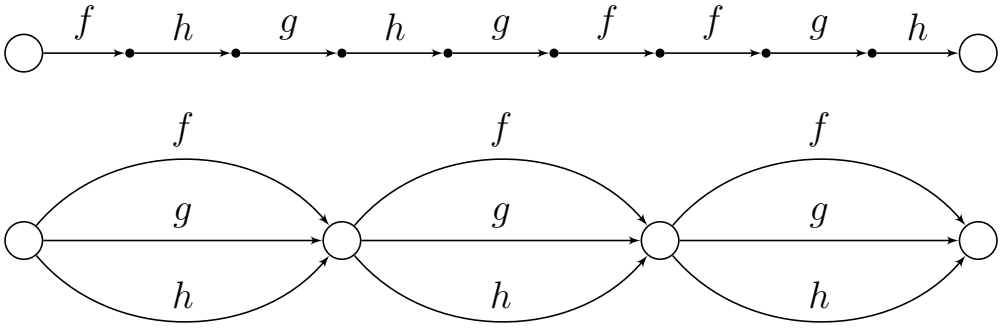

Figure 11: Two approaches to saturation: applying transformations sequentially and applying transformations in parallel, generation by generation

Since the first two issues are more or less solvable, we will pay more attention to the last one. Ideally, we would like to have a fine-grained control over the time (number of steps) it takes to reach the saturated state. We've tried to use the depth-based heuristics we used for driving, but they turned out to be too coarse. So we decided to use another solution.

We propose to apply transformations in parallel: given a graph we determine what transformations can be applied to it and what new hyperedges should be added, and then we add all the new hyperedges (the new generation of hyperedges) at once. Then we repeat the procedure, thus adding hyperedges to the graph, generation by generation (Figure 11). Each new generation depends only on previous generations. This approach has certain advantages:

– Since transformations don't interfere when we build a new generation, we can relax the monotonicity condition. It opens opportunities for complex whistles that can use information about the whole graph.

– The number of hyperedges in a single generation is not that big, and so we have a more fine-grained control over time we would like to spend on supercompilation. Still this number grows quite rapidly with the generation index, so it is not a complete solution.

– We can stop and think between generations. During this break we can analyze the graph and adjust further supercompilation strategy, we can apply some complex transformations like merging by isomorphism, or we can check if we've already achieved the goal (for example if we've merged the nodes we wanted to prove equal) and should stop.

We don't claim that this is the only right way, but at least it deals with some issues. Note though that we cannot do without heuristics, and traditional supercompilation heuristics are hard to make work in hypergraph setting. Building *practical* hypergraph-based supercompilers using the machinery described in this paper is still largely unresearched.

7 Example

In this section we consider an example of supercompiling the following function:

$$\begin{aligned} f\,n = \textbf{case}\ n\ \textbf{of}& \\ Z &\to Z \\ S\,n &\to f\,(f\,n) \end{aligned}$$

This function unwinds its argument until it reaches zero (Z) and then returns Z, but does it in an inefficient way by running it through itself. Indeed, here is the reduction of the term $f\,(S\,(S\,Z))$:

$$\begin{aligned} &f\,(S\,(S\,Z)) \to \\ &f\,(f\,(S\,Z)) \to \\ &f\,(f\,(f\,Z)) \to \\ &f\,(f\,Z) \to \\ &f\,Z \to \\ &Z \end{aligned}$$

We will prove that the function f is equal to the following function g which does the same thing more efficiently:

$$\begin{aligned} g\,n = \textbf{case}\ n\ \textbf{of}& \\ Z &\to Z \\ S\,n &\to g\,n \end{aligned}$$

The equivalence of f and g can be proved by induction using the following lemma:

$$f\,(f\,n) = f\,n$$

which in its turn can also be proved by induction.

This example is very simple and yet it needs higher-level supercompilation (because it uses a lemma). Moreover, the lemma is not easy to prove by supercompilation using the standard approach that implies supercompiling both sides and then comparing the residual programs. However, hypergraph-based supercompilation easily solves this example. Actually it doesn't even need generalization to solve it.

We will mostly follow the steps our current experimental implementation performs. The first step is to convert both functions into a hypergraph. The result can be seen in Figure 12. Nodes are drawn as rectangles, and hyperedges are drawn as rounded rectangles with sections for arguments at the bottom. We place small arguments like variables and constants right in these sections although they really have their own nodes and hyperedges. If an argument is too big, we put an ellipsis (...) in the corresponding section and draw an outgoing arrow.

We will not draw the subgraph representing the function g on the subsequent figures as it is already in the fully saturated state and will not change until the last

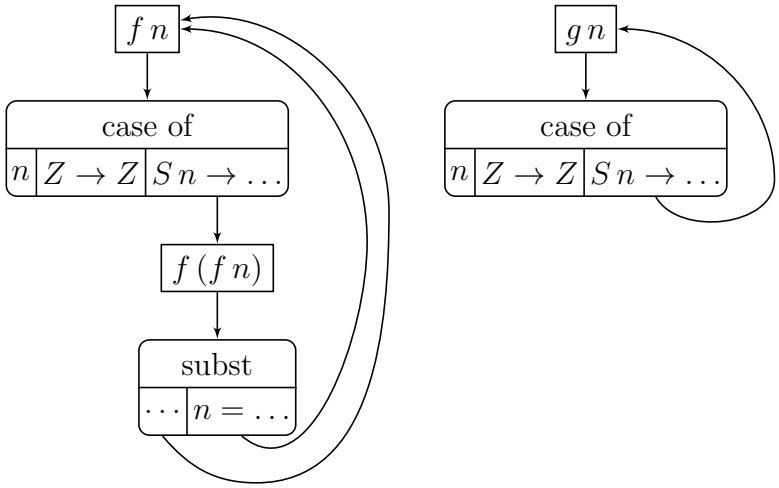

Figure 12: Initial graph (the zeroth generation of hyperedges)

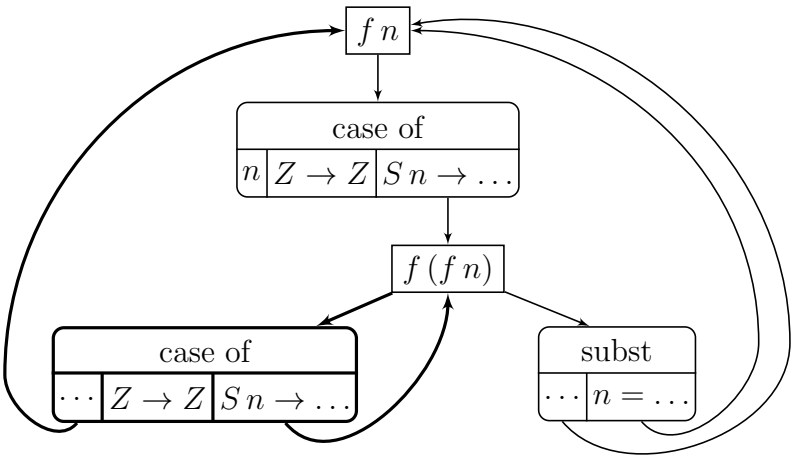

Figure 13: After the first iteration (the first generation of hyperedges)

step. All transformations of interest are happening with the subgraph representing the function f.

After the functions has been converted to graphs, transformations are applied to them. In this case we use only driving. The result of the first iteration of applying transformations is shown in Figure 13 (new hyperedges are drawn with thicker lines). There is only one new hyperedge, which is the result of driving $f\,(f\,n)$, stating that:

$$\begin{aligned} f\,(f\,n) = \mathbf{case}\ f\,n\ \mathbf{of}& \\ Z &\ \rightarrow\ Z \\ S\,n &\ \rightarrow\ f\,(f\,n) \end{aligned}$$

This hyperedge is the result of applying the transformation (subst-case-of) (see Figure 6) to the hyperedges of the original graph with subsequent normalization of the resultant hyperedge using the transformation (subst-id). In our current implementation the transformations (subst-id) and (case-of-constr) are applied automatically, the original hyperedges being removed from the graph.

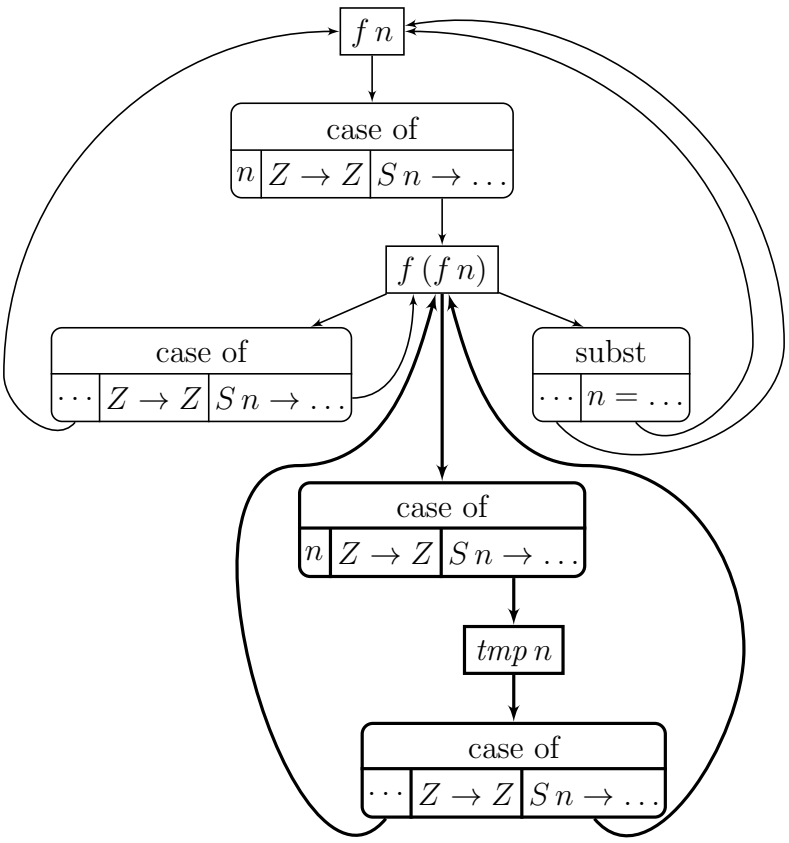

Figure 14: After the second iteration (the second generation of hyperedges)

Textually this transformation would look as follows:

$$
\begin{aligned}
& f\,(f\,n) = (f\,n)\,\{n = f\,n\} = \\
& \quad (\textbf{case}\ n\ \textbf{of}\ \{Z\ \to\ Z;\ S\,n\ \to\ f\,(f\,n)\})\,\{n = f\,n\} = && \text{(subst-case-of)} \\
& \quad \textbf{case}\ n\ \{n = f\,n\}\ \textbf{of}\ \{Z\ \to\ Z;\ S\,n\ \to\ f\,(f\,n)\} = && \text{(subst-id)} \\
& \quad \textbf{case}\ f\,n\ \textbf{of}\ \{Z\ \to\ Z;\ S\,n\ \to\ f\,(f\,n)\}
\end{aligned}
$$

Note that the substitution doesn't propagate into case-of's branches because they don't use the variable n (in the S branch n is bound). Tracking sets of used variables is another important trick that prevents the graph from becoming over-complicated.

In the second iteration further driving of $f\,(f\,n)$ is performed (using the transformation (case-of-case-of)), resulting in the graph shown in Figure 14. In this case an intermediate node ($tmp\,n$) was needed.

It might not be obvious from the first glance, but now we have everything prepared to perform merging by isomorphism, i.e. we have two isomorphic subgraphs growing from the nodes $f\,n$ and $f\,(f\,n)$. The corresponding proof graph (which almost repeats the execution tree of the function ISOMORPHIC?) is shown in Figure 15. Note that this graph has a leaf node $f\,(f\,n) = f\,(f\,n)$. It doesn't have any outgoing hyperedges since the statement it represents is obviously true. This

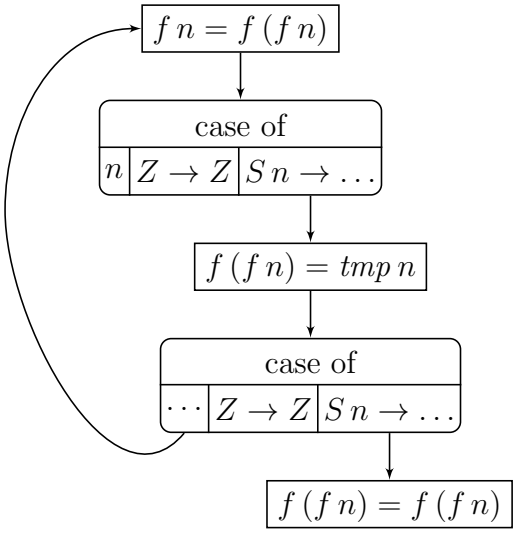

Figure 15: A proof graph for the equivalence $f\,n = f\,(f\,n)$

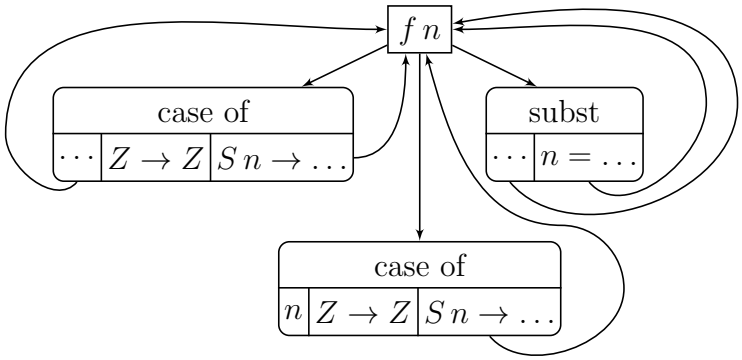

Figure 16: The final state

is an example of a shortcut that is not taken into account in the usual approach to proving equivalence by supercompilation.

After performing merging by isomorphism, the graph collapses to the state shown in Figure 16 where the nodes $f\,n$, $f\,(f\,n)$ and $tmp\,n$ are identified. The last step is to perform another merging by isomorphism to merge the nodes $f\,n$ and $g\,n$. As a result, the graph will look exactly like the graph in Figure 16, moreover, it will be in the fully saturated state.

8 Conclusion and related work

The main contribution of our work is expressing multi-result supercompilation in terms of hypergraph transformations. As it has been shown, this new formulation gives us higher-level supercompilation "for free" by means of merging by isomorphism.

Because of the important role merging plays in hypergraph-based supercompilation, it seems that its most natural application is proving equivalence of programs. Residualization in multi-result setting still has some issues like bad per-

formance and lack of good heuristics for choosing the best program.

The safety condition we have used for merging by isomorphism suggests that our form of supercompilation may be better suited to total languages (like Agda), but it's still an open question.

This paper may be considered as a continuation of our previous work [3], in which we used a special kind of hypergraph called overgraph. The main distinctive feature of an overgraph is that its nodes contain configurations. Now we have got rid of configurations which makes it possible to merge more nodes. However, some ideas from the previous article are still applicable, e.g. the residualization algorithm.

The idea that nodes should be merged modulo renaming (already used in our previous work) is extremely important for hypergraph-based supercompilation, although we haven't payed much attention to it in this paper for the sake of simplicity.

The idea of using a graph-based data structure to compactly represent large sets of programs (or similar objects) is not new, but usually E-graphs are used for this purpose [1, 8, 20]. However And-Or-graphs (which are closer to hypergraph) are also used, for example in the field of transition systems [2] for applications related to multi-result supercompilation [9].

The idea of multi-result supercompilation was put forward by I. Klyuchnikov and S. Romanenko [14], mainly to aid higher-level supercompilation. Higher-level supercompilation is a broad term denoting systems that use supercompilation as a primitive operation, but we used it here in a narrow sense as a supercompilation technique similar to two-level supercompilation and distillation. Distillation was introduced by G. Hamilton [4, 6, 7]. Later I. Klyuchnikov and S. Romanenko proposed two-level supercompilation based on similar ideas [10, 12]. Our version of higher-level supercompilation is more similar to the recent definition of distillation [5] in several respects, particularly in the ability of gradually moving to higher levels. We also do not have a clear distinction between different levels, as opposed to two-level supercompilation.

Two-level supercompilation uses a tick-based approach to ensuring transformation correctness, which is the most rigorous approach so far. It may work better for some programs than the approach we use, since the syntactic conditions of guarded and structural recursion reject some programs even for single-level supercompilation, so we are thinking of employing the tick-based approach in the hypergraph supercompiler. Note thought that the tick-based approach has certain issues with levels higher than two, so it may be more effective to combine both methods. It is worth noting that the guardedness condition in the field of supercompilation has also been used to check if the supercompiled program is productive (which implies that the original program is productive too) in the work by G. Mendel-Gleason and G. Hamilton [17].

Acknowledgements

The author would like to express his gratitude to Sergei Romanenko and Andrei Klimov for regular fruitful discussions on this work and useful comments.

References

[1] Detlefs, Nelson, and Saxe. Simplify: A theorem prover for program checking. *JACM: Journal of the ACM*, 52, 2005.

[2] G. Geeraerts. Coverability and expressiveness properties of well-structured transition systems, May 02 2007.

[3] 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.

[4] 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.

[5] G. W. Hamilton. A hierarchy of program transformers. In A. Klimov and S. Romanenko, editors, *Proceedings of the Third International Valentin Turchin Workshop on Metacomputation*, pages 66–86, Pereslavl-Zalessky, Russia, July 2012. Pereslavl-Zalessky: Ailamazyan University of Pereslavl.

[6] G. W. Hamilton and N. D. Jones. Distillation with labelled transition systems. In O. Kiselyov and S. Thompson, editors, *PEPM*, pages 15–24. ACM, 2012.

[7] G. W. Hamilton and M. H. Kabir. Constructing programs from metasystem transition proofs. In *Proceedings of the First International Workshop on Metacomputation in Russia*, 2008.

[8] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. *ACM SIGPLAN Notices*, 37(5):304–314, May 2002.

[9] A. V. Klimov. Why multi-result supercompilation matters: Case study of reachability problems for transition systems. In A. Klimov and S. Romanenko, editors, *Proceedings of the Third International Valentin Turchin Workshop on Metacomputation*, pages 91–111, Pereslavl-Zalessky, Russia, July 2012. Pereslavl-Zalessky: Ailamazyan University of Pereslavl.

[10] 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`.

[11] 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.

[12] I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.

[13] I. G. Klyuchnikov and S. A. Romanenko. Formalizing and implementing multi-result supercompilation. In A. Klimov and S. Romanenko, editors, *Proceedings of the Third International Valentin Turchin Workshop on Metacomputation*, pages 142–164, Pereslavl-Zalessky, Russia, July 2012. Pereslavl-Zalessky: Ailamazyan University of Pereslavl.

[14] 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.

[15] A. Lisitsa and A. P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.

[16] 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.

[17] G. E. Mendel-Gleason and G. W. Hamilton. Development of the productive forces. In A. Klimov and S. Romanenko, editors, *Proceedings of the Third International Valentin Turchin Workshop on Metacomputation*, pages 184–202, Pereslavl-Zalessky, Russia, July 2012. Pereslavl-Zalessky: Ailamazyan University of Pereslavl.

[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] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.

[21] V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.