Overgraph Representation for Multi-Result Supercompilation*

Sergei Grechanik

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

Abstract. In this paper we present a new approach to multi-result supercompilation based on joining together process graphs into a single graph and welding nodes corresponding to equal configurations. We show that this leads to a considerable reduction of nodes being built during supercompilation and enables more efficient configuration graph processing algorithms on the example of an enhanced residualization algorithm.

1 Introduction

Supercompilation [15] is traditionally seen as a program transformation which produces a single program that is equivalent in some sense to the original one. This approach is very understandable: we usually need only one program, actually the most optimal one we can produce. But this approach goes deep down the process of supercompilation. Being a complex transformation, supercompilation consists of smaller steps (like driving, folding and generalization) that can be applied in some order. The principle of single result dictates us to choose a single step each time we have a choice. This means that we ought to make decisions using some a priori heuristics which may lead us to a solution that is far from optimal. Of course there is no purity in the world and supercompilers often implement backtracking through generalization of an upper configuration.

The flaw of this approach becomes more pronounced when we consider a problem that doesn't consist in finding an optimal solution, for example proving of equivalence of two programs. The standard approach is to supercompile both of them and then compare the resultant programs syntactically [6, 10]. But we don't really need an optimal solution to do it, and since *the* optimal solution is hard to find, we would increase chances of success by constructing a set of equivalent programs for each original program and then checking if the sets intersect. Another example – program analysis. In this case a supercompiler can be used as a transformation that simplifies a program into one that is easier

* 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.

to analyze. But "easier to analyze" might not be the same as "more efficient" and it may be more difficult to express using heuristics.

Undoubtedly it is possible to stuff the supercompiler with all sorts of heuristics and then tune it to suit the specific task. This approach has the right to live. It must be said that it is also used outside the field of supercompilation, for example, it is common to automatically adjust the set of parameters of optimising compilers to a certain architecture or even to a set of programs [2, 3, 11].

An alternative approach is to dismiss the idea of the single path to the optimal program. Let supercompiler perform all possible steps simultaneously when it has a choice. This idea is known as the multi-result supercompilation [9]. It first appeared not a very long time ago and was intended to be used primarily for program analysis. It should be noted that similar idea had appeared in the field of optimizing compilers [14] where it enabled selecting the best program by evaluating the programs a posteriori using global profitability heuristics; this indicates that the idea of multiple results is advantageous for optimization problems.

The main problem of multi-resultness is lack of efficiency. Each branching point multiplies the number of programs leading to combinatorial explosion. This issue can be resolved by restricting branching using heuristics (which is not desirable but seems unavoidable) or by using some better representation of the set of programs to diminish redundancy. In this paper the latter approach is discussed.

2 The Approach of MRSC

MRSC is a multi-result supercompilation framework written in Scala [8][1]. The goal of MRSC is to provide a set of components to rapidly build various multi-result supercompilers. MRSC consists of a core which implements basic domain-independent operations over process graphs, and several domain-specific (i.e. specific to different languages and tasks) modules which give meaning to these graphs and guide their construction.

MRSC is based on explicit construction of configuration graphs. Being a multi-result supercompiler, it takes an initial configuration and produces a list of corresponding configuration graphs which is then transformed into a list of programs through the process known as residualization. MRSC issues configuration graphs incrementally. In particular, this design choice makes it possible to use MRSC as a traditional single-result supercompiler without efficiency loss by simply taking the first issued graph. On the other hand it determines that MRSC should use depth-first traversal to build and issue every next graph as quickly as possible. Thus, the MRSC supercompilation algorithm can be formulated using a stack of graphs as follows:

1. Put a graph with only one node containing the initial configuration on the stack.

[1] The source code of MRSC is available at `https://github.com/ilya-klyuchnikov/mrsc`

2. While the stack is not empty repeat the following steps.
3. Pop a graph g from the stack.
4. If the graph g is completed, issue it.
5. If the graph is incomplete, transform it according to some domain-specific rules and push the resultant graphs (there may be many of them as the supercompilation is multi-result) onto the stack

It should be noted that although we talk about graphs they are actually represented as trees with back edges (which are represented differently from normal edges).

Each graph has a list of complete nodes and a list of incomplete nodes. The first incomplete node of a graph is called the current node and represents the place where the graph is growing. A graph is completed if it doesn't contain incomplete nodes.

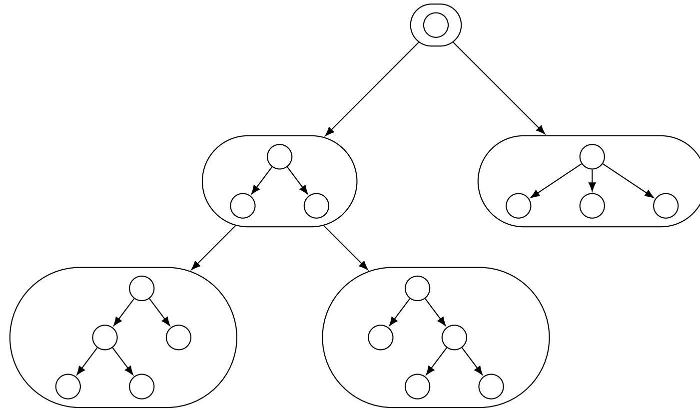

Fig. 1. Tree of graphs

Graph transforming rules are specified with a function that maps a graph into a list of steps to perform on it (by "perform" we don't mean that they actually modify it, we apply steps in a functional way). When a step is performed, the current node usually becomes complete and the next incomplete node becomes current (if there are incomplete nodes left). A step can be one of the following:

1. *CompleteCurrentNode* – just mark the current node complete and move on to the next incomplete node. It is usually used if the the current node contains a fully evaluated expression as a configuration.
2. *Fold*(b) – make a back edge from the current node to b.
3. *AddChildNodes*(cs) – append nodes to the current one.
4. *Rebuild*(c) – replace the configuration of the current node with c. This step does not make the current node complete.

5. *Rollback*(n, c) – perform an upper rebuilding: remove the subgraph that grows from the node n and then replace the configuration of n with c. n becomes current.

If there is no step produced by the rules function, the graph is thrown away as it is incomplete and cannot be completed. It is usually the case when the whistle blows but there is no possibility of generalizing, or the graph doesn't meet some conditions we require (like the safety condition in the case of counter systems).

So what if the rules prescribe several different steps to one incomplete graph? All of them will be performed on it simultaneously, producing several graphs. This leads to the idea of a tree of graphs (Fig. 1). The tree of graphs is a mental construction rather than an actual tree that resides in a memory, but it can be used to give another formulation of MRSC supercompilation algorithm: MRSC just performs the depth-first traversal of this tree filtering out incomplete graphs.

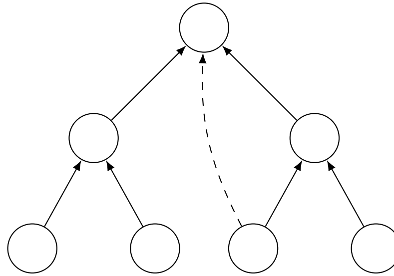

Fig. 2. Spaghetti-stack-based graph representation

When more than one step is applicable the number of graphs multiplies by the number of steps, and it may seem that MRSC doesn't cope with the problem of combinatorial explosion very well. But actually MRSC uses a clever representation to reduce memory usage by exploiting sharing of common subgraphs. This representation is based on spaghetti-stacks [1]. A graph represented this way can be thought of as a configuration graph with all edges reversed (except folding edges which are treated separately) (Fig. 2). This allows initial parts of graphs to be shared. Moreover this representation makes it possible to work with graphs in a functional way efficiently. Note that this optimization doesn't interfere with our view on graphs as separate objects.

3 Can we do better?

Certainly there is a room for improvement. Let us look when MRSC does not work very well. Consider a configuration that being driven produces two child nodes: b and c. Then the node b will become current. Let it have multiple different steps that can be applied to it. We get at least two different graphs with

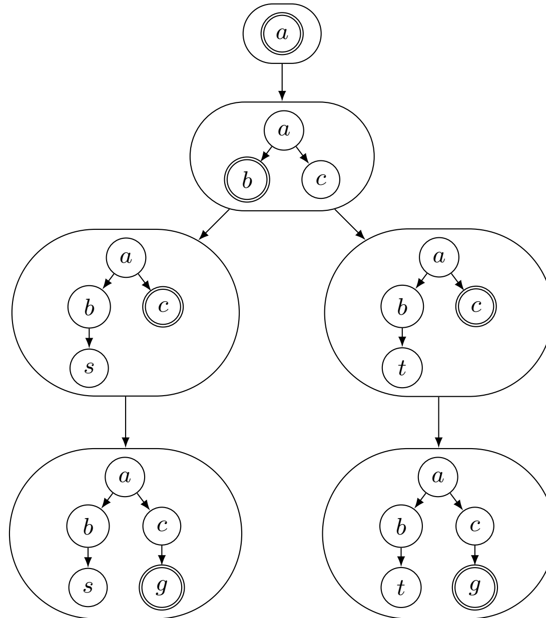

Fig. 3. Two different nodes with the same configuration and identical histories

incomplete node c (Fig. 3). That means that the c node will be current at least twice (in different graphs) and thus each subgraph growing from it will be built at least twice and won't be shared, which might be considered as a drawback. This happens because a graph growing from a node is usually determined by the configuration in this node and the predecessors of the node. But MRSC is too general as it assumes that rules work on whole graphs, not just paths from a root to a current node. So it is possible to write rules which prescribe different steps for the node c in the considered example. It means that we should put more restrictions on rules in order to perform more sophisticated optimizations.

Another drawback of MRSC is a premature loss of information. As graphs are seen as independent from a graph consumer point of view we cannot use more efficient algorithms that can make use of their interconnections. To give an example of such an algorithm let's consider a problem of finding the smallest program among the residual programs. A brute-force solution would be to residualize all completed graphs and then compute the sizes of the programs and find the minimum. A more clever solution would be to pick the smallest residual subprogram for each intermediate node while building the residual program. The algorithm enabling this optimization will be discussed in more detail later in this article.

Now we can see what representation would be more advantageous – let's replace a set of graphs with its description by merging the graphs into one huge graph, which we will call an *overtree* to underline that the graphs are still

essentially trees with back edges. It is convenient to combine edges representing a single step into a *hyperedge* with one source and several destinations (Fig. 4, hyperedges are drawn as bundles of edges going from one point). Then it is possible to extract a graph representing a program from an overtree by selecting one outcoming hyperedge for each node and then removing unreachable nodes. Note that it is convenient to consider terminal nodes (which represent constants, variables, etc.) as having outcoming hyperedges with zero destination nodes.

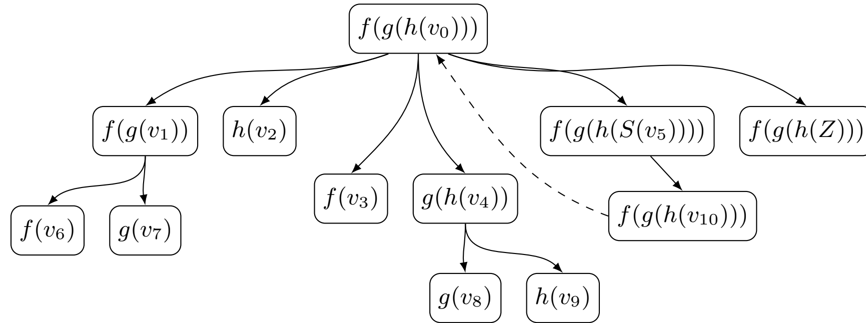

Fig. 4. Overtree representation

This representation, which we will call the overtree representation, doesn't change the graph building algorithm very much, it actually becomes much more similar to the traditional single-result supercompilation. This representation has been implemented in MRSC as an experiment. This experiment unveiled some shortcomings of the proposed representation: turned out there were a lot of equal but unshared subtrees. It is quite clear where they were coming from. Consider some complex program involving multiplication (implemented as a function on Peano numbers). There will be a lot of different ways of supercompiling this program, some of them will have the multiplication function fused with other operations, but some will prefer to generalize it out leading to multiple equal subgraphs corresponding to the multiplication and scattered over the overtree. Obviously we do not want to supercompile the same thing several times, so the next natural step is to get rid of duplicated configurations. We could have done it by introducing cross edges similarly to back edges, but there is a cleaner solution – let's shift from a de facto tree representation to a fully fledged graph (Fig. 5). That is if during supercompilation we encounter a configuration which we have already processed, we do not create a new node for it. Thus each configuration corresponds to no more than one node. This new representation can be called the *overgraph* representation. Note that configurations equal up to renaming should be identified.

Unlike the overtree representation this one seems to be a bit more groundbreaking. Special folding edges are not needed anymore as they can be represented as ordinary edges. However, we cannot safely use traditional binary

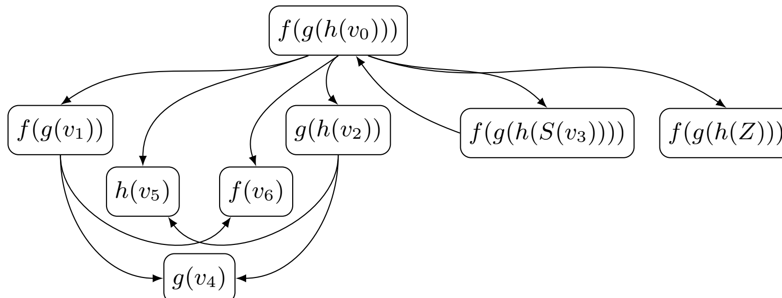

Fig. 5. Overgraph representation

whistles because possible steps cannot depend on the history of computation (and hence unary whistles can still be used). Why is it so? Because each node may have multiple immediate predecessors and thus multiple histories. Let us describe the overgraph representation more formally and consider how these issues can be addressed.

4 Overgraph Representation

In this section a multi-result supercompilation with overgraph representation is discussed. A configuration graph corresponds to (and actually can be represented by) a set of statements of the following form:

$$s \xrightarrow{l} (d_1, \dots, d_n)$$

where s is a source configuration and d_i are destination configurations. Configurations are in one-to-one correspondence with nodes. The whole statement corresponds to a hyperedge labeled with some information l. The exact meaning of the statements depends on the interpretation of the relation $\rightarrow$ which is different for different applications of supercompilation. For traditional supercompilation of functional programs it can be like this:

$$s \xrightarrow{f} (d_1, \dots, d_n) \overset{def}{\iff} s \gtrsim f(d_1, \dots, d_n)$$

Note the use of the improvement relation ($\gtrsim$) instead of a simple equality ($=$). This is due to the necessity of ensuring correctness of folding [12,13]. Informally $a \gtrsim b$ means that a is not only operationally equal to b but also more efficient than b (needs fewer reduction steps to be evaluated in any context). We won't elaborate on the topic of correctness since it is not very troublesome in the case of one-level supercompilation, but importance of the improvement relation consists in asymmetry it brings. When a hyperedge connects only two nodes we cannot generally consider it undirected which could be inferred if the relation were symmetric.

If we do not put any restrictions on the set thus representing a graph, then we get an overgraph. If we want to represent a singular configuration graph then we should state that each configuration can appear to the left of the $\to$ no more than once. Several singular graphs can be merged to form an overgraph by simple union.

The supercompilation with overgraphs is proposed to be performed in three steps:

1. Graph construction. In this step edges and nodes are only being added.
2. Graph truncation. In this step edges and nodes are only being removed.
3. Residualization. As it will be seen this step becomes a bit more nontrivial.

For construction and truncation we will use the rules of the form

$$\frac{\text{precondition}}{s \xrightarrow{l} (d_1, \ldots, d_n)} \qquad \frac{\text{precondition}}{\mathit{remove}(\text{node or hyperedge})}$$

to add a new hyperedge (together with necessary nodes) and to remove a hyperedge or a node (together with all incident hyperedges) correspondingly.

4.1 Graph Construction

This step seems to be rather straightforward: start with a graph containing only a root node (with the configuration being supercompiled) and then apply the rules until the saturation is reached, i.e. there is no applicable rule that can add a new node or edge. There are two problems: the rules formulation and the order of their application. The rules should ensure termination and provide enough diversity but not too much. The traditional way of ensuring termination is to use a whistle. But whistles have been used not only for the termination problem but also for picking the configurations that are to be generalized. Since we use the multi-result supercompilation the generalization can be decoupled from the whistle and thus a much simpler whistle can be used. For example, it is possible to simply limit the depth of nodes, although this solution does not scale very well.

Concerning the problem of diversity, there are usually a lot of possible generalizations of a configuration, and if we take all of them, even the overgraph representation won't help us fight combinatorial explosion. If we pick few of them, we risk missing some good opportunities. Obviously heuristics are unavoidable. Limiting the number of unbound variables appeared to be a good one for preventing overgeneralization. Good heuristics for multi-result supercompilation are yet to be found and researched.

A problem of rules application order may arise when there are rules with non-monotonic preconditions, i.e. a rule precondition can change its value from `true` to `false` when certain nodes or edges are *added* to the graph. For example consider the following rule:

$$\frac{\neg\exists m\colon m \in V \land \mathit{whistle}(n, m)}{n \to \mathit{drive}(n)}$$

where V is the set of nodes of the graph. It prescribes a drive step for a node n if there is no node m in the graph which makes a binary whistle blow. If there is another rule that adds such a node m then these two rules won't commute.

Since we agreed not to remove edges and nodes on the graph construction step, there won't be any problem if all the preconditions are monotonic. For example it is possible to use a whistle this way: let's allow driving if there *is* another node that does *not* make the binary whistle blow.

4.2 Graph Truncation

This step is dual to the previous. Its purpose is to reduce the number of nodes by removing useless ones. Useless nodes are those which are unreachable from the root or have no outcoming edges (remember that we have agreed to consider terminal nodes as having an outcoming edge without destination nodes). When a node is deleted, all hyperedges incident with it must be deleted as well, leading to new useless nodes. That's why this procedure should be repeated until there are no useless nodes left. It can be also described with the following rules:

$$\frac{\neg\exists l, d_1, \ldots, d_n : s \xrightarrow{l} (d_1, \ldots, d_n)}{remove(s)}$$

$$\frac{\neg\exists p : p \text{ is a path from the root to } s}{remove(s)}$$

In this step it is also possible to apply a whistle. On the one hand it may seem a bit too late: the graph has already been built by now, so we don't need to ensure termination on this step. Moreover, we can't return consumed CPU time by removing parts of the graph (we can return some memory though). But on the other hand experiments show that most time is being consumed later on the residualization step, so reducing the number of nodes and edges of the graph is a good thing to do. However it is still impossible to use a binary whistle in a traditional way for the same reason: the traditional usage may lead to noncommutativity of rules. At this stage to ensure commutativity preconditions should be monotonically decreasing, and actually it is possible to use rules from the previous step by negating their preconditions, i.e. "if the whistle doesn't blow, add a node" becomes "if the whistle blows, remove a node". The only advantage of using whistles in this step seems that now it is possible to look at the graph as a whole and extract some useful information to adjust the whistle.

Note also that although commutativity of rules makes a supercompiler a bit cleaner and more predictable, this property is not obligatory and can be discarded in practice.

5 Residualization

The residualization step is special because it transforms a set of graphs represented as one graph into an actual set of graphs (or programs which are a special case of graphs), and here the danger of explosion threatens again.

Firstly we should agree upon what graphs we will consider residual. In this section we will study the case of trees with back edges (but without cross edges), i.e. we won't identify equivalent subprograms in different contexts. This choice was initially determined by the structure of the language for which the overtree representation had been initially implemented (it will be dicussed in the next section). It may look a bit inconsistent: why not go further and represent programs as graphs (especially recalling that it is a traditional program representation)? We will return to this question later in Section 6.2.

$$
\begin{aligned}
&\mathcal{R}[\![n, h]\!] = \{n\} && \text{if } n \text{ is terminal}\\
&\mathcal{R}[\![n, h]\!] = \{\mathrm{Call}(n)\} && \text{if } n \in h\\
&\mathcal{R}[\![n, h]\!] = \Big\{\mathrm{Def}(n)[\![f(r_1, \ldots, r_k)]\!] \;\Big|\; n \xrightarrow{f} (d_1, \ldots, d_k), \\
&\qquad\qquad\qquad\qquad\qquad\qquad r_i \in \mathcal{R}[\![d_i, h \cup \{n\}]\!]\Big\} && \text{otherwise}
\end{aligned}
$$

Fig. 6. Naive residualization algorithm

Consider a naive residualization algorithm (Fig. 6). It takes a node and a history and returns a set of residual programs. It is usually applied to the root node and the empty history: $\mathcal{R}[\![root, \emptyset]\!]$. The algorithm recursively traverses the graph memorizing visited nodes in the history h. If it encounters a node that is already in the history, it creates a function call which is designated as $\mathrm{Call}(n)$. If a node is not in the history and has successors, a function definition should be created with the construction $\mathrm{Def}(n)[\![body]\!]$, so as it can be called with a $\mathrm{Call}(n)$ construction from within the *body*. The implementation of Call and Def depends on the output language, for example the Def construction can be implemented with letrec expressions:

$$\mathrm{Def}(n)[\![body]\!] = \textbf{letrec}\ n = body\ \textbf{in}\ n$$

Usually it is a bit more complex because of unbound variables. Note also that practical implementations of the algorithm should create a new function definition only if there is a corresponding function call in the body, we just create a function definition for each node for simplicity.

When applied, this residualization algorithm will visit each node the number of times equal to the number of computation paths from the root to it. So the problem reappeared: we need to compute (almost) the same thing many times.

5.1 Enhanced Residualization Algorithm

The solution to the problem is quite obvious – cache residual programs for intermediate nodes. It cannot be applied directly though because the naive residualization algorithm takes a history of computation besides a node. However,

residualization doesn't need full information contained in a history, i.e. the residual program for a node may be the same for different histories. So if we can do with less information, the algorithm will be feasible to memoize.

To do this we need analyze the structure of a computation history for a given node. Let's first give a couple of definitions.

Definition *A node m is a successor of a node n, $n \to^* m$, if there is a directed path from n to m (possibly with zero length). A set of successors will be denoted as $succs(n)$.*

Definition *A node m is a predecessor of a node n if n is a successor of m. A set of predecessors will be denoted as $preds(n)$.*

Definition *A node n dominates a node m, n dom m, if every path from the root to m contains n.*

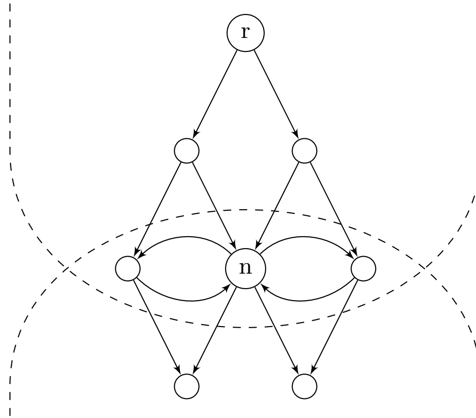

Fig. 7. Nodes whose presence in a history can influence residualization process are in the intersection of the set of all predecessors and the set of all successors

Given a node n, the nodes that are not its predecessors are not interesting as they won't be in a history. The nodes that are not successors are not interesting either because they don't influence the residualization process (they can be in a history but they won't be encountered again). Thus we care only about nodes which are successors and predecessors of n at the same time (i.e. those which are in the same strongly connected component with n, Fig. 7), so we just need to remove nodes that are not successors from history when calling the residualization function:

$$\mathcal{R}[\![n, h]\!] = \left\{ \mathrm{Def}(n)[\![f(r_1, \ldots, r_k)]\!] \;\middle|\; n \xrightarrow{f} (d_1, \ldots, d_k), \; r_i \in \mathcal{R}[\![d_i, (h \cup \{n\}) \cap succs(d_i)]\!] \right\}$$

This small modification to the last equation of the residualization algorithm is sufficient to make it ready for memoization. An interesting side-effect of the memoization is that we get residual programs for intermediate nodes for free. These programs can then be used to enable two-level supercompilation (although the topic of two-level supercompilation in context of overgraph representation is yet to be researched as few results have been obtained so far). Note though that the residual programs for intermediate nodes are likely to be defined in terms of other nodes, being used as if they were built-in functions.

Let's look at the structure of a history in more detail. It can contain only nodes from $S(n) = \mathit{preds}(n) \cap \mathit{succs}(n)$. The elements of this set fall into the following groups:

- Dominators of n except n. These elements are always in a history.
- Dominatees of n except n. If there is such a node in a history then the history must also contain the node n and this falls into the second equation of the algorithm.
- Neither dominators nor dominatees of n. These nodes are responsible for the diversity of possible histories.
- n itself.

We believe that the information about the history structure can be used somehow to develop more sophisticated residualization algorithms.

6 Implementation and Experimental Results

The described overgraph representation has been implemented as a part of MRSC. The implementation is experimental and not language-independent yet. The language being used is higher-order although it is meant to be used primarily as first-order.

$e ::= v$	variable
$\mid\ \lambda v.e$	λ-abstraction
$\mid\ e_1\ e_2$	application
$\mid\ \mathbf{fix}\ e$	fixed point
$\mid\ c\ e_1 \ldots e_n$	constructor
$\mid\ \mathbf{case}\ e_0\ \mathbf{of}\ \{c_1\ v_1 \ldots v_{k_1} \to e_1;\ \ldots\ \}$	case-expression

The explicit fixed point operator is meant to be the only source of nontermination. There are no let-expressions, so there is no possibility of expressing sharing. This was done for the sake of simplicity.

Configurations are just expressions. They may contain unbound variables which are named in some canonical way (e.g. numbered from left to right), so as to make expressions, equal up to a renaming, syntactically equal. Configurations are self-contained, i.e. we don't need some external set of function definitions to understand their meaning (this is necessary for higher-level supercompilation [4]).

The following rules are used for graph construction:

$$depth(n) = \min\{|p| \mid p \text{ is a path from the root to } n\}$$

$$\frac{depth(n) \leq MaxDepth \qquad (f,d) = drive(n)}{n \xrightarrow{f} d}$$

$$\frac{depth(n) \leq MaxDepth \qquad (f,g,h) \in rebuildings(n) \qquad |\mathrm{FV}(g)| \leq MaxFree}{n \xrightarrow{f} (g,h)}$$

The *drive* function performs a drive step and returns a pair (f, d) where d is a reduced expression, or a tuple of expressions if it is a case analysis step, and f is a function such that $f(d) = n$. The function $rebuildings(n)$ returns a set of triples (f_v, g, h) where f_v is a substitution function such that for all expressions x and y

$$f_v(x, y) = x[v := y]$$

and expressions g and h are such that

$$f_v(g, h) = g[v := h] = n$$

where v appears in g exactly once. In theory, the latter restriction can be lifted leading to the ability of introducing sharing (note that we don't need let-expressions as the configuration is being immediately split).

The constants *MaxDepth* and *MaxFree* limit the depth of nodes and the maximum number of free variables in configurations respectively.

The functions Call and Def for this language look like this:

$$\mathrm{Call}(n) = v_n\ \overrightarrow{x}$$
$$\mathrm{Def}(n)[\![b]\!] = \mathbf{fix}\ (\lambda v_n.\lambda \overrightarrow{x}.\ b)$$

where the variable v_n has a unique name corresponding to the configuration n and $\overrightarrow{x}$ is a vector of n's free variables.

6.1 Results

The graph representations and the residualization algorithms have been assessed on the following programs:

$$\begin{aligned}
\mathrm{add} &= \mathbf{fix}\ \lambda fxy.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to S\ (f\ x\ y);\ Z \to y;\} \\
\mathrm{mul} &= \mathbf{fix}\ \lambda fxy.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to \mathrm{add}\ y\ (f\ x\ y);\ Z \to Z;\} \\
\mathrm{fict} &= \mathbf{fix}\ \lambda fxy.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to f\ x\ (S\ y);\ Z \to Z;\} \\
\mathrm{idle} &= \mathbf{fix}\ \lambda fx.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to f\ (f\ x);\ Z \to Z;\} \\
\mathrm{evenBad} &= \mathbf{fix}\ \lambda fx.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to \mathbf{case}\ (f\ x)\ \mathbf{of}\ \{T \to F;\ F \to T;\}; \\
&\qquad\qquad Z \to T;\} \\
\mathrm{nrev} &= \mathbf{fix}\ \lambda fx.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to \mathrm{add}\ (f\ x)\ (S\ Z);\ Z \to Z;\}
\end{aligned}$$

At this stage of research some numerical characteristics of the described ideas were measured rather than their ability to solve problems. It can be seen that the overgraph representation leads to much fewer nodes even in comparison with the overtree representation which enables full sharing of initial subtrees (Fig. 8).

	overtree	overgraph
add	7	6
mul	379	77
fict	132	30
idle	5237	139
evenBad	27307	242
nrev	46320	277

Fig. 8. Comparison of graph representations: the number of nodes created by the supercompiler, $MaxDepth = 10$

At the same time the results of the caching residualization algorithm look much more modest (Fig. 9). Note that in this case the *MaxDepth* constant was chosen to be 6, so the residualization could terminate in a reasonable amount of time. The number of nodes left after truncation are shown to compare with the number of node visits each algorithm makes. Although the caching algorithm performs much better than the naive one, the growth of the node visits is too rapid.

	nodes after truncation	nodes visited		residuals
		naive	caching	
add	6	9	8	1
mul	19	81	53	4
fict	13	52	48	4
idle	33	2413	682	112
evenBad	76	33223	2751	229
nrev	30	4269	402	19

Fig. 9. Comparison of residualization algorithms, $MaxDepth = 6$

6.2 Application to Counter Systems

There was a hypothesis that the overgraph representation would be advantageous for other domains. It was checked on the domain of counter transition systems. This choice was quite obvious as a considerable work had been done to implement and evaluate a supercompiler for counter systems within MRSC [5], so we had to simply replace the core of this supercompiler and compare it with the original one.

Let's briefly describe the domain. Counter systems are similar to programs but much simpler. A system can be in some state represented by a tuple of integers. The system can nondeterministically move from one state to another according to certain rules. Configurations represent sets of states and are actually tuples of integers and wildcards, wildcard meaning "any number". So a configuration can be driven by applying rules, or generalized by replacing a number with the wildcard. The task is to find a minimal correctness proof for some protocol modelled by a counter system. For each protocol certain states are unsafe, so a protocol is correct if the corresponding counter system can't turn out in an unsafe state. A proof is actually a completed graph (i.e. each node has an outcoming edge) without unsafe configurations. The whistle used in MRSC for this task was a unary one, so it was used with the overgraph-based core without any problem.

The results of applying the overgraph representation with the residualization algorithm described above were disappointing. The supercompiler with overgraph core not only spent much more time than the original one, but also failed to find the minimal proofs. It is easy to see why this happened – our residualization algorithm was designed to find trees with back edges, not graphs, and thus equivalent proof parts were duplicated. On the other hand, the original implementation of supercompiler had the ability to introduce cross edges which made it almost as powerful as the overgraph supercompiler.

	original	overgraph truncation
Synapse	12	12
MSI	10	10
MOSI	36	34
MESI	46	18
MOESI	156	57
Illinois	58	19
Berkley	50	33
Firefly	18	15
Futurebus	476106	1715
Xerox	94	43
Java	109410	12165
ReaderWriter	2540	154
DataRace	21	12

Fig. 10. Number of nodes visited by the supercompilers for each protocol

So the residualization algorithm must have been replaced with some different algorithm. We have tried a depth-first search with pruning of too large graphs which is exactly what MRSC was doing. So the only important difference left was that our implementation built an overgraph explicitly. Actually there is an advantage of explicit overgraph building: an overgraph can be truncated and

thus we can avoid obviously useless branches. It was quite difficult to compare the implementations fairly because of multiple subtle differences affecting the results. The comparison of the number of nodes visited during the supercompilation by the original supercompiler and the supercompiler with overgraph truncation enabled is shown on Figure 10. As can be seen, truncation turned out to be quite advantageous on complex protocols.

This unsuccessful application of the presented earlier residualization algorithm to another domain doesn't mean that it is useless. It was just applied to the kind of tasks it hadn't been designed for, namely to the task of optimizing for size (in this case a proof size). It is very hard to divide a task of graph minimization into subtasks of subgraph minimization because these subgraphs may have intersections, i.e. the minimal size of the whole doesn't equal to the sum of the minimal sizes of the parts. However if we want the same node to be residualized differently in different contexts, our residualization algorithm may be used.

7 Conclusion

We have presented a more efficient representation of configuration graphs for multi-result supercompilation and shown that this representation enables a more efficient residualization algorithm.

The idea of representing the space of configurations as a graph rather than a tree is quite obvious. The process of supercompilation can be viewed as some kind of graph search similar to finding paths in mazes, and it is natural to use the graph structure instead of unrolling it into a tree.

The overgraph representation also gives rise to a parallel to the field of optimizing compilers which manifests itself in the similarity of a configuration graph and a control flow graph. It is not yet obvious if this parallel is fruitful.

One of the closest work to this one seems to be the work on equality saturation [14]. One of the most important difference is that we need to support folding and thus we work with some directed relations rather than simple equalities to ensure correctness.

We believe that the new approach leads to many directions of reasearch and further improvement.

The residualization algorithm is still a bottleneck. There are many possible solutions. It may be useful to do without residualization if the task doesn't actually consist in producing a program, e.g. there should be some algorithm for proving programs equality which works directly on configuration graphs rather than on sets of residual programs. Another way is to reduce the number of nodes using some heuristics, especially interesting are methods that make possible to tune how long the supercompilation will run.

As it has been seen, there should be different residualization algorithms for different tasks. In the case of optimizing for size its goal is to extract a minimal supgraph. Apparently there may be some residualization algorithm of this sort

which would take advantage of the overgraph representation but it hasn't been found yet. Besides, it may be advantageous to represent programs as graphs.

It also seems interesting to apply the new approach to different domains. We also plan to add support for higher-level supercompilation [7] which may benefit from sharing information among multiple lower-level supercompilation instances.

Acknowledgements

The author would like to express his gratitude to Sergei Romanenko, Andrei Klimov and Ilya Klyuchnikov for their comments, fruitful discussions and encouragement.

References

1. D. G. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Commun. ACM*, 16:591–603, October 1973.
2. K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 34(7):1–9, July 1999.
3. G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, Ottawa, Canada, 2008. MILEPOST project (http://www.milepost.eu).
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. A. V. Klimov, I. G. Klyuchnikov, and S. A. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation. Preprint 19, Keldysh Institute of Applied Mathematics, 2012.
6. 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.
7. I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.
8. I. G. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute of Applied Mathematics, 2011.
9. I. G. Klyuchnikov and S. A. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. volume 7162 of *Lecture Notes in Computer Science*, pages 210–226. Springer, 2012.
10. 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.
11. Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, pages 319–332. IEEE Computer Society, 2006.
12. D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theor. Comput. Sci.*, 167(1-2):193–233, 1996.

13. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.
14. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.
15. V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.