Nonlinear Configurations for Superlinear Speedup by Supercompilation

Robert Glück[1], Andrei Klimov[2⋆], and Antonina Nepeivoda[3⋆⋆]

[1] DIKU, Department of Computer Science, University of Copenhagen
[2] Keldysh Institute of Applied Mathematics, Russian Academy of Sciences
[3] Ailamazyan Program Systems Institute, Russian Academy of Sciences

Abstract. It is a widely held belief that supercompilation like partial evaluation is only capable of linear-time program speedups. The purpose of this paper is to dispel this myth. We show that supercompilation is capable of *superlinear* speedups and demonstrate this with several examples. We analyze the transformation and identify the *most-specific generalization* (msg) as the source of the speedup. Based on our analysis, we propose a conservative extension to supercompilation using *equality indices* that extends the range of msg-based superlinear speedups. Among other benefits, the increased accuracy improves the time complexity of the palindrome-suffix problem from $O(2^{n^2})$ to $O(n^2)$.

Keywords: program transformation, supercompilation, unification-based information propagation, most-specific generalization, asymptotic complexity.

1 Introduction

Jones *et al.* reported that partial evaluation can only achieve linear speedups [11, 12]. The question of whether supercompilation has the same limit remains contentious. It has been said that supercompilation is incapable of superlinear (s.l.) speedups (*e.g.*, [13]).

However, in this paper we report that supercompilation is capable of s.l. speedups and demonstrate this with several examples. We analyze the transformation and identify the *most-specific generalization* (msg) as the source of this optimization. Based on our analysis, we propose a straightforward extension to supercompilation using *equality indices* that extends the range of s.l. speedups. Among other benefits, the time complexity of the palindrome-suffix problem is improved from $O(2^{n^2})$ to $O(n^2)$ by our extension. Without our extension the msg-based speedup applies to few "interesting programs".

First, let us distinguish between two definitions of supercompilation:

$$\textit{supercompilation} = \textit{driving} + \textit{folding} + \textit{generalization} \tag{1}$$

and

$$\textit{supercompilation} = \textit{driving} + \textit{identical folding} \tag{2}$$

⋆ Supported by RFBR, research project No. 16-01-00813-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

⋆⋆ Partially supported by RFBR, research project No. 14-07-00133-a, and Russian Academy of Sciences, research project No. AAAA-A16-116021760039-0.

Examples of these have been described, SCP (1) [7, 17, 27, 28, 30] and SCP (2) [4, 25]. In brief, the former use various sophisticated whistle and generalization algorithms to terminate the transformation. In contrast, the latter only fold two configurations that are *identical modulo variable renaming* (α-identical folding) [25]. This means, SCP (2) terminate less often than do SCP (1). Nevertheless, non-trivial optimization can be achieved by SCP (2) due to the unification-based information propagation by driving, such as the optimization of a string matcher [4].

It was reported as proven that supercompilation, like partial evaluation, is not capable of s.l. speedups. However, that proof [23, Ch. 11] applies only to SCP (2) using identical folding without online generalization during supercompilation [23, Ch. 3], it does not apply to SCP (1).[4] We will show that s.l. speedups require folding and most-specific generalization.

We begin by giving an example that demonstrates that supercompilation is capable of s.l. speedups in Section 2. In Section 3, we briefly review driving and generalization of core supercompilation, the latter of which is the key to our main technical result. We characterize the limits of s.l. speedup by supercompilation in Section 4. In Section 5 we present our technique using equality indices which extend the range of superlinear speedups that a core supercompiler can achieve. Section 6 is the conclusion.

We assume that the reader is familiar with the basic notion of supercompilation (*e.g.*, [4, 7, 25]) and partial evaluation (*e.g.*, [12]). This paper follows the terminology [7] where more details and definitions can be found.

2 A Small Example of Superlinear Speedup by Generalization

To dispel the "myth" that supercompilation is only capable of linear speedups, let us compare supercompiler types, SCP(1) and SCP(2), using an example and analyze the transformations with and without generalization. This is perhaps the smallest example that demonstrates that an exponential speedup by supercompilation is possible.

The actual program transformations in this paper were performed by *Simple Supercompiler* (SPSC) [17][5], a clean implementation of a supercompiler with positive driving and generalization. The example programs are written in the object language of that system, a first-order functional language with normal-order reduction to weak head normal form [7]. All functions and constructors have a fixed arity. Pattern matching is only possible on the first argument of a function. For example, the function definition $f(S(x), y) = f(x, y)$ is admissible, but $f(x, S(y)) = f(x, y)$ and $f(S(x), S(y)) = f(x, y)$ are not.

Example 1. Consider a function f that returns its second argument unchanged if the first argument is the nullary constructor Z and calls itself recursively if it has the unary constructor S as the outermost constructor. The function always returns Z as a result. The computation of term $f(x, x)$ in the start function $p(x)$ takes exponential time $O(2^n)$ due to the nested double recursion in the definition of f (here, $n = |x|$ is the number

[4] Similar to partial evaluation, the result of a function call can be generalized by inserting let-expressions in the source program by hand (offline generalization) [2].

[5] SPSC home page http://spsc.appspot.com.

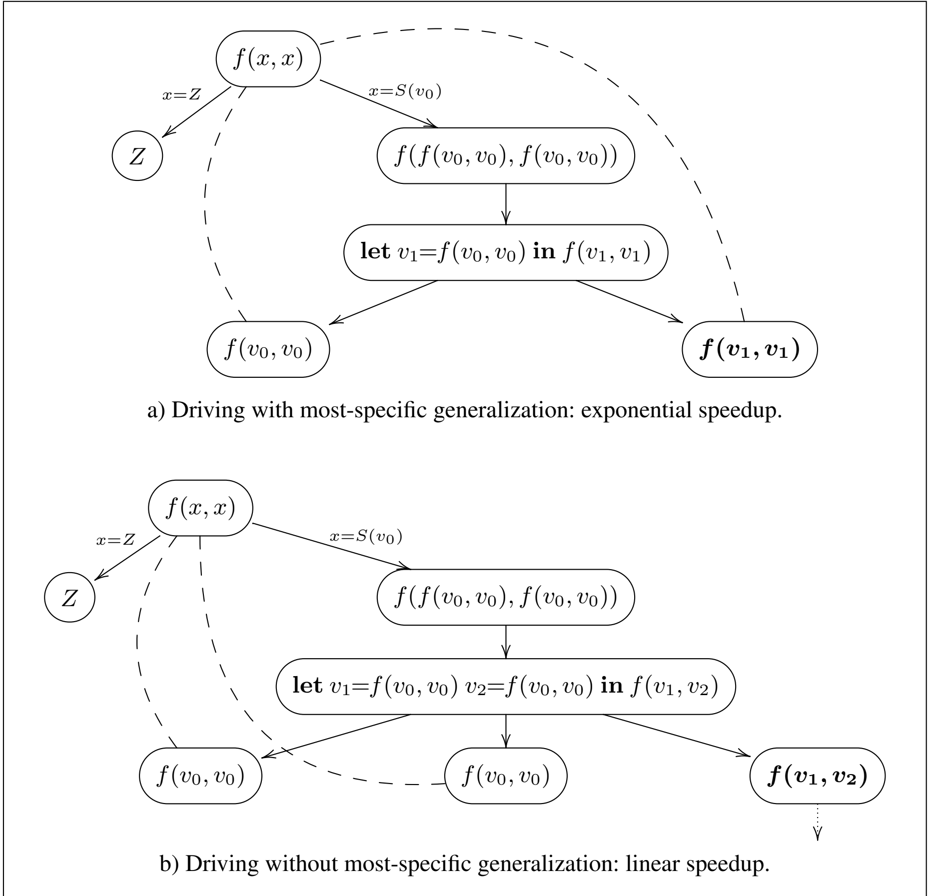

Fig. 1. The partial process trees of a supercompiler with and without generalization.

of S-constructors in the unary number x). The residual program f_1 produced by SPSC takes time $O(n)$ on the same input, that is an *exponential speedup* is achieved.[6]

Source program	Residual program
Start $p(x) = f(x, x)$;	Start $p1(x) = f_1(x)$;
$f(Z, y) = y$; $f(S(x), y) = f(f(x, x), f(x, x))$;	$f_1(Z) = Z$; $f_1(S(x)) = f_1(f_1(x))$;

To understand what causes this optimization, compare the two process trees produced by a supercompiler with and without generalization (Fig. 1a,b). In both cases, driving the initial term $s = f(x, x)$ branches into two subtrees. One for $x{=}Z$ and one for $x{=}S(v_0)$. Driving the left subtree terminates with term Z. In the right subtree each

[6] The speedup is independent of whether the programs are run in CBV, CBN or lazy semantics.

occurrence of x in the right-hand side of f is replaced by $S(v_0)$ and driving leads to the new term $t = f(f(v_0, v_0), f(v_0, v_0))$. The msg of the initial term s and the term t captures the repeated term $f(v_0, v_0)$:

$$\lfloor s, t \rfloor = (f(v_1, v_1), \{v_1 := x\}, \{v_1 := f(v_0, v_0)\}) , \quad (3)$$

and leads to the creation of the generalized node in the process tree (a):

$$\textbf{let } v_1{=}f(v_0, v_0) \textbf{ in } f(v_1, v_1) . \quad (4)$$

Both of the subterms in this node are instances of the initial term s and can be folded back to it. Hence, the process tree is closed and driving terminates. The linear-time residual program f_1 in Example 1 is obtained from process tree (a). Each function takes as many arguments as there are configuration variables in the corresponding configuration. Now only one variable remains, and the residual function becomes unary. It is the second rule of the msg (defined in Sect. 3) that unifies equal subterms and introduces term sharing in the process tree. We call this mechanism *msg-sharing*.

This optimization cannot be achieved by an SCP (2) supercompiler using identical folding without msg. In the same situation, as shown in process tree (b), such a supercompiler loses the connection between the two arguments in the body of the let-expression: $f(v_1, v_2)$. As a result, the original function definition is rebuilt in the residual program. In fact, without inserting let-expressions in one way or another, a supercompiler with identical folding does not terminate because the configurations continue to grow due to unfolding function calls. Like partial evaluation, s.l. speedup cannot be achieved by SCP (2). That supercompilation with msg can change the asymptotic time complexity was demonstrated above.

The speedup results from core positive supercompilation with msg [7], with no additional transformation techniques. The speedups are achieved both in call-by-name and call-by-value performance.

To illustrate the optimization, we compare side-by-side the runs of the source function and the residual function with unary input $SSSZ$ (abbreviated by 3). Fig. 2 shows the relevant normal-order reduction steps and omits some of the intermediate steps. The reader will notice that one call $f(3, 3)$ in the source-program run (left side) corresponds to one call $f_1(3)$ in the residual-program run (right side), two calls $f(2, 2)$ to one call $f_1(2)$, and four calls of $f(1, 1)$ to one call $f_1(1)$. In general, 2^m calls $f(n-m, n-m)$ correspond to one call $f_1(n-m)$ where n is the number represented by the unary input and $0 \leq m < n$. This shows that s.l. speedup relates to a variable-sized input, where the improvement increases superlinearly as the size of the input grows.

3 The Core Supercompiler

In this section we briefly review supercompilation with generalization. Since supercompilation is formally defined in the literature (*e.g.*, [7]) we will skip the full definition here and only review the algorithm of msg which introduces the sharing of terms.

A supercompiler takes an initial term and a program, and constructs a possibly infinite process tree. If the process tree is finite, a new term and a residual program are generated. The two main components of a supercompiler for developing the process tree are driving and generalization.

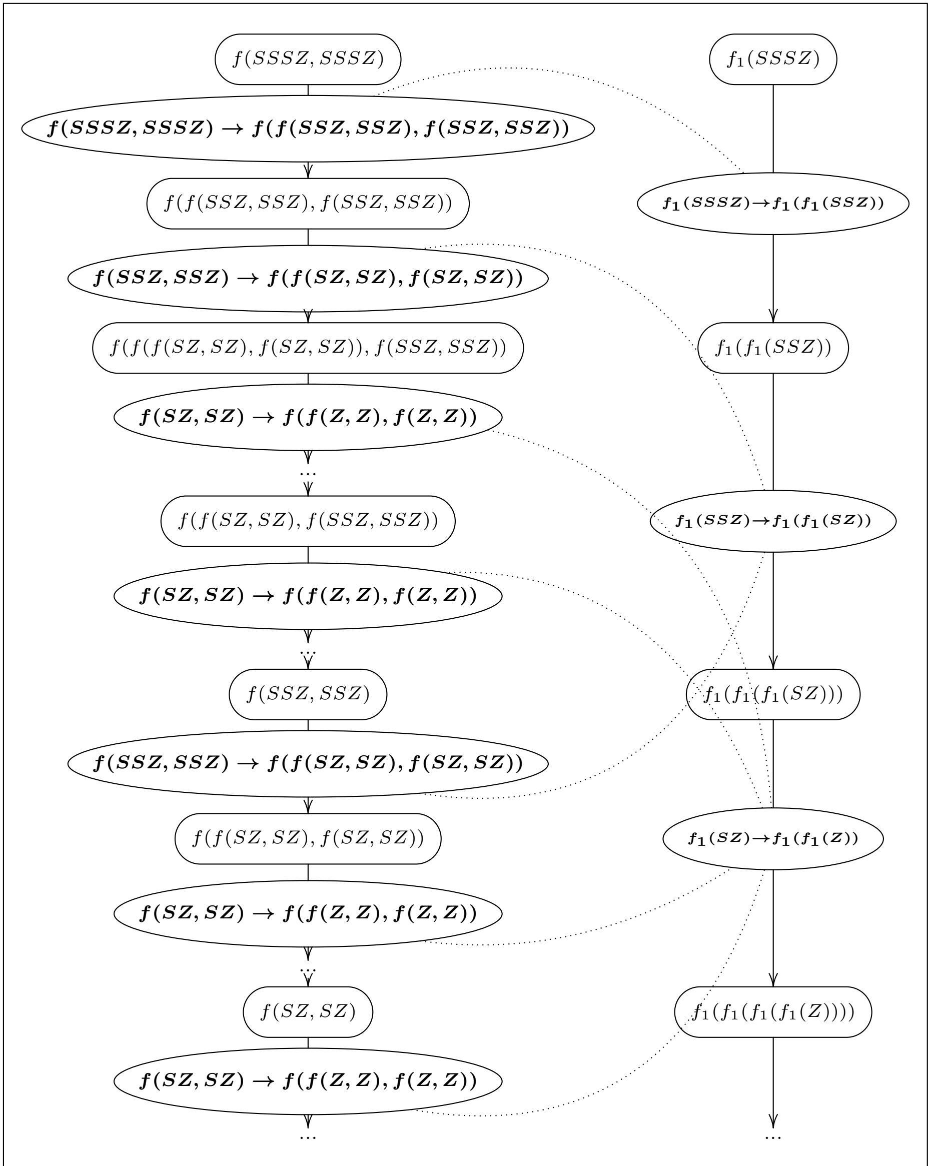

Fig. 2. Exponential- and linear-time runs of the source and residual functions of Example 1.

Driving makes use of *unification-based information propagation* to construct the process tree during supercompilation. Common to all driving methods regardless of their degree of information propagation [4] is the use of *configuration variables* and *nonlinear configurations*. These features distinguish driving from the constant-based information propagation of partial evaluation. It is essential that configuration variables may

occur repeatedly in a configuration. This is key to expressing term equality and sharing results of a subcomputation as demonstrated in Example 1.

Generalization To ensure that a finite process tree is constructed from which a residual program can be generated, supercompilation uses the most specific generalization. The result of the msg is used to decide whether to continue driving, fold to an existing configuration, or to generalize two configurations to a new one. There are various strategies to choose which configurations the msg is applied to and how its result is used including the use of transient configurations, upward and downward generalization. The particular strategy is irrelevant to this paper except that different strategies present more or less sharing opportunities to the msg.

We review the definition of msg used in positive supercompilation (taken from [24]). The msg of two terms, $\lfloor s, t \rfloor$, is computed by exhaustively applying the following rewrite rules to the triple $(x, \{x := s\}, \{x := t\})$. The result is the msg $(t_g, \theta_1, \theta_2)$ including a generalized term t_g and two substitutions θ_1 and θ_2 such that $t_g\theta_1 = s$ and $t_g\theta_2 = t$. Any two terms s and t have an msg which is unique up to renaming.

$$\begin{pmatrix} t_g \\ \{x := \sigma(s_1,\ldots,s_n)\} \cup \theta_1 \\ \{x := \sigma(t_1,\ldots,t_n)\} \cup \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g\{x := \sigma(y_1,\ldots,y_n)\} \\ \{y_1 := s_1,\ldots,y_n := s_n\} \cup \theta_1 \\ \{y_1 := t_1,\ldots,y_n := t_n\} \cup \theta_2 \end{pmatrix}$$

$$\begin{pmatrix} t_g \\ \{x := s, y := s\} \cup \theta_1 \\ \{x := t, y := t\} \cup \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g\{x := y\} \\ \{y := s\} \cup \theta_1 \\ \{y := t\} \cup \theta_2 \end{pmatrix}$$

Take for example the msg of the two configurations in Eq. 3 of Example 1. It identified the term-equality pattern in terms and led to the sharing by a let-expression.

An important property is that msg-based generalization is *semantics preserving*. The branches merged in the process tree are identical. No computation is deleted, and only the result of a computation is shared by a let-expression. The term equality expressed by a let-expression, *e.g.* **let** $v_1{=}f(v_0, v_0)$ **in** $f(v_1, v_1)$.

The sharing that supercompilation introduces may look like common subexpression elimination, but it is a *dynamic property* that occurs only during driving as the following example illustrates.

Example 2. Consider a variant of Example 1 where the right-hand side of function g contains the permuted terms $f(x, y)$ and $f(y, x)$. As before, given the initial configuration $f(x, x)$, the supercompiler achieves an exponential speedup. The term equality is discovered by the msg because the configuration variables are propagated during driving. Common subexpression elimination will not discover this equality.

Source program	Residual program
Start $p(x) = f(x, x)$;	Start $p1(x) = f_1(x)$;
$f(Z, y) = y$; $f(S(x), y) = g(y, x)$; $g(Z, y) = y$; $g(S(x), y) = f(f(x, y), f(y, x))$;	$f_1(Z) = Z$; $f_1(S(x)) = f_1(f_1(x))$;

4 Limitations of Sharing by Supercompilation

We identified the msg as the mechanism that introduces sharing of equal terms. We demonstrated that s.l. speedups are possible by supercompilation contradicting a common belief, but found few "interesting problems" to which this applies. For example, why does supercompilation not improve the well-known naive, exponential-time Fibonacci function? Some of the limitations that we identified are due to the strategy of developing process trees, *e.g.* identical configurations are not shared across subtrees. Other limitations are due to the unselective application of the msg to configurations, which leads to a loss of term equalities in the process tree. In this section, we examine two important limitations of core supercompilation. For the latter, we propose a more accurate technique of equality indices in the following section.

Limitation 1: No sharing across subtrees The speedups described so far are due to the sharing of computations. Core supercompilation does not identify all equal subterms in a single term (unlike the general form of jungle driving [21]). The msg identifies equal subterms when comparing *two configurations*, each of which contains equal subterms in the *same positions*. Thus, after a configuration is decomposed into subterms, *e.g.* by a let-expression, equal subterms are no longer identifiable as equal by the msg. The msg works *locally* on two given configurations in the process tree. Therefore, well-known examples with obvious opportunities for sharing cannot be improved by msg-sharing. They require a different supercompilation strategy for building process trees. This includes the naive Fibonacci function which requires the sharing of equal terms across subtrees. Let us illustrate this limitation.

Example 3. Consider the naive Fibonacci function:

$$\begin{aligned}
&f(Z) = S(Z);\\
&f(S(x)) = \mathit{ff}(x);\\
&\mathit{ff}(Z) = S(Z);\\
&\mathit{ff}(S(x)) = add(f(x), f(S(x))).
\end{aligned}$$

There are several repeated terms in the process tree of the naive Fibonacci function (Fig. 3), but they occur in separate subtrees. For example, the terms $\mathit{ff}(v_2)$ as well as $\mathit{ff}(v_3)$. The msg alone cannot find equal nodes across different subtrees. To share such terms requires another strategy of building process trees in the supercompiler. Clearly, msg-sharing cannot speed up the naive Fibonacci function, which is a limitation of the supercompilation strategy of building process trees.

Limitation 2: Unselective generalization Two configurations with the same pattern of equal subterms are more likely to appear when a generalization in a branch of the process tree with certain contractions of the variables is considered than when a generalization with the initial configuration is done. But in that case, the general configuration which is a parent of the more specific configuration may be generalized with this configuration resulting in the loss of the subterm equalities. This limitation of the strategy of applying msg-sharing can be avoided by a more accurate generalization as we will show in the next section. Let us illustrate this problem.

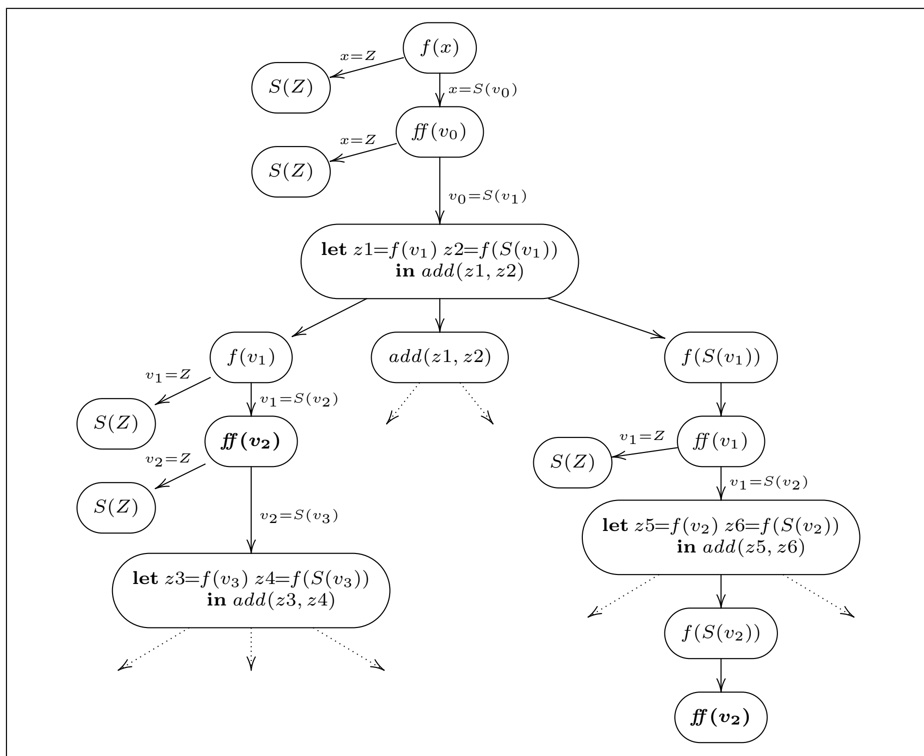

Fig. 3. A fragment of the process trees of the naive Fibonacci function.

Example 4. We replace the initial configuration of Example 1 by $p(x,y) = f(x,y)$. Given the process tree for $p(x,y)$, the configurations $f(x,y)$ and $f(f(x,x), f(x,x))$ are generalized to $f(x,y)$. After such a generalization, no speedup can be achieved by sharing. The residual program replicates the source program and no speedup is achieved. However, the second argument of the initial configuration is significant only for the first evaluation of $f(x,y)$. In all following evaluation steps, the initial value of y does not occur, and all calls of f have equal first and second arguments. The lesson learned from this example is that term equalities should be preserved in configurations. We shall use this observation as the basis for our extension.

Source program	Residual program
Start $p(x,y) = f(x,y)$;	Start $p(x,y) = f_1(x,y)$;
$f(Z,y) = y$; $f(S(x),y) = f(f(x,x), f(x,x))$;	$f_1(Z,y) = y$; $f_1(S(x),y) = f_1(f_1(x,x), f_1(x,x))$;

5 Increasing the Accuracy of Sharing: Equality Indices

Based on our analysis above, namely the generalization of term equalities in Example 4, we propose a new, conservative extension to supercompilation that makes the *applica-*

tion of the msg more accurate. We propose *equality indices* to reduces the loss of term equalities. Our goal is not to change the definition of the msg or the driving strategy, but to exploit the msg such that more term equalities can be preserved. We want residual programs to contain as many shared function calls as possible. A technique that assists in this task must distinguish function calls that have different term-equality patterns. Also, the technique must be applicable after each driving step during supercompilation because term equalities in configurations are a dynamic property (*cf.* Example 2).

5.1 How it Works

We examined several examples that show there are more chances for a deep optimization by preserving the term equalities uncovered dynamically during driving. It is more likely that this can be exploited by supercompiling a call with term equalities, *e.g.* $f(t,t)$, than a call without such equalities, *e.g.* $f(s,t)$ where $s \neq t$. We have seen in Example 4 that the loss of these equalities during msg limits the overall optimization by the supercompiler. Example 1 showed that in the residual program the arity of $f(t,t)$ is *reduced* to $f_1(t)$ provided a process graph can be built preserving this equality.

Our goal is to increase the accuracy of sharing by distinguishing calls that contain different patterns of syntactic term equalities. Our approach has two components. The annotation of all calls with equality indices, and the synchronization of annotated terms after driving and before calculating the msg.

First, let us annotate all function names in a configuration with equality indices. For example, the two calls from above are annotated as $f_{[1]}(t,t)$ and $f_{[2]}(s,t)$. Equality indices are textual representations of equality patterns, which can be seen with function calls represented by directed acyclic graphs (dag):

The msg will treat function names with different equality indices as different functions, thereby avoiding the loss of subterm equalities. The form and calculation of the indices, also for the nested case, will be treated in more detail below. For now let it suffice to say that they textually represent different sharing patterns (subterm equalities in calls).

The second issue that we need to approach is that driving can obscure term equalities. For example, driving the following nested call in Example 4 forces the unfolding of the first argument (underlined) and a substitution on the second argument (underlined). This yields in one step a configuration where the term equality disappears (the pattern matching on the first argument in a function definition forces the first argument to be unfolded, but not the second argument).

$$\begin{array}{rcl} f(\underline{f(x,x)}, f(x,x)) & \equiv & f(t,t) \\ \downarrow \text{driving with } x = S(v_1) & & \\ f(\underline{f(f(v_1,v_1), f(v_1,v_1))}, f(\underline{S(v_1)}, \underline{S(v_1)})) & \equiv & f(t',t''), t' \neq t'' \end{array} \quad (5)$$

Even though both arguments are identical before driving, this is obscured by unfolding the first one and substituting $S(v_1)$ into the second one. In fact, both arguments should be driven at the same time (in lockstep) to preserve the synchronization!

The new configuration should be as follows.

$$f(\underline{f(f(v_1,v_1),f(v_1,v_1))},\underline{f(f(v_1,v_1),f(v_1,v_1))}) \equiv f(t',t') \tag{6}$$

There are various ways to drive two (or more) terms in lockstep. Our solution is simpler and does not require an extension of the driving strategy. We restore ("synchronize") the argument equality after the driving step using the equality indices. This simple syntactic technique will be made more precise below, also for multiple arguments.

For our extension we need two small algorithms:

1. Annotate every call in a configuration with an equality index, *e.g.* $f(t,t) \stackrel{ind}{\to} f_{[1]}(t,t)$.
2. Synchronize all arguments in a driven configuration using the equality indices, *e.g.*

$$f_{[1]}(f_{[1]}(t,t),S(t)) \stackrel{sync}{\to} f_{[1]}(f_{[1]}(t,t),f_{[1]}(t,t)).$$

The two operations will be performed after driving and before applying the msg:

$$\text{driven configuration} \to \text{(1) index} \to \text{(2) synchronize} \to \text{msg} \to \text{drive} \tag{7}$$

A node in the process tree will from then on contain a term annotated with equality indices, t_{Ind}. This amplifies the power of the msg-based sharing mechanism that is present in core supercompilation, and does not require a fundamental change of supercompilation. It is not difficult to add this refinement to an existing system.

5.2 The Equality-Index Algorithm

We now present the technique in more detail. First, for each function call we find the arguments which are equal, and identify them by marking each function name with an equality index. Driving itself ignores the indexes, whereas the msg treats function names with different indices differently. The equality indexes are used after each driving step to restore argument equalities. The two algorithms, indexing and synchronization, are applied after each driving step and before the whistle and msg algorithms are applied.

An *equality index* lists for each argument in a function call the smallest number of the argument to which it is syntactically equal, in particular its own number if all arguments to its left are different. For brevity, the equality index for the first argument is omitted as it is always equal to 1.

For example, a call $f(x,x,z)$ is marked with the equality index $[1,3]$, which means that the second argument x is equal to the first argument x, and the third argument z is not equal to an argument to its left. Not all index combinations are possible, *e.g.* no call can be marked by $[1,2]$. A call $f(t_1,\ldots,t_n)$ in which all arguments differ has the index $[2,\ldots,n]$ and a call $f(t,\ldots,t)$ with identical arguments has the index $[1,\ldots,1]$.

Given a node in the process tree of a program and the driven configuration t in the node, we apply the following two algorithms to update the configuration.

First, we construct an annotated version of the configuration, which differs from the plain configuration only by function call indexes.

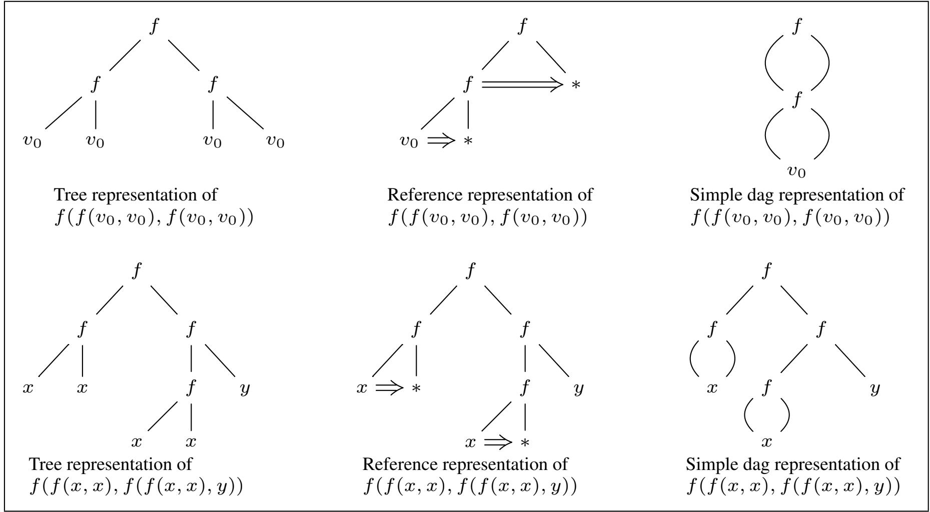

Fig. 4. Tree representations of terms vs. simple dag representations. A double arrow in the reference representation indicates the equality of arguments in a function call.

Algorithm 1 (Assigning equality indexes) *1. Given a configuration t, choose the innermost, leftmost function call in t that is not annotated. Let that call be $f(t_1, \ldots, t_n)$.*

(a) Assign auxiliary values $i = 2$, $j = 1$, $\Gamma = []$ (the empty list).

(b) If $t_i = t_j$ (syntactically), append $[j]$ to Γ, increase i, and set $j = 1$. Otherwise, increase j and repeat step (b). When j reaches i, $t_i = t_j$ is always true. So the number of steps in (b) is finite.

(c) When i reaches $n + 1$, rename this call of f in t to f_Γ. Then mark it as an annotated call and proceed with step (1).

Γ is *the equality index*. The length of the list is equal to the arity of the indexed function minus 1. This index shows which arguments of the call repeat each other. The equality index makes function calls with different argument equalities differ from each other, which preserves the equalities during the msg.

We illustrate how the Algorithm 1 works on term $f(f(v_0, v_0), f(v_0, v_0))$. The innermost, leftmost function call is $f(v_0, v_0)$, which is the first argument of the outer call. Step (a) assigns $i = 2$, $j = 1$, $\Gamma = []$. Step (b) checks whether $t_1 = t_2$ (the two arguments of the call f). Because $v_0 = v_0$, we get $\Gamma = [1]$ and change i to 3. Step (c) applies and the call is renamed to $f_{[1]}(v_0, v_0)$. The term becomes $f(f_{[1]}(v_0, v_0), f(v_0, v_0))$. The three steps are repeated with the second call $f(v_0, v_0)$ and the next term is $f(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0))$. Finally, after repeating the steps with the outermost call, the final term is $f_{[1]}(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0))$. The indices in this term identify for each argument the leftmost identical argument (Fig. 5, left tree).

All functions have a fixed arity, so for each function the set of possible equality indices is finite. The annotation of a binary function can lead to at most two different functions. A ternary function has at most five different equality indices and a function

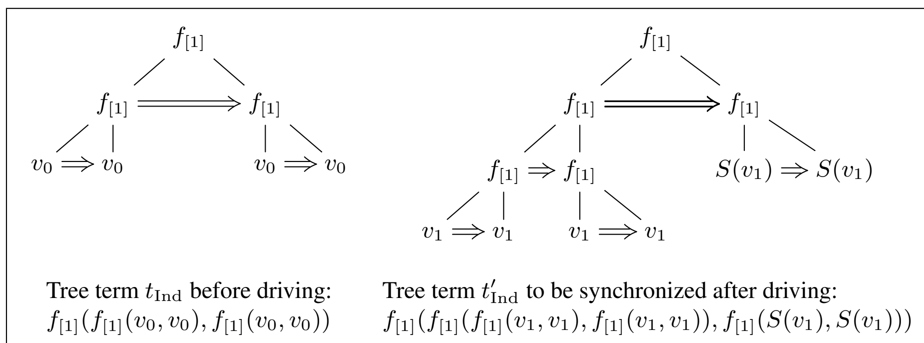

Fig. 5. Equality indexes identify the leftmost identical argument in a function call before driving, and are used to restore the lost argument equalities after driving (bold double arrow, right tree).

with four arguments has at most 15 different equality indices (and it is rare that all 15 arrangements appear during the supercompilation of the same program). Usually, the same function has only a few different equality indices.

The second algorithm restores obscured argument equalities using equality indexes.

Algorithm 2 (Synchronization) *1. Given an indexed term t_{Ind}, choose the innermost, leftmost unsynchronized function call. Let that call be $f_{[k_2,\ldots,k_n]}(t_1, \ldots, t_n)$ and set $j = 2$.*

2. *If $k_j = 1$, replace the k_j-th argument by the first argument. Otherwise, do nothing. Then, in both cases, increase j by 1.*
3. *When $j = n+1$, proceed with step (1) because the synchronization of function call $f_{[k_2,\ldots,k_n]}(t_1, \ldots, t_n)$ is finished. Otherwise, proceed with step (2).*

For example, given the term $f_{[1]}(f_{[1]}(v_0, v_0), S(v_1))$, the innermost unsynchronized call is $f_{[1]}(v_0, v_0)$. Since the equality index contains 1 at the first position, we perform the replacement and obtain the unchanged call $f_{[1]}(v_0, v_0)$. Next, the innermost unsynchronized call is the outermost call $f_{[1]}(\ldots)$. Again, we replace the second argument by the first one. The synchronized term becomes $f_{[1]}(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0))$.

Given calls $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$. If $\forall i, j \,.\, t_i = t_j \Leftrightarrow s_i = s_j$ then the equality indices for $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$ are identical. Hence, if the msg treats function names with different equality indices as different functions, it will never generalize two equal arguments of a call to two unequal terms.

Correctness Why does the synchronization not change semantics of the computation? The answer is that an argument can be copied in the call iff on some step of the computation it is syntactically identical to the other argument. If the call whose argument is copied retains its equality index, and the arguments of the call lose their syntactic equality, it means that the call itself was not driven itself — only its first argument. The source language guarantees that the driven argument is always the first argument — and the first argument is never changed by the synchronization. So, when the term is synchronized, its arguments are replaced only by semantically equal arguments that

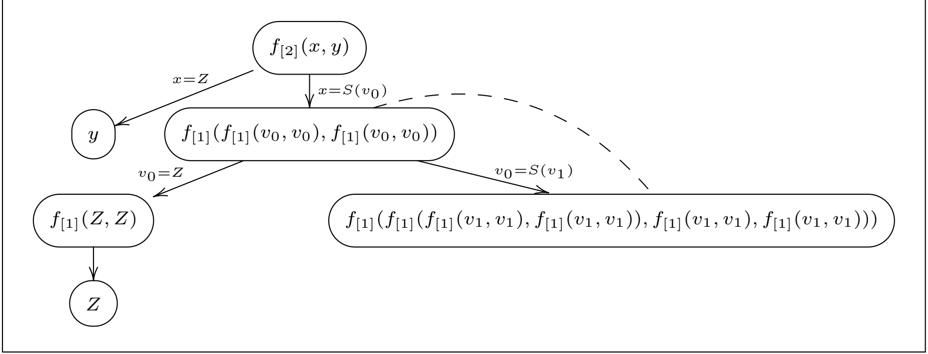

Fig. 6. Unfolding the process tree with equality indices of Example 4.

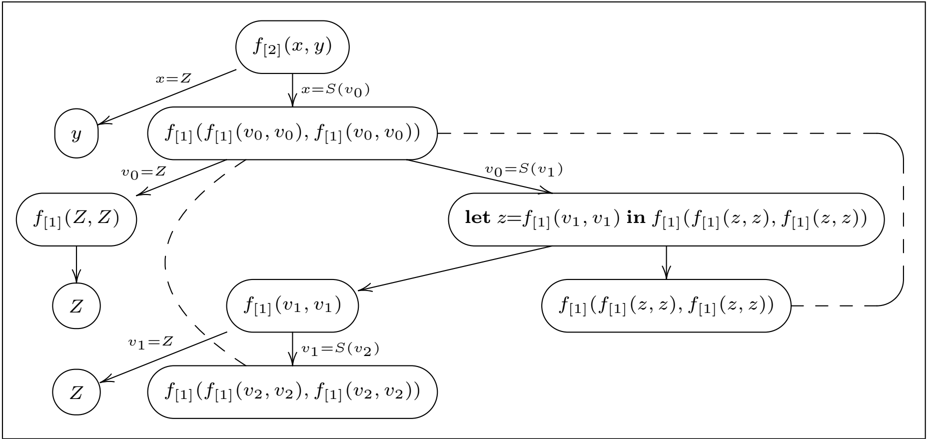

Fig. 7. The closed process tree with equality indices of Example 4.

came through more driving steps. Thus, the termination property of the program is preserved.[7]

Example 5. After introduction of the equality indexes, the process tree of Example 4 is constructed as follows. The first call $f(x, y)$ is annotated by equality index $[2]$ because its two arguments are syntactically different.

Under the assumption that $x = S(v_0)$, driving yields the next configuration

$$f(f(v_0, v_0), f(v_0, v_0)) \,. \tag{8}$$

First, the two innermost calls $f(v_0, v_0)$ are annotated by equality index $[1]$ (which means that the second argument is equal to the first). Then the outermost function call

[7] The source language plays a key role in this reasoning. If the language admits patterns with static constructors also in other argument positions, Algorithm 2 must be changed. We must not only copy the first argument, but any argument in which the actual driving step was done.

is also annotated by equality index [1] (its two arguments are equal). We obtain the indexed configuration (Fig. 5, left tree)

$$f_{[1]}(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0))\,. \tag{9}$$

Although driving this configuration unfolds the call of f in the first argument of the outermost call and substitutes into the second argument due to assumption $v_0 = S(v_1)$,

$$f_{[1]}(f(f(v_1, v_1), f(v_1, v_1)), f_{[1]}(S(v_1), S(v_1)))\,, \tag{10}$$

by which the two arguments become different, the configuration is synchronized to

$$f_{[1]}(f_{[1]}(f_{[1]}(v_1, v_1), f(v_1, v_1)), f_{[1]}(f_{[1]}(v_1, v_1), f(v_1, v_1))) \tag{11}$$

because the equality index [1] of the outermost call of f is unchanged by the driving step (Fig. 5, right tree) and the synchronization algorithm restores the lost syntactic equality of the two arguments (Fig. 5, bold double arrow, right tree). The assumption $v_0 = S(v_1)$, which was used for the substitution into the second argument, remains in the process tree (Fig. 6). Thus, no important driving information is lost by synchronization.

After the last step of driving, the msg of the two configurations (9) and (11),

$$\begin{aligned}&\lfloor f_{[1]}(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0)),\\ &\ f_{[1]}(f_{[1]}(f_{[1]}(v_1, v_1), f_{[1]}(v_1, v_1)), f_{[1]}(f_{[1]}(v_1, v_1), f_{[1]}(v_1, v_1)))\rfloor\\ &\ = (f_{[1]}(f_{[1]}(z, z), f_{[1]}(z, z)), \theta_1, \theta_2)\,,\end{aligned} \tag{12}$$

leads to the creation of the following generalization node in the process tree (Fig. 7):

$$\boxed{\textbf{let } z{=}f_{[1]}(v_1, v_1) \textbf{ in } f_{[1]}(f_{[1]}(z, z), f_{[1]}(z, z))}\,. \tag{13}$$

Finally, after driving $f_{[1]}(v_1, v_1)$ in (13), we obtain the closed process tree in Fig. 7. A linear-time residual program is generated from this tree (shown below). An exponential speedup is achieved by *supercompilation with equality indices*, which was not possible in Example 4 without the generalization precision induced by the equality indices.

Source program	Residual program
Start $p(x, y) = f(x, y)$;	Start $p(x, y) = f_2(x, y)$;
$f(Z, y) = y$; $f(S(x), y) = f(f(x, x), f(x, x))$;	$f_2(Z, y) = y$; $f_2(S(x), y) = \mathit{ff}(x)$; $\mathit{ff}(Z) = Z$; $\mathit{ff}(S(x)) = \mathit{ff}(f_1(x))$; $f_1(Z) = Z$; $f_1(S(x)) = \mathit{ff}(x)$;

In the residual program, the initial function f_2 has the general form, and functions ff and f_1 are generated as a specialization of f having two equal arguments.

We conclude that the equality indices algorithm makes more calls with identical arguments appear in the process tree. This method increases the precision of generalization without changing the driving and msg algorithms of the core supercompiler, but it is not powerful enough to handle old problems such as optimization of the naive Fibonacci function (Sect. 4). Equality indices do not give references to equal terms other than those local in the arguments of a function call, but they are an effective mechanism to preserve important argument equalities in a core supercompiler.

5.3 More Examples of the Equality-Index Algorithm

By using the equality indexes algorithm, we can achieve a superlinear speedup as per the following example.

Example 6. The palindrome-suffix program returns a suffix of either x or y that is a palindrome. If $x = A(A(A(B(Z))))$ and $y = A(B(A(B(Z))))$ then the program returns $B(A(B(Z)))$. The program has run time $O(2^{n^2})$ where $n = |x| + |y|$, that is the number of A- and B-constructors in x and y. Supercompilation without equality indices does not build a superlinear speedup of the program.

Source program	
Start: $p(x, y) = f(x, y)$;	$pal(A(z)) = and(palA(z, Z()), pal(z))$;
$f(x, y) = g(pal(x), pal(y), x, y)$;	$pal(B(z)) = and(palB(z, Z()), pal(z))$;
$g(T, z, x, y) = gt(z, x, y)$;	$palA(A(x), y) = palA(x, A(Z))$;
$gt(T, x, y) = x$;	$palA(B(x), y) = palA(x, B(Z))$;
$gt(F, x, y) = fmin1(y, y)$;	$palA(Z(), y) = ifA(y)$;
$gf(T, x, y) = y$;	$palB(A(x), y) = palB(x, A(Z))$;
$gf(F, x, y) = f(fmin1(x, x), fmin1(y, y))$;	$palB(B(x), y) = palB(x, B(Z))$;
$fmin1(A(x), y) = fmin2(y, x)$;	$palB(Z(), y) = ifB(y)$;
$fmin1(B(x), y) = fmin2(y, x)$;	$ifA(A(x)) = T$;
$fmin2(A(y), x) = f(x, y)$;	$ifA(B(x)) = F$;
$fmin2(B(y), x) = f(x, y)$;	$ifA(Z()) = F$;
$and(T, x) = x$;	$ifB(B(x)) = T$;
$and(F, x) = F$;	$ifB(A(x)) = F$;
	$ifB(Z()) = F$;

Function $fmin1$ that removes the first letter from the list always has equal arguments. However, equality is lost because the function call $f(x, x)$ after execution of $fmin1$ is generalized with $f(x, y)$ which has unequal arguments.

Consider the fragment of the process tree in Fig. 8. The call of $f_{[1]}$ with two equal arguments is generalized only with another call of $f_{[1]}$ with two equal arguments. Thus, the functions are transformed to functions with fewer arguments, and an unary version of f is generated. The residual program with the residual version of $fmin2$ calling the unary version of f, and the unary version of f calling the binary version of g has run time $O(n^2)$ where $n = |x|$.

Example 7. Now consider a negative example where the equality indexes do not help. Let us drive the naive Fibonacci function (Example 3) from an initial configuration $f(x_0)$. When the common order of driving is used by choosing the leftmost redex, repeated function calls are never observed in any configuration. To reveal the repeated calls inherent in the Fibonacci function the following strategy of supercompilation should be used.[8] After each contraction of a configuration variable and making a step of the redex, consider all function calls in the current configuration and make all steps that

[8] To the best of the authors' knowledge, it has not been implemented in any supercompiler.

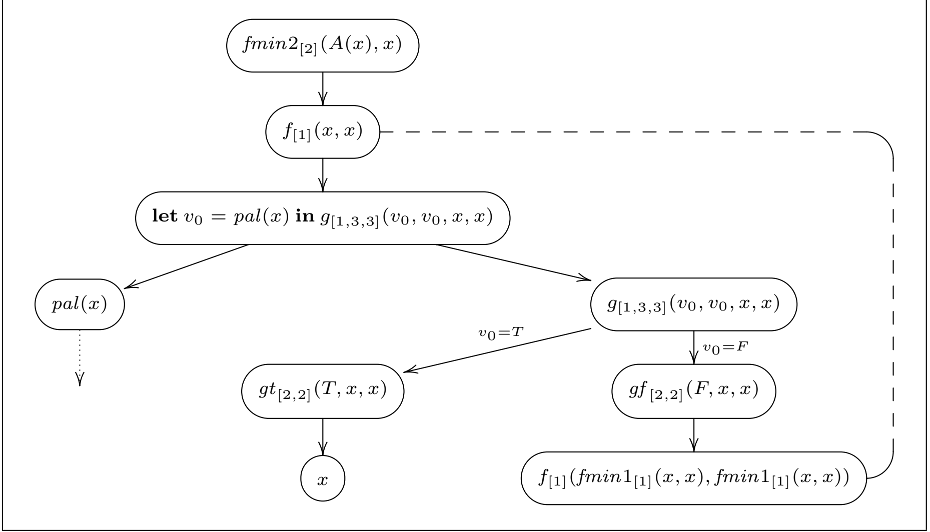

Fig. 8. A fragment of the updated process tree of Example 6.

become possible after substitution of the contraction, namely “normalize” the configuration. Then the repeated call $\mathit{ff}(x_3)$ is found at the fifth step along the following path:

$$\begin{array}{rcl} & & f(x_0) \\ x_0 = S(x_1) & \longrightarrow & \mathit{ff}(x_1) \\ x_1 = S(x_2) & \longrightarrow & add(f(x_2), f(S(x_2)) \\ & \longrightarrow & add(f(x_2), \mathit{ff}(x_2)) \\ x_2 = S(x_3) & \longrightarrow & add(\mathit{ff}(x_3), add(f(x_3), f(S(x_3))) \\ & \longrightarrow & add(\underline{\mathit{ff}(x_3)}, add(f(x_3), \underline{\mathit{ff}(x_3)})) \end{array} \tag{14}$$

The equality indexes do not capture the repeated call $\mathit{ff}(x_3)$ as it occurs as arguments of two different calls to add. Moreover, with the common termination strategy based on homeomorphic embedding, this configuration is not reached since the whistle blows earlier and premature generalization is performed. Hence, to superlinearly speed-up definitions, such as the Fibonacci example a new termination strategy is needed that enables performance of more driving steps before generalization.

5.4 Discussion

The two parts of our method can be considered separately.

Equality indexes and their treatment as function names can be used to construct a whistle that is more precise than the homeomorphic embedding relation used in the standard generalization algorithm [24]. It may be combined together with known refinements of the homeomorphic embedding relation, *e.g.* [20], forcing the relation to distinguish between strict and non-strict substitutions. In simple cases like Example 4,

using the strict embedding alone can also avoid the problem of unselective generalization. In case of comparing other configuration, *e.g.* $f(x, y)$ and $f(g(x, x, y), g(x, x, y))$, none of the refined embedding relations ($\trianglelefteq^*$, $\trianglelefteq_{var}$, $\trianglelefteq^+$) [20] can steer clear of the unselective generalization. Equality indices help to avoid the loss of sharing in this case.

Synchronization preserves term equalities including any syntactically lost due to standard driving. The synchronization feature can be viewed as a very simple version of jungle driving [21] that treats equal terms as one, but due to its simplicity and textual representation, our approach does not require a change of driving.

Partial deduction and driving are transformation techniques that achieve their transformational effects by nonlinear configurations [6]. Thus, our method may be not only useful in supercompilation, but in the context of verification by abstraction-based partial deduction [5] and other advanced program transformers [3].

6 Conclusion

We demonstrated how core supercompilation as it was originally defined by Valentin Turchin and used in many works, is capable of achieving superlinear (up to exponential) speedups on certain programs that repeatedly evaluate some function calls with the same arguments. Although core supercompilation does not check for repeated subterms in configurations explicitly (although it could), it contains an operation that captures equal terms implicitly: *most specific generalization*.

We described two types of supercompilation: SCP (2) using identical folding without online generalization, and SCP (1) with more sophisticated folding and generalization strategies. Identical folding limits SCP (2) to linear speedups similar to the limit of partial evaluation, but performs deeper transformations than partial evaluation due to the unification-based information propagation of driving (*e.g.*, SCP (2) passes the KMP-test [4, 25] which partial evaluation does not [12]).

Folding is a powerful technique which in combination with other transformation techniques can perform deep program optimizations (*e.g.*, improve the naive Fibonacci function [1]). The limitation of SCP (2) lies only in the restriction to identical folding, not in driving. We showed that even a core supercompiler using driving and most-specific generalization could change the asymptotic time complexity of programs, and dispelled the myth that supercompilation is only capable of linear-time program speedups.

A possible reason for the persistence of this myth is that it is sometimes forgotten that generalization without checking the equality of subterms, *e.g.* the generalization of $f(g(x), g(x))$ and $f(h(a, b), h(a, b))$ to $f(x, y)$, is *not* most specific. The most specific in this case being the generalization to $f(x, x)$ with two substitutions $\{x := g(x)\}$ and $\{x := h(a, b)\}$. When these substitutions are residualized as assignments, they evaluate the terms $g(x)$ and $h(a, b)$ only once. We refer to this phenomenon as *msg-sharing*.

Several methods more powerful than core supercompilation have been proposed that achieve superlinear speedup: distillation [10], various forms of higher-level supercompilation [9, 17], including old ideas of walk grammars by Valentin Turchin [26, 29] which are grounds for further exploration.

Nevertheless, gradual extensions of core supercompilation (which may be termed *first-level supercompilation* as they mainly operate on configurations rather than graphs,

walk grammars, *etc.*) are also possible. Here are some ideas which have been investigated or proposed, some of which have been supercompilation folklore for decades:

- Look for repeated subterms in each configuration and restructure the configuration into a let-term if found.
- *Collapsed jungle driving* [21, 22] generalizes the previous idea and formalizes it in terms of dag representation of configurations and operation of collapsing that merges topmost nodes of equal subgraphs.
- Various techniques simpler than dags, which achieve the same effects, but on fewer programs, can be added to existing supercompilers. We demonstrated one such method consisting of the addition of equality indices and the synchronization of terms after driving (Section 5). This method captures repeated subterms occurring in the list of the arguments of the same function call. It is less powerful than the use of dags, but it is much simpler because it deals with terms rather than dags plus it works with the operations of driving, generalization, homeomorphic embedding, whistles, and preserves termination proofs, while the respective operations on dags and the proofs are to be developed afresh.

Future work In preparing this paper the authors found the topic of capturing repeated function calls in first-level supercompilation was undeveloped deserving more attention. Possibly, the only systematic study in the context of supercompilation is [22]. The topic should be revisited in the context of modern research in supercompilation. Another avenue to explore is “old chestnut” problems including the naive Fibonacci function, the sum of factorials, *etc.* Solutions of their superlinear speedup by extended core supercompilation should be presented and used as test cases to develop algorithmic supercompilation strategies that reveal repeated subterms. One of the intriguing questions is the role of multi-result supercompilation [8, 14, 15, 18, 19] in these solutions, and what can be done in the single-result case. First-level methods should be compared with higher-level ones such as distillation, and their relative power and inherent limits should be studied.

Acknowledgments. The authors thank the anonymous reviewers for their constructive feedback. This work benefited greatly from discussions with the participants of the Fourth International Valentin Turchin Workshop on Metacomputation. The first author expresses his deepest thanks to Akihiko Takano for providing him with excellent working conditions at the National Institute of Informatics, Tokyo, and Masami Hagiya, Kanae Tsushima, and Zhenjiang Hu for their invaluable support.

References

1. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (1977)
2. Christensen, N.H., Glück, R.: Offline partial evaluation can be as accurate as online partial evaluation. ACM TOPLAS 26(1), 191–220 (2004)
3. Futamura, Y., Konishi, Z., Glück, R.: Program transformation system based on generalized partial computation. New Generation Computing 20(1), 75–99 (2002)

4. Glück, R., Klimov, A.V.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Cousot, P., et al. (eds.) Static Analysis. Proceedings. pp. 112–123. LNCS 724, Springer-Verlag (1993)
5. Glück, R., Leuschel, M.: Abstraction-based partial deduction for solving inverse problems: a transformational approach to software verification. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) Perspectives of System Informatics. Proceedings. pp. 93–100. LNCS 1755, Springer-Verlag (2000)
6. Glück, R., Sørensen, M.H.: Partial deduction and driving are equivalent. In: Hermenegildo, M., Penjam, J. (eds.) Programming Language Implementation and Logic Programming. Proceedings. pp. 165–181. LNCS 844, Springer-Verlag (1994)
7. Glück, R., Sørensen, M.H.: A roadmap to metacomputation by supercompilation. In: Danvy, O., Glück, R., Thiemann, P. (eds.) Partial Evaluation. Proceedings. pp. 137–160. LNCS 1110, Springer-Verlag (1996)
8. Grechanik, S.A.: Overgraph representation for multi-result supercompilation. In: Klimov and Romanenko [16], pp. 48–65
9. Grechanik, S.A.: Inductive prover based on equality saturation for a lazy functional language. In: Virbitskaite, I., Voronkov, A. (eds.) Perspectives of System Informatics. Proceedings. pp. 127–141. LNCS 8974, Springer-Verlag (2015)
10. Hamilton, G.W.: Distillation: extracting the essence of programs. In: Partial Evaluation and Program Manipulation. Proceedings. pp. 61–70. ACM Press (2007)
11. Jones, N.D.: Transformation by interpreter specialization. Science of Computer Programming 52(1-3), 307–339 (2004)
12. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993)
13. Jones, N.D., Hamilton, G.W.: Towards understanding superlinear speedup by distillation. In: Klimov, A.V., Romanenko, S.A. (eds.) Fourth International Valentin Turchin Workshop on Metacomputation. Proceedings. pp. 94–109. University of Pereslavl, Russia (2014), http://meta2014.pereslavl.ru/papers/2014_Jones_Hamilton__Towards_Understanding_Superlinear_Speedup_by_Distillation.pdf
14. Klimov, A.V.: Why multi-result supercompilation matters: Case study of reachability problems for transition systems. In: Klimov and Romanenko [16], pp. 91–111
15. Klimov, A.V., Klyuchnikov, I.G., Romanenko, S.A.: Automatic verification of counter systems via domain-specific multi-result supercompilation. In: Klimov and Romanenko [16], pp. 112–141
16. Klimov, A.V., Romanenko, S.A. (eds.): Third International Valentin Turchin Workshop on Metacomputation. Proceedings. University of Pereslavl, Russia (2012)
17. Klyuchnikov, I., Romanenko, S.: SPSC: a simple supercompiler in Scala. In: Bulyonkov, M.A., Glück, R. (eds.) Program Understanding. Proceedings. Ershov Institute of Informatics Systems, Russian Academy of Sciences, Novosibirsk, Russia (2009), http://spsc.googlecode.com/files/Klyuchnikov__Romanenko__SPSC_a_Simple_Supercompiler_in_Scala.pdf
18. Klyuchnikov, I.G., Romanenko, S.A.: Formalizing and implementing multi-result supercompilation. In: Klimov and Romanenko [16], pp. 142–164
19. Klyuchnikov, I.G., Romanenko, S.A.: Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In: Clarke, E.M., Virbitskaite, I., Voronkov, A. (eds.) Perspectives of System Informatics. Proceedings. pp. 210–226. LNCS 7162, Springer-Verlag (2012)
20. Leuschel, M.: Improving homeomorphic embedding for online termination. In: Flener, P. (ed.) Logic-Based Program Synthesis and Transformation. Proceedings. pp. 199–218. LNCS 1559, Springer-Verlag (1999)
21. Secher, J.P.: Driving in the jungle. In: Danvy, O., Filinsky, A. (eds.) Programs as Data Objects. Proceedings. pp. 198–217. LNCS 2053, Springer-Verlag (2001)

22. Secher, J.P.: Driving-based Program Transformation in Theory and Practice. Ph.D. thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark (2002)
23. Sørensen, M.H.: Turchin's supercompiler revisited. DIKU Report 94/9, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark (1994)
24. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In: Lloyd, J.W. (ed.) Logic Programming: Proceedings of the 1995 International Symposium. pp. 465–479. MIT Press (1995)
25. Sørensen, M.H., Glück, R., Jones, N.D.: A positive supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)
26. Turchin, V.F.: The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University (1980)
27. Turchin, V.F.: The concept of a supercompiler. ACM TOPLAS 8(3), 292–325 (1986)
28. Turchin, V.F.: The algorithm of generalization in the supercompiler. In: Bjørner, D., Ershov, A.P., Jones, N.D. (eds.) Partial Evaluation and Mixed Computation. pp. 531–549. North-Holland (1988)
29. Turchin, V.F.: Program transformation with metasystem transitions. Journal of Functional Programming 3(3), 283–313 (1993)
30. Turchin, V.F.: Supercompilation: techniques and results. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) Perspectives of System Informatics. Proceedings. pp. 227–248. LNCS 1181, Springer-Verlag (1996)