Towards Higher-Level Supercompilation*

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
ilya.klyuchnikov@gmail.com romansa@keldysh.ru

Abstract. We show that the power of supercompilation can be increased by constructing a hierarchy of supercompilers, in which a lower-level supercompiler is used by a higher-level one for proving *improvement lemmas*. The lemmas thus obtained are used to transform expressions labeling nodes in process trees, in order to avoid premature generalizations. Such kind of supercompilation, based on a combination of several metalevels, is called *higher-level* supercompilation (to differentiate it from *higher-order* supercompilation related to transforming higher-order functions). Higher-level supercompilation may be considered as an application of a more general principle of *metasystem transition*.

1 Introduction

The concept of *metasystem transition* was introduced by V.F.Turchin in his 1977 book *The Phenomenon of Science* [26]. In the context of computer science, Turchin gives the following (somewhat simplified) formulation of the main idea of metasystem transition [28]:

> Consider a system S of any kind. Suppose that there is a way to make some number of copies of it, possibly with variations. Suppose that these systems are united into a new system S' which has the systems of the S type as its subsystems, and includes also an additional mechanism which somehow examines, controls, modifies and reproduces the S-subsystems. Then we call S' a *metasystem* with respect to S, and the creation of S' a *metasystem transition*. As a result of consecutive metasystem transitions a multilevel hierarchy of control arises, which exhibits complicated forms of behavior.

Futamura projections [6] may serve as a good example of metasystem transition. Let p be a program, i an interpreter, and s a program specializer. Then $s(i, p)$ may be regarded as a compiled program (the "first projection"), $s(s, i)$ as a compiler (the "second projection") and $s(s, s)$ as a compiler generator (the "third

* Supported by Russian Foundation for Basic Research projects No. 08-07-00280-a and No. 09-01-00834-a.

projection"). (The second Futamura projection is also referred to by Ershov as "Turchin's theorem of double driving" [5].)

In the second projection the evaluation of $s(s, i)$ involves two copies of the specializer s, the second copy "examining and controlling" the first one. In the third projection, there are 3 copies of s, the third one "controlling" the second one "controlling" the first one. Moreover, as shown by Glück [7], there may be considered the "fourth" Futamura projection, corresponding to the next metasystem transition!

Futamura projections, however, are not the only possible way of exploiting the idea of metasystem transition by combining program transformers. In the present paper we consider another technique of constructing a multilevel hierarchy of control using *supercompilers* as its building blocks.

A supercompiler is a program transformer based on *supercompilation* [27], a program transformation technique bearing close relation to the fold/unfold method by Burstall and Darlington [4]. Unfortunately, "pure" supercompilation is known not to be very good at transforming non-linear (i.e. containing repeated variables) expressions and functions with accumulating parameters.

We argue, however, that the power of supercompilation can be increased by combining several copies of a "classic" supercompiler and making them control each other. Such kind of supercompilation, based on a combination of several metalevels will be called *higher-level* supercompilation (to differentiate it from *higher-order* supercompilation related to transforming higher-order functions).

The technique suggested in the paper is (conceptually) simple and modular, and is based on the use of *improvement lemmas* [20,21], which are automatically generated by lower-level supercompilers for a higher-level supercompiler.

2 Higher-Level Supercompilation

2.1 What is a "zero-level" supercompiler?

The descriptions of supercompilation given in the literature differ in some secondary details, irrelevant to the main idea of higher-level supercompilation. We follow the terminology and notation used by Sørensen and Glück [24,23,25].

All program transformation examples considered in the paper have been carried out by HOSC [13,14], a higher-order supercompiler whose general structure is shown in Fig. 1.

2.2 Accumulating parameter: "zero-level" supercompilation

Let us try to apply the supercompiler HOSC [13] to the program shown in Fig. 2. At the beginning, a few steps of driving produce the process tree shown in Fig. 3.

At this point the whistle signals that there is a node b embedding a previously encountered node a, but b is not an instance of a:

```
case double x Z of {Z → True; S y → odd y);}
    ⊴c case double n (S (S Z)) of {Z → True; S m → odd m;}
```

```
def scp(tree)
  b = unprocessed_leaf(tree)
  if b == null
    return makeProgram(tree)
  if trivial(b)
    return scp(drive(b, tree))
  a = ancestor(tree, b, renaming)
  if a != null
    return scp(fold(tree, a, b))
  a = ancestor(tree, b, instance)
  if a != null
    return scp(abstract(tree, b, a))
  a = ancestor(tree, b, whistle)
  if a == null
    return scp(drive(b, tree))
  return scp(abstract(tree, a, b))
```

- `unprocessed_leaf(tree)` returns an unprocessed leaf `b` in the process tree.
- `trivial(b)` checks whether the leaf `b` is "trivial". (A leaf is trivial if driving it does not result in unfolding a function call or applying a substitution to a variable.)
- `drive(b,tree)` performs a driving step for a node `b` and returns the modified tree.
- `ancestor(tree,b,renaming)` returns a node `a` such that `b` is a renaming of `a`.
- `ancestor(tree,b,instance)` returns a node `a` such that `b` is an instance of `a`.
- `ancestor(tree,b,whistle)` returns a node `a` such that `a` is homeomorphically embedded in `b` by coupling.
- `fold(t,a,b)` makes a cycle in the process tree from `b` to `a`.
- `abstract(tree,a,b)` generalizes `a` to `b`.

Fig. 1. "Zero-level" supercompilation algorithm

```
data Bool = True | False;
data Nat = Z | S Nat;

even (double x Z) where

even = λx → case x of { Z → True; S x1 → odd x1;};
odd = λx → case x of { Z → False; S x1 → even x1;};

double = λx y → case x of { Z → y; S x1 → double x1 (S (S y));};
```

Fig. 2. even (double x Z): source program

Hence, HOSC has to throw away the whole subtree under a and "generalize" a by replacing a with a new node a', such that a and b are instances of a'. Then the supercompilation continues to produce the residual program shown in Fig. 4.

This result is correct, but it is not especially exciting! In the goal expression `even (double x Z)` the inner function `double` multiplies its argument by 2, and the outer function `even` checks whether this number is even. Hence, the whole expression never returns `False`. But this can not be immediately seen from the residual program, the program text still containing `False`.

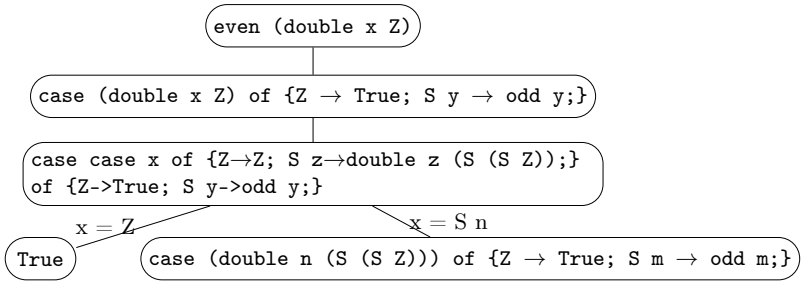

Fig. 3. even (double x Z): driving

```
letrec
  f=λw2 λp2→
    case w2 of {
      Z →
        letrec g=λr2→
          case r2 of {
            S r → case r of {Z → False; S z2 → g z2;};
            Z → True;}
        in g p2;
      S z → f z (S (S p2));
    }
in f x Z
```

Fig. 4. The result of "zero-level" supercompilation

2.3 Accumulating parameter: applying a lemma

As was pointed out by Burstall and Darlington [4], the power of a program transformation system can be increased by enabling it to use "laws" or "lemmas" (such as the associativity and commutativity of addition and multiplication). In terms of supercompilation it amounts to replacing a node in the process tree with an "equivalent" one.

Let us return to the tree in Fig. 3. The result of supercompilation was not good enough, because HOSC had to generalize a node, and a generalization resulted in a "loss of precision". But we can avoid generalization by making use of the following (mysterious) equivalence

```
case double n (S (S Z)) of {Z → True; S m → odd m;} ≅
    ≅ even (double n Z)
```

Since the node `even (double n Z)` is a renaming of an upper one, the supercompiler can now form a cycle to produce the program shown in Fig. 5. Note that this time there is no occurrences of `False`, hence the supercompiler succeeded in proving that `False` can never be returned by the program.

```
letrec f=λt→
  case t of {Z → True; S s → f s;}
in f x
```

Fig. 5. The result of applying a lemma

```
def scp(tree, n)
  b = unprocessed_leaf(tree)
  if b == null
    return makeProgram(tree)
  if trivial(b)
    return scp(drive(b, tree), n)
  a = ancestor(tree, b, renaming)
  if a != null
    return scp(fold(t, a, b), n)
  a = ancestor(tree, b, instance)
  if a != null
    return scp(abstract(tree, b, a))
  a = ancestor(tree, b, whistle)
  if a == null
    return scp(drive(b, tree), n)
  if n > 0
    e = findEqExpr(b.expr, n)
    if e != null
      return scp(replace(tree, b, e), n)
  return scp(abstract(tree, a, b))
```

```
def findEqExpr(e1, n)
  e = scp(e1, n-1)
  cands = candidates(e1, n)
  for cand <- cands
    if equivalent(
         scp(cand, n-1), e)
      return cand
  return null

def candidates(e1, n)
  ...

...
```

Fig. 6. "Multi-level" supercompilation algorithm

Hence, there are good reasons to believe that lemmas are a good thing, but there appear two questions: (1) how to prove lemmas, and (2) how to find useful lemmas.

2.4 Proving lemmas by supercompilation

In [15] we have shown that the supercompiler HOSC [13] may be used for proving interesting equivalences of higher-order expressions. The technique is quite straightforward. Let e_1 and e_2 be expressions appearing in a program p. Let e'_1 and e'_2 be the results of supercompiling e_1 and e_2 with respect to p. Then, if e'_1 and e'_2 are the same (modulo alpha-renaming), then the original expressions e_1 and e_2 are equivalent (provided that the supercompiler strictly preserves the equivalence of programs). Since e_1 and e_2 may contain free variables, the use of a higher-order supercompiler enables us to prove equalities with universal quantification over functions and infinite data types by using a higher-order supercompiler.

```
def scp(tree, n)
  b = unprocessed_leaf(tree)
  if b == null
    return [makeProgram(tree)]
  if trivial(b)
    return scp(drive(b, tree), n)
  a = ancestor(tree, b, renaming)
  if a != null
    return scp(fold(t, a, b), n)
  a = ancestor(tree, b, instance)
  if a != null
    return scp(abstract(tree, b, a))
  a = ancestor(tree, b, whistle)
  if a == null
    return scp(drive(b, tree), n)
  progs = scp(abstract(tree, a, b))
  if n > 0
    for e <- findEqExpr(b.expr, n)
      progs ++= scp(replace(tree, b, e), n)
  return progs
```

```
def findEqExpr(e1, n)
  es = scp(e1, n-1)
  cands = candidates(e1, n)
  exps = []
  for cand <- cands
    if not_disjoint(
          scp(cand, n-1), es)
      exps = exps + cand
  return exps

def candidates(e1, n)
  ...

...
```

Fig. 7. "Multi-level" supercompilation algorithm: multiple residual programs

Thus the reasoning about operational equivalence ($\cong$) of programs can be reduced to a trivial check of the syntactic equality of supercompiled programs. Since all residual programs produced by HOSC are expressions (that may contain letrec-subexpressions), checking the equality of residual programs boils down to a syntactical comparison of expressions.

Generally speaking, the idea of proving equivalence by normalization is a well-known one, being a standard technique in such fields as computer algebra. The idea of using supercompilation for normalization is due to Lisitsa and Webster [17], who have successfully applied supercompilation for proving the equivalence of programs written in a first-order functional language, on condition that the programs deal with finite input data and are guaranteed to terminate. Later it has been found [15,13] that these restrictions can be lifted in cases where the supercompiler deals with programs in a lazy functional language and preserves the termination properties of programs.

2.5 Stacking supercompilers, jumping to higher-level supercompilation

As we have seen the power of supercompilation can be increased by using lemmas (i.e. replacing some expressions with equivalent ones). On the other hand, supercompilers can be used for checking the equality of expressions. Hence, recalling the principle of metasystem transition [26,28], we come to the following idea: let us construct a tower of supercompilers, the higher-level ones running the lower-level ones in order to obtain useful lemmas.

This can be done by adding to the function `scp(tree)` shown in Fig. 1 an additional parameter `n`, the "level" the supercompiler is invoked at. The modified supercompilation algorithm is shown in Fig. 6.

Note that for `n = 0` the algorithm degrades to the "classic" supercompilation. But in cases where `n > 0` the supercompiler calls the function `findEqExpr(b.expr,n)` passing to it the expression in the node `b` and the current level. The function tries to produce an expression equivalent to `b.expr` by generating a set of candidate expressions and selecting an expression equivalent to `b.expr`. The check for equivalence is performed by invoking the same supercompiler at the lower level `n - 1`.

2.6 Generating sets of residual programs

The algorithm in Fig. 6 assumes that there is a single result of supercompilation. However, there are certain points in the process of supercompilation, where the supercompiler has an opportunity to make a choice among several options, so that, given a source program, several (equivalent) residual programs may be generated.

This may be used for increasing the power of the supercompilation-based equality check. Suppose we have to check for equivalence two expressions e_1 and e_2. Then, instead of generating and comparing just two residual expressions e'_1 and e'_2, we can supercompile e_1 and e_2 to produce two sets of residual expressions and try to find a residual expression common to both sets (modulo alpha-renaming).

To implement this idea we need a version of a supercompilation algorithm producing a set of residual programs (see Fig. 7). This version, instead of choosing an arbitrary acceptable expression from the set of candidate expressions, returns the set of all acceptable expressions. Note that for any n the set of residual programs produced by the algorithm includes the result returned by the zero-level supercompiler.

2.7 A few open questions

Fig. 6 and Fig. 7 present the general idea of higher-level supercompilation, but there still remains a few open questions.

Correctness At the first glance, replacing an expression with an equivalent one looks as a "natural" and "safe-by-construction" operation. And yet, as shown by Sands [20,21], the unrestricted use of equivalences may lead to incorrect transformations that do not preserve the meaning of programs.

How to generate candidate expressions Supercompilation can be used for checking the equivalence of expressions, but it does not help us in finding candidate expressions that are worth being checked for equivalence.

3 Correctness = equivalence + improvement

3.1 Notation

It is clear that the meaning of expressions may depend on the context. Thus, to avoid making our notation unnecessarily cumbersome, when speaking about the equivalence of expressions, we will assume that the expressions are evaluated and supercompiled in the context of the same program.

We use $\mathcal{SC}[\![e]\!]$ to denote the expression produced by supercompiling the expression e by a "classic", "zero-level" supercompiler, and $e \equiv e'$ to denote the fact that e is the same as e' (modulo alpha-renaming).

3.2 Operational equivalence

Definition 1 (Operational approximation). *An expression e operationally approximates e', $e \mathrel{\tilde{\sqsubseteq}} e'$, if for all contexts C such that $C[e]$, $C[e']$ are closed, if the evaluation of $C[e]$ terminates then so does the evaluation of $C[e']$.*

Definition 2 (Operational equivalence). *An expression e is operationally equivalent to e', $e \cong e'$ if $e \mathrel{\tilde{\sqsubseteq}} e'$ and $e' \mathrel{\tilde{\sqsubseteq}} e$.*

In the following we assume the supercompilers to preserve operational equivalence, i.e. that $e' = \mathcal{SC}[\![e]\!]$ implies $e' \cong \mathcal{SC}[\![e]\!]$ (which is true of the supercompiler HOSC [13]).

3.3 Improvement

The replacement of an expression e with an equivalent expression e', followed by a fold, may result in producing an incorrect residual program (some examples can be found in [21]).

Definition 3 (Improvement). *An expression e is improved by e', $e \trianglerighteq e'$, if for all contexts C such that $C[e]$ and $C[e']$ are closed, if the computation of $C[e]$ terminates using n function calls, then the computation of $C[e']$ also terminates, and uses no more than n function calls.*

As has been shown by Sands [21], the replacement of an expression e_1 with an expression e_2 will not violate the correctness of transformation if the following conditions are met: $e_1 \cong e_2$ and $e_1 \trianglerighteq e_2$.

Definition 4 (Improvement lemma). *A pair (e_1, e_2) is an improvement lemma if $e_1 \cong e_2$ and $e_1 \trianglerighteq e_2$.*

3.4 Checking the improvement relation by supercompilation

Let e_1 and e_2 be expressions whose equivalence has been proven by supercompilation. Does it mean that one of the expressions is an improvement over the other one? Not at all!

Supercompiling the following two expressions with respect to the program shown in Fig. 10 proves them to be operationally equivalent:

```
or (even n) (odd n)
   ≅ case (even n) of {True → True; False → odd (S (S n));}
```

However, neither is an improvement over another one. Indeed, if n = `Z`, the evaluation of the expressions involves 2 and 1 function calls, respectively. But, if n = `S Z`, the evaluation involves 5 and 6 function calls. Therefore, this lemma is unsafe to be used in program transformation.

Fortunately, the check that $e_1 \trianglerighteq e_2$ holds for two expressions e_1 and e_2, such that $e_1 \cong e_2$, can also be performed by supercompilation! And this can be done almost for free in the following way.

In order to check e_1 and e_2 for equivalence, we have to supercompile them to e'_1 and e'_2. The check for equivalence succeeds if e'_1 and e'_2 are the same (modulo alpha-renaming). Therefore, e'_1 and e'_2 contain insufficient information to make any conclusions about e_2 being an improvement over e_1. However, the process trees produced by supercompiling e_1 and e_2 contain more information than the residual expressions.

Namely, let us examine the process trees and mark with a star (*) the edges corresponding to an unfolding (a function call). For example, supercompiling the expressions considered above produces the process trees shown in Fig. 12 and Fig. 14 (the subtrees for `odd x` are omitted for brevity). Now the starred edges provide some information that can be used for checking that an expression improves another one. But this information is not accessible from outside. But, in the case of HOSC [13], this information can be made visible by modifying the algorithm that converts process trees into residual expression.

The modified algorithm converts starred edges into annotations in residual expressions. When traversing a starred edge, the residual expression produced by traversing the (single) child node is annotated with a star (*). In this way the information about unfolds is recorded in residual expressions, so that a lower-level supercompiler can be used as a "black box".

For example, Fig. 13 and Fig. 16 show the annotated programs produced from the process trees in Fig. 12 and Fig. 14.

Now, let $\rhd^*_\sim$ denote a binary relation on expressions such that $e \rhd^*_\sim e'$ iff (1) e and e' differ only in their annotations, and (2) e can be transformed into e' by erasing some stars in e. See Fig. 8 for a more formal definition, where ${}^n\phi$ denotes a functor ϕ prefixed with n stars. (It is curious to note that $\rhd^*_\sim$ can be considered as a special case of homeomorphic embedding relation.)

Theorem 1. *Let* $e'_1 = \mathcal{SC}[\![e_1]\!]$ *and* $e'_2 = \mathcal{SC}[\![e_1]\!]$. *If* $e'_1 \equiv e'_2$ *and* $e'_1 \rhd^*_\sim e'_2$, *then* $e_1 \trianglerighteq e_2$.

$$\frac{m \geq n \qquad \forall i : e_i \rhd^*_\sim e'_i}{{}^m\phi(e_1, \ldots, e_k) \rhd^*_\sim {}^n\phi(e'_1, \ldots, e'_k)} \qquad \frac{\mathcal{SC}[\![e_1]\!] \cong \mathcal{SC}[\![e_2]\!] \qquad \mathcal{SC}[\![e_1]\!] \rhd^*_\sim \mathcal{SC}[\![e_2]\!]}{e_1 \trianglerighteq e_2}$$

Fig. 8. Estimation of improvement based on annotated supercompiled expressions

Proof. Since $e'_1 \equiv e'_2$, e'_1 is the same as e'_2 (modulo annotations and a bound variable renaming). Therefore, if we put the expressions in the same context C and try to evaluate $C[e'_1]$ and $C[e'_2]$ (disregarding the annotations), this will result in two sequences of reduction steps, differing only in the number of stars encountered during computation. Since $e'_1 \rhd^*_\sim e'_2$, after any number of reduction steps, the number of stars encountered in the evaluation of $C[e'_2]$ cannot be greater that the number of stars encountered in the evaluation of $C[e'_1]$, and the stars correspond to unfoldings in the original expressions e_1 and e_2. So, by evaluating $C[e'_1]$ and $C[e'_2]$ and counting stars we can count the number of unfolds in the evaluation of $C[e_1]$ and $C[e_2]$. Hence, the number of unfolds in the evaluation of $C[e_2]$ is no more than in the evaluation of $C[e_1]$. Therefore $e_1 \trianglerighteq e_2$.

Thus, by examining annotated supercompiled expressions, we can check the improvement relation for the original expressions. For example, consider the annotated supercompiled expressions for

```
or (even n) (odd n)
```

and

```
case (even n) of {True → True; False → odd (S (S n));}
```

shown in Fig. 13 and Fig. 16. Since the supercompiled expressions are not related by $\rhd^*_\sim$, we cannot make the conclusion that the original expressions are related by $\trianglerighteq$.

4 A proof-of-concept implementation

Although, the general idea of higher-level supercompilation is conceptually simple, there are a number of problems to be solved in a practical implementation.

- Which "zero-level" supercompiler to use as the basic for implementing higher-level supercompilation?
- How to guarantee the correctness of transformations?
- How to generate useful lemmas?
- How to ensure the termination of higher-level supercompilation?

To show the feasibility of higher-level supercompilation we have implemented a simple "proof-of-concept" higher-level supercompiler HLSC by modifying the supercompiler HOSC [13,15]. HOSC has been chosen because it (1) preserves

the meaning of programs (including their termination properties), (2) is able to prove lemmas with universal quantification over functions and infinite data types, (3) generates residual programs in the form of expressions, which enables the program equivalence checking to be reduced to expression equivalence checking.

The correctness of the transformations is guaranteed, since HLSC uses lemmas that are improvement ones. Note, however, that the check for improvement is based on "zero-level" supercompilation, for which reason HLSC currently implements only a two-level hierarchy of supercompilers, rather than a multi-level one, consisting of the "top" and "bottom" supercompilers.

The least elaborated points are the search for useful lemmas and ensuring the termination of higher-level supercompilation.

$$
\begin{array}{ll}
\mathcal{S}[\![v]\!] & = 1 \\
\mathcal{S}[\![c\ \overline{e_i}]\!] & = 1 + \sum_i \mathcal{S}[\![e_i]\!] \\
\mathcal{S}[\![\lambda v \to e]\!] & = 1 + \mathcal{S}[\![e]\!] \\
\mathcal{S}[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] & = 1 + \mathcal{S}[\![e_0]\!] + \sum_i \mathcal{S}[\![e_i]\!] \\
\mathcal{S}[\![e_1\ e_2]\!] & = \mathcal{S}[\![e_1]\!] + \mathcal{S}[\![e_2]\!]
\end{array}
$$

Fig. 9. The size of expression

Presently the generation of candidate expressions is implemented in a rather crude and straightforward way. When the top supercompiler finds a node b containing an expression e and embedding a previously encountered node a, it generates and tries all expressions e' whose size (see Fig. 9) is less than the size of e. Then the bottom supercompiler is used to check whether (e, e') is an improvement lemma and the search for a lemma stops.

Ensuring the termination of the top supercompiler is an exciting problem that requires further investigation. The termination of zero-level supercompilation is achieved by the check for homeomorphic embedding and generalization [23,22,13]. However, the main idea of higher-level supercompilation (in the version presented in Fig. 6) consists in avoiding generalization. When an embedding is detected, the use of an improvement lemma enables the supercompiler to avoid generalization and continue to build the process tree. And this, potentially, may lead to non-termination.

From the practical point of view, though, the non-termination can be avoided by imposing some restrictions on the use of lemmas. A simple and straightforward solution is the following.

Suppose, the check for embedding finds that there an upper node a is embedded in a lower node b. Then b is replaced with b' with the aid of an improvement lemma, and supercompilation continues without generalization. But this fact is recorded in the node a, so that next time when a gets embedded in another node, no lemma will be used, and a will be generalized as in the case of zero-level supercompilation.

```
data Bool = True | False;
data Nat = Z | S Nat;

or (even m) (odd m) where

even = λx → case x of { Z → True; S x1 → odd x1;};
odd = λx → case x of { Z → False; S x1 → even x1;};

or = λx y → case x of { True → True; False → y;};
```

Fig. 10. or (even m) (odd m): the source program

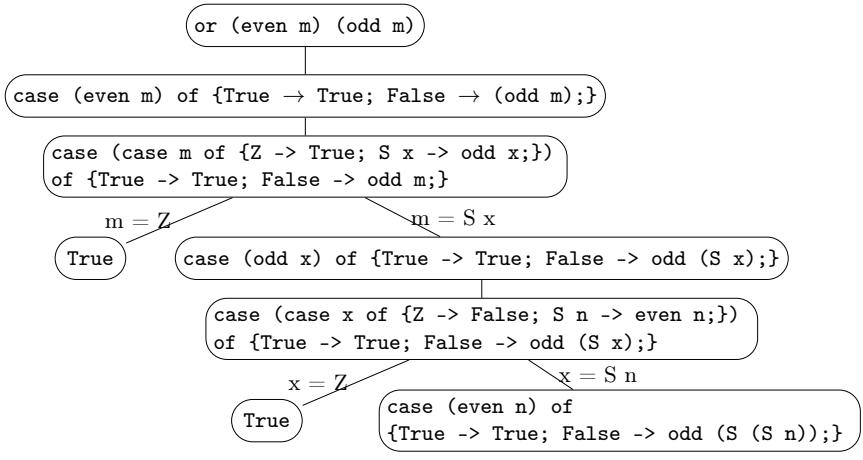

Fig. 11. *or* (*even m*) (*odd m*): the whistle blows

To some extent this idea is alike to "cross-fertilization" used by Boyer and Moore [2] in their theorem prover. They argue that the induction hypothesis should be used just once, and then thrown away. (And then comes the turn of generalization.)

5 Examples

5.1 Supercompiling a non-linear expression

Let us try to supercompile the program shown in Fig. 10. We know in advance that the expression `or (even m) (odd m)` can never return `False`, since a natural number m is either even or odd. But this cannot be readily seen from the program's text! After a few driving steps, we get the process tree shown in Fig. 11. At this point an embedding (by coupling) is detected:

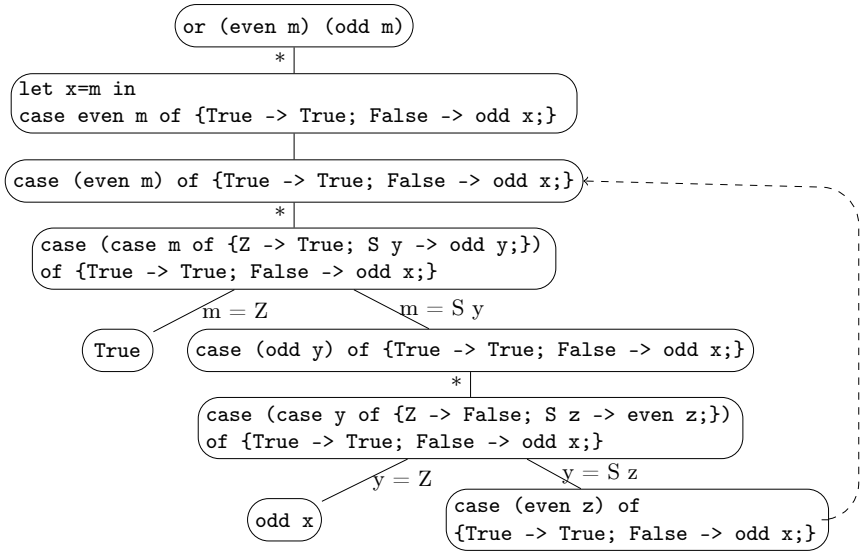

Fig. 12. *or* (*even m*) (*odd m*): after generalization

```
*(letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          letrec g = *(λw→
            case w of {
              Z → False;
              S t → *(case t of {Z → True; S z → g z;});})
           in g m;
        S x → f x;
      });
  })
in f m)
```

Fig. 13. or (even m) (odd m): the result of "zero-level" supercompilation

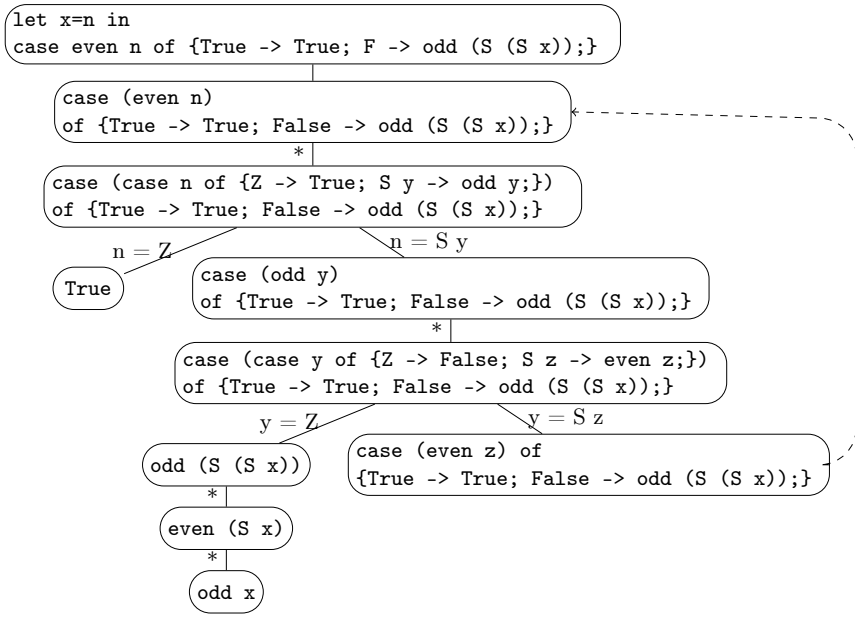

Fig. 14. *case even n of* $\{True \to True; False \to odd\ (S\ (S\ n)); \}$: annotated process tree

```
case (even m) of {True → True; False → (odd m);}
    ⊴c case (even n) of {True → True; False → (odd (S (S n)));}
```

But the second expression is not an instance of the first one, for which reason a folding cannot be performed. So the zero-level supercompiler HOSC would perform a generalization by replacing the first expression with the let-expression:

```
let x = m in case (even m) of {True → True; False → (odd x);}
```

Then it would continue by transforming the body of the let-expression, instead of the original expression, thereby "forgetting" that x and m have the same value. This loss of information would result in the residual program in Fig. 13, containing `False`, despite the fact that `False` can never be returned by the program.

The higher-level version of HOSC, however, tries to find and apply an improvement lemma. The first lemma it finds has the size 5:

```
case (even n) of {True → True; False → (odd (S (S n)));}
    ≅ or (even n) (odd n)
```

But this lemma is not an improvement one, and the higher-level supercompiler rejects it by supercompiling its left and right sides with the bottom supercompiler to produce annotated expressions in Fig. 16 and Fig. 13, respectively.

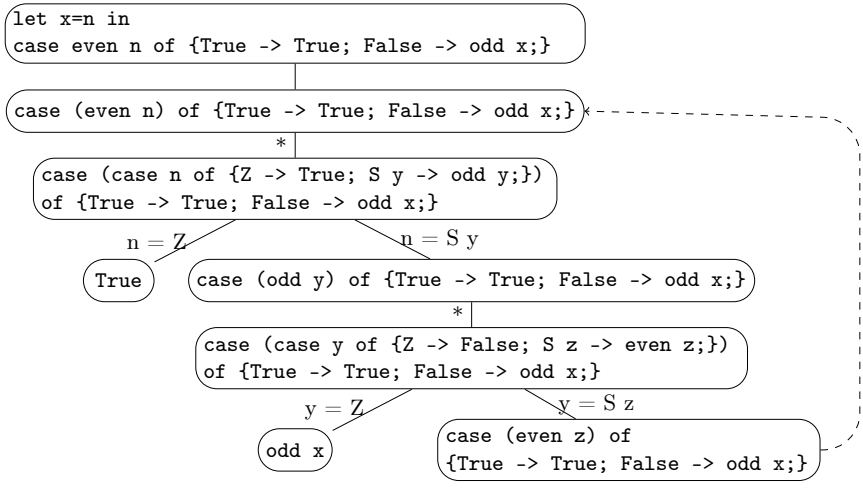

Fig. 15. *case even n of* $\{True \to True; False \to odd\ n;\}$: annotated process tree

```
letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          **(letrec g = *(λw→
            case w of {
              Z → False;
              S t → *(case t of {Z → True; S z → g z;});})
           in g n);
        S x → f x;
      };)
  })
in f n
```

Fig. 16. *case even n of* $\{True \to True; False \to odd\ (S\ (S\ n));\}$: annotated residual program

```
letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          (letrec g = *(λw→
            case w of {
              Z → False;
              S t → *(case t of {Z → True; S z → g z;});})*
           in g n);
        S x → f x;
      };)
  })
in f n
```

Fig. 17. *case even n of* $\{True \to True; False \to odd\ n;\}$: annotated residual program

```
letrec f=λw→
  case w of {
    Z → True;
    S x → case x of { Z → True; S z → f z;};
  }
in f m
```

Fig. 18. or (even m) (odd m): the result of higher-level supercompilation

However, there exist two improvement lemmas of size 6:

```
case (even n) of {True → True; False → odd (S (S n));}
    ≳ case (even n) of {True → True; False → odd n;}

case (even n) of {True → True; False → odd (S (S n));}
    ≳ case (odd n) of {True → odd n; False → True;}
```

The higher-level HOSC finds and applies the first one, thereby avoiding generalization and producing the program in Fig. 18. Now `False` does not appear in the program!

5.2 Accumulating parameter: using an improvement lemma

Let us reconsider the program with an accumulating parameter shown in Fig. 2. If we try to supercompile it, the whistle blows for the following expressions:

```
case double x Z of {Z → True; S y → odd y);}
    ⊴_c case double n (S (S Z)) of {Z → True; S m → odd m;}
```

There are two improvement lemmas (of minimal size):

```
case double n (S (S Z)) of {Z → True; S m → odd m;}
       ⊵ case double n (S Z) of {Z → True; S m → even m;}
case double n (S (S Z)) of {Z → True; S m → odd m;}
       ⊵ case double n (S Z) of {Z → False; S m → even m;}
```

The higher-level HOSC finds and applies the first one and, after some driving, the whistle blows for the second time:

```
case double n (S Z) of {Z → True; S m → even m;}
    ⊴c case double p (S (S (S Z))) of {Z → True; S m → even m;}
```

Again, there are two improvement lemmas (of minimal size):

```
case double p (S (S (S Z))) of {Z → True; S m → even m;}
        ⊵ case double p (S Z) of {Z → True; S m → even m;}
case double p (S (S (S Z))) of {Z → True; S m → even m;}
        ⊵ case double p (S Z) of {Z → False; S m → even m;}
```

The application of the first lemma enables a fold to be performed without generalization, so that the higher-level HOSC produces the program in Fig. 19.

```
case x of {
  Z → True;
  S y1 →
    letrec f=λt2→
      case t2 of {Z → True; S u2 → f u2;}
    in f y1;
}
```

Fig. 19. even (double x Z): the result of higher-level supercompilation

6 Discussion and conclusion

The main idea of higher-level supercompilation is based on the principle of *metasystem transition* [27,28].

Another approach to increasing the power of supercompilation based on metasystem transition is *distillation* [8,10,9].

In many cases distillation and higher-level supercompilation produce similar results, but, seemingly, an advantage of higher-level supercompilation is its conceptual simplicity and modularity: it can be implemented by a slight modification of a "classic" supercompiler, adding a (conceptually) trivial lemma generator, and making several copies of the same supercompiler to interact. Since the lemma generator uses the supercompiler as a "black box", its design does not depend on the subtle details of the supercompilation process.

Our current implementation of higher-level supercompilation is a proof-of-concept one, is rather "naive", and can be improved in a variety of ways.

First, the higher-level supercompilation algorithm shown in Fig. 6 tries to apply a lemma only to the whole embedding (lower) expression. But lemmas could be applied in a more refined way.

- An (instance of an) improvement lemma could be applied to a subexpression of the embedding expression.
- To avoid generalization, an (instance of an) improvement lemma could also be applied to a (sub)expression of the embedded (upper) expression.

Second, the search for lemmas is implemented in a straightforward way: no attempt is made to take into account the structure of the embedding (lower) and embedded (upper) expressions. However, there are a few techniques developed in the field of inductive theorem proving (like difference matching [1], rippling [3] and divergence critic [29]) that could be used in implementing a more refined lemma generator.

Higher-level supercompilation does not depend on minor implementation details of the supercompiler it is based upon. However, some properties of the supercompiler do matter. First of all, the check whether a pair of expressions forms an improvement lemma [20,21] relies on the supercompiler preserving termination properties of programs [15,13]. This requirement is not met by all supercompilers. For example, the supercompiler SCP4 [16] dealing with programs in Refal, a strict first-order functional language, may extend the domain of a transformed function, for which reason the equivalence of expressions can be proven by supercompilation only for total expressions operating on finite data structures [17].

During supercompilation, termination properties are easier to preserve for a lazy functional language, than for a strict one. Nevertheless, Jonsson [11] succeeded in developing a termination-preserving supercompilation technique for a higher-order call-by-value language. Therefore, higher-level supercompilation is certainly applicable to higher-order strict languages.

Since any residual program produced by HOSC is a self-contained expression, the check for equality and improvement amounts to a trivial comparison of expressions. In the case of a supercompiler like Supero [19,18], residual programs may have less trivial structure, therefore, comparing them for syntactic isomorphism may be more intricate that in case of HOSC.

In principle, higher-level supercompilation should be implementable also on the basis of a supercompiler for an imperative or object oriented language, such as the Java supercompiler by Klimov [12], but there remains a number of technical problems to be investigated.

Acknowledgements We would like to express our gratitude for Andrei V. Klimov and Yuri A. Klimov for their comments and fruitful discussions. Special thanks are due to Sergei M. Abramov for his encouragement and support for the project.

References

1. D. A. Basin and T. Walsh. Difference matching. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 295–309, London, UK, 1992. Springer-Verlag.
2. R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *J. ACM*, 22(1):129–144, 1975.
3. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artif. Intell.*, 62(2):185–253, 1993.
4. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.
5. A. P. Ershov. On the essence of compilation. In E. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.
6. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
7. R. Glück. Is there a fourth Futamura projection? In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 51–60, New York, NY, USA, 2009. ACM.
8. 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.
9. G. W. Hamilton. Extracting the essence of distillation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 151–164, 2010.
10. G.W. Hamilton and M.H. Kabir. Constructing programs from metasystem transition proofs. In *Proceedings of the First International Workshop on Metacomputation in Russia*, pages 9–26, Pereslavl-Zalessky, Russia, July 2008. Ailamazyan University of Pereslavl.
11. P. A. Jonsson. Positive supercompilation for a higher-order call-by-value language. Master's thesis, Luleå University of Technology, 2008.
12. And. V. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 185–192, 2010.
13. I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, 2009.
14. I. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. Preprint 21, Keldysh Institute of Applied Mathematics, 2010.
15. 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.
16. A. P. Lisitsa and A. P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Program. Comput. Softw.*, 33(1):14–23, 2007.
17. A.P. Lisitsa and M. Webster. Supercompilation for equivalence testing in metamorphic computer viruses detection. In *Proceedings of the First International Workshop on Metacomputation in Russia*, pages 113–118. Ailamazyan University of Pereslavl, 2008.
18. N. Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, 2008.
19. N. Mitchell and C. Runciman. A supercompiler for core Haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *LNCS*, pages 147–164, Berlin, Heidelberg, 2008. Springer-Verlag.

20. D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1-2):193–233, 1996.
21. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.
22. M. H. Sørensen. Convergence of program transformers in the metric space of trees. *Sci. Comput. Program.*, 37(1-3):163–205, 2000.
23. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Logic Programming: The 1995 International Symposium*, pages 465–479, 1995.
24. M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 485–500, London, UK, 1994. Springer-Verlag.
25. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
26. V. F. Turchin. *The phenomen of science. A cybernetic approach to human evolution.* Columbia University Press, New York, 1977.
27. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.
28. V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In *Partial Evaluation*, volume 1110 of *LNCS*, pages 481–509. Springer, 1996.
29. T. Walsh. A divergence critic for inductive proof. *J. Artif. Int. Res.*, 4(1):209–235, 1996.