Keldysh Institute • Publication search

Keldysh Institute preprints • Preprint No. 81, 2010

Klyuchnikov I.G.

Towards Effective Two-Level Supercompilation

Recommended form of bibliographic references: Klyuchnikov I.G. Towards Effective Two-Level Supercompilation. Keldysh Institute preprints, 2010, No. 81, 28 p. URL: http://library.keldysh.ru/preprint.asp?id=2010-81&lg=e

KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Ilya Klyuchnikov

Towards effective two-level supercompilation

Moscow
2010

Ilya Klyuchnikov. Towards effective two-level supercompilation The paper presents a number of improvements to the method of two-level supercompilation: a fast technique of lemma discovering by analyzing the expressions in the partial process tree, an enhancement to the algorithm of checking improvement lemmas based on the normalization of tick annotations, and a few techniques of finding simplified versions of lemmas discovered in the process of two-level supercompilation.

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

Илья Ключников. На пути к быстрой двухуровневой суперкомпиляции Представлены некоторые усовершенствования к методу двухуровневой суперкомпиляции: быстрый метод поиска лемм основанный на анализе выражений в узлах частичного дерева процесса, улучшенный алгоритм распознавания улучшающих лемм, основанный на нормализации аннотаций ("тиков"), а также некоторые методы упрощения лемм, обнаруженных в процессе двухуровневой суперкомпиляции.

Работа выполнена при поддержке проектов РФФИ № 08-07-00280-а и № 09-01-00834-а.

Contents

1 Introduction **3**

2 Transformation relation $HOSC^2$ **4**

3 Search for lemmas **7**

4 Normalization of ticks **10**

5 Multi-result supercompilation **14**

6 Extraction of more general lemmas **17**
- 6.1 Removing the common context 17
- 6.2 Generalization by abstracting common subexpressions 19
- 6.3 On the power of homeomorphic embedding for lemma extraction . 20

7 Simplification of lemma syntax **20**

8 Parameterized two-level supercompiler **21**

9 Conclusion **23**
- 9.1 Goals and methods of two-level supercompilation 23
- 9.2 Related work and further development 24

References **26**

1 Introduction

A new method of multi-level supercompilation was suggested in [15]. It has been shown that (1) multi-level supercompilation is superior to the classical (single-level) supercompilation and (2) a practical implementation of multi-level supercompiler is possible in principle.

However, little attention was paid to the efficiency of two-level supercompilation. In the experimental "proof-of-concept" two-level supercompiler, an improvement lemma (e, e') for a given expression e was generated by "blind search": just by enumerating *all* e' not exceeding e in size, followed by checking (e, e') for being an improvement lemma.

The purpose of this work is, first, to present some techniques of speeding up the search for improvement lemmas by narrowing the search space and, second, to increase the number of improvement lemmas provable by supercompilation. We consider the following ideas:

1. The detection of lemmas should be based on the analysis of the structure of expressions appearing in the nodes of a partial process tree, rather than on a brute force search.

2. The number of lemmas recognizable by supercompilation can be increased by normalizing annotations ("ticks") in residual expressions.

3. The "upper" and "lower" supercompilers may differ not only in how they behave when the whistle blows, but also in other details of their behavior, which enables more lemmas to be recognized.

4. The lemmas discovered by a two-level supercompiler often happen to be rather "specific": cumbersome and only applicable in rare cases. However, we suggest some techniques of extracting more abstract, more reusable and more "human-friendly" lemmas from the "specific" ones. We consider two approaches:

   (a) Removing the common context.

   (b) Generalization by abstracting common subexpressions.

In this work we consider only the case of two-level supercompilation. Thus, for the sake of brevity, we use the terms "lower" and "upper" supercompilers, when referring to the "first-level" and "second-level" supercompilers.

We use the same examples as in [15] and show that the lemmas that had been automatically discovered in [15] by blind search now can be found out more efficiently by analyzing expressions appearing in the partial process tree.

We give an additional example demonstrating that two-level supercompilation is capable of achieving asymptotic runtime improvements (namely, super linear speed-ups). Thus, despite the fact that the supercompiler HOSC was designed as a program analysis tool, it may also be useful for program optimization.

The rest of the paper is organized as follows. We give a number of examples showing that lemmas can be detected in a fast and effective way by analyzing paths in the partial process tree. Section 2 presents two-level supercompilation as a transformation relation and extends the algorithm of constructing the residual program from a partial process tree so that it now inserts tick annotations into residual programs. Section 3 suggests a simple technique capable of discovering useful improvement lemmas by analyzing the partial process tree, rather than by blind search. Section 4 presents a technique, based on reducing tick annotations to a normal form, which increases the number of lemmas recognizable by supercompilation. Section 5 shows that the results produced by two-level supercompilation may be improved by using different options for the upper and lower supercompilers, which may (and should) differ in their behavior. Moreover, a two-level supercompiler may automatically try various combinations of options for the upper and lower supercompilers, to produce several (equivalent, yet different) residual programs, thereby implementing a more general idea of *multi-result supercompilation*. Some lemmas automatically discovered by a two-level supercompiler happen to be too "specific" and cumbersome. Section 6 describes two techniques of extracting more general and abstract lemmas from "specific" ones. Section 7 shows that, in some cases, lemmas discovered by a two-level supercompiler can be simplified by term rewriting guided by an analysis of the partial process tree. Section 8 gives a "recipe" for remaking an ordinary supercompiler into a two-level one, presenting a parameterized two-level supercompilation algorithm.

Finally, in Section 9, we explain the rationale behind two-level supercompilation, its goals and purposes, and consider related works and possible lines of further research.

The residual programs and lemmas presented in the paper have been produced by a two-level supercompiler HLSC constructed by modifying the supercompiler HOSC 1.5 described in [10, 12], whose behavior may be controlled by a number of options. For the purposes of presentation, we have manually chosen the combinations of options that result in producing interesting residual programs of reasonable size. However, in the case of *multi-result supercompilation*, in order to achieve best results, a two-level supercompiler should automatically try various combinations of options.

2 Transformation relation $HOSC^2$

Multi-level supercompilation was described in [15] a bit informally – as an extension of some existing supercompiler. In this section we will consider two-level supercompilation in more abstract terms – as a transformation relation (Fig. 1). The base (single-level) supercompiler is a supercompiler implementing the transformation relation $HOSC$ [13]. In order to distinguish supercompilers belonging to different levels, we will designate the level of a supercompiler by a superscript:

Figure 1 Transformation relations

(a) *$HOSC^1$*

```
1 t = [e0]
2 while incomplete(t) do
3 |  β = unprocessedLeaf(t)
4 |  t = choice{drive*(t, β), generalize(t, β), fold(t, β)}
5 end
```

(b) *$HOSC^2$*

```
1 t = [e0]
2 while incomplete(t) do
3 |  β = unprocessedLeaf(t)
4 |  t = choice{drive*(t, β), generalize(t, β), fold(t, β), applyLemma(t, β)}
5 end
```

(c) auxiliary operations

$applyLemma(t,\ \beta)$	$replace(t,\ \beta,\ e')$, where $(\beta.expr, e)$ – an improvement lemma, detected by means of *some* transformation $HOSC^1$.

$HOSC^1$, $HOSC^2$. The transformation relation $HOSC^1$ specifies a nondeterministic algorithm of constructing a partial process tree (Fig. 1a, the details and the proof of correctness of the transformation relation $HOSC^1$ can be found in [13]).

The correctness of the transformation relation means that a residual program is equivalent to an original one. The equivalence is understood as follows:

Definition 1 (Equivalence)**.** Two HLL-expressions e_1 and e_2 are operationally equivalent, $e_1 \cong e_2$, if for all contexts C, such that $C[e_1]$ and $C[e_2]$ are closed, either $C[e_1]$ and $C[e_2]$ both converge or both diverge.

The parameterized supercompiler SC^1_{ijk}, which is guaranteed to terminate and satisfies the transformation relation $HOSC^1$, was described in [12].

The paper [14] suggested a way of proving the equivalence of expressions via normalization by supercompilation.

The goal of multi-level supercompilation is to construct a perfect partial process tree (a tree built without a generalization of configurations). The main idea is “to silence the whistle” by using lemmas: a “bad” configuration (activating the whistle) is replaced by a “good” one (silencing the whistle). In some cases it results in building a perfect partial process tree or, at least, to increasing the depth of transformation.

In order to guarantee the correctness of two-level supercompilation, we require all the lemmas to be *improvement lemmas* [17].

Figure 2 Generation of a residual program annotated with ticks

$$
\begin{array}{lrr}
\mathcal{C}\,[\![\alpha]\!]_{t,\Sigma} & & \\
\quad \Rightarrow letrec\ f' = \checkmark_{[\![\alpha.expr]\!]}(\lambda\overline{v_i} \to (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma'})\theta') & \text{if } \lceil \alpha \curlywedge t \rceil \neq \bullet & (C_1) \\
\qquad in\ f'\overline{v_i'} & & \\
\qquad\quad \text{where} & & \\
\qquad\qquad \overline{[\beta_i]} = \lceil \alpha \curlywedge t \rceil,\ \theta_i = \alpha.expr \otimes \beta_i.expr, & & \\
\qquad\qquad \overline{v_i'} = domain(\bigcup \overline{\theta_i}),\ \theta' = \{\overline{v_i' := v_i}\}, & & \\
\qquad\qquad \Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i}),\ f' \text{ and } \overline{v_i} \text{ are fresh} & & \\
\quad \Rightarrow f'_{sig}\theta & \text{if } \lceil \alpha \upuparrows t \rceil \neq \bullet & (C_2) \\
\qquad\quad \text{where} & & \\
\qquad\qquad f'_{sig} = \Sigma(\lceil \alpha \upuparrows t \rceil),\ \theta = \lceil \alpha \upuparrows t \rceil.expr \otimes \alpha.expr & & \\
\quad \Rightarrow\ \checkmark_{[\![\alpha.expr]\!]}(\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma}) & \text{otherwise} & (C_3)
\end{array}
$$

$$
\begin{array}{lr}
\mathcal{C}'[\![let\ \overline{v_i = e_i};\ in\ e]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma}}\} & (C_1') \\
\mathcal{C}'[\![v\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} & (C_2') \\
\mathcal{C}'[\![c\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} & (C_3') \\
\mathcal{C}'[\![\lambda v_0 \to e_0]\!]_{t,\alpha,\Sigma} \Rightarrow \lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C_4') \\
\mathcal{C}'[\![con\langle case\ v\ \overline{e_j'}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} & \\
\quad \Rightarrow case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'} of \{\overline{p_i \to \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma'};}\} & (C_5') \\
\mathcal{C}'[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C_6') \\
\mathcal{C}'[\![con\langle case\ c\ \overline{e_j'}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C_7') \\
\mathcal{C}'[\![con\langle f\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C_8')
\end{array}
$$

$$
\begin{array}{lr}
\checkmark_{[\![con\langle f\rangle]\!]}(e) = \checkmark e & (T_1) \\
\checkmark_{[\![e']\!]}(e) = e & (T_2)
\end{array}
$$

Definition 2 (Improvement lemma)**.** An ordered pair of operationally equivalent expressions (e_1, e_2) is an improvement lemma, $e_1 \trianglerighteq_s e_2$, if for all contexts C, such that $C[e_1]$ and $C[e_2]$ are closed, if the computation of $C[e_1]$ terminates using n function calls, then the computation of $C[e_2]$ also terminates, but uses no more than n function calls.

It should be noted that in [17] Sands only gives a *definition* of an improvement lemma, but provides no clue how to *mechanically check* a pair of expressions for being an improvement lemma. In [15] we suggested a method based on supercompilation which, in many cases, succeeds in automatically detecting improvement lemmas of practical interest.

The method is based on a small modification of the algorithm generating the residual program $prog'$ from a partial process tree t. The residual program is built according to the rules in Fig. 2:

$$prog' = \mathcal{C}\ [\![t.root]\!]_{t,\{\}}$$

When processing a configuration in the form $con\langle f\rangle$, ticks (annotations ✓) are

Figure 3 Algorithm of improvement lemma detection

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

Figure 4 `or (even x) (odd x)`: input program

```
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; };
```

inserted in the residual program *prog'*, showing that in this particular place in the *original* program there was a function call.

The supercompiler thus modified is used as follows. Suppose, (e_1, e_2) has to be checked for being an improvement lemma. Then e_1 and e_2 are supercompiled and corresponding residual programs e'_1 and e'_2 are compared. If e'_1 are e'_2 the same (modulo tick annotations and alpha renaming), we conclude that e_1 and e_2 are operationally equivalent and then perform an additional test. The meaning of the second test is to check that each subexpression of e'_1 is annotated by no less number of ticks than the corresponding subexpression of e'_2. If this is the case, we conclude that e_2 is an improvement of e_1.

This method of detecting improvement lemmas is the basis of the nondeterministic two-level supercompilation algorithm $HOSC^2$ shown in Fig. 1b. This algorithm, in some cases, choses an arbitrary expression e and checks, whether $(\beta.expr, e)$ is an improvement lemma. If this check succeeds, then the lemma thus obtained is applied.

However, from a practical standpoint, we need a *deterministic* algorithm, discovering improvement lemmas in a systematic way, rather than by "inspiration" or blind search.

3 Search for lemmas

Let us consider the transformation of the program in Fig. 4 by the supercompiler SC^2_{+--}. After a few steps an embedding of configurations is detected:

$$e_1 \trianglelefteq^*_c e_2,$$

Figure 5 $e_1 \trianglelefteq^*_c e_2$

(a) e_1

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

(b) e_2

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

where expressions e_1 and e_2 are shown in Fig. 5a and Fig. 5b, correspondingly.

Let t be a partial process tree. We will write $e \in t$ if there is a node in t labelled by the expression e.

We may try to find an improvement lemma by blind search, but this approach is not effective. Besides, a lemma thus found may turn out to be "alien" to the original program, which may result in constructing a partial process tree that looks "unnatural" and "obscure" from the human viewpoint. However, the expressions labelling nodes of partial process tree may be used as the source of syntactic material for the search for lemmas!

Let us try to find an expression $e' \in t$, such that:

$$SC^1_{+--}[\![e_2]\!] \equiv SC^1_{+--}[\![e']\!] \qquad SC^1_{+--}[\![e_2]\!] \trianglerighteq^{\checkmark}_{\sim} SC^1_{+--}[\![e']\!]$$

If we succeed, the correctness of the replacement of e_2 with e' is guaranteed. Also it seems reasonable to try first those expressions $e' \in t$, that have something in common with the expression e_2. But, if the expression e_1 is embedded into e_2, it certainly means that e_1 are e_2 syntactically similar, and e_1 is smaller in size than e_2. So, why not to check whether the pair (e_2, e_1) is an improvement lemma?

This trick seems to be quite natural, but, unfortunately, in this simple form, it is only applicable in rare cases. However, there may be used the following small improvement.

Let us find an expression $e' \in t$, such that $SC^1_{+--}[\![e_2]\!]$ and $SC^1_{+--}[\![e']\!]$ are the same, modulo a variable renaming[1]:

$$SC^1_{+--}[\![e_2]\!] \equiv SC^1_{+--}[\![e']\!]\{\overline{v_i := v'_i}\}$$

It follows that

$$SC^1_{+--}[\![e_2]\!] \equiv SC^1_{+--}[\![e'\{\overline{v_i := v'_i}\}]\!]$$

So, if

$$SC^1_{+--}[\![e_2]\!] \equiv SC^1_{+--}[\![e']\!]\{\overline{v_i := v'_i}\} \qquad SC^1_{+--}[\![e_2]\!] \trianglerighteq^{\checkmark}_{\sim} SC^1_{+--}[\![e'\{\overline{v_i := v'_i}\}]\!],$$

then an upper supercompiler can correctly replace e_2 by $e'\{\overline{v_i := v'_i}\}$ when constructing a partial process tree. *The leading candidate for a lemma will be an expression embedded into the current configuration.*

[1] All v'_i are different

Figure 6 Supercompiled expressions

(a) $SC^1_{+--}[\![e_1]\!]$

```
letrec f = ✓λx →
  case (x) of {
    S v32 → ✓case v32 of {
      S v35 → f v35;
      Z → letrec g = ✓λy →
             case y of {
                S v38 → ✓case v38 of { S v41 → g v41; Z → True; };
                Z → False;
             }
          in g m;
     };
  Z → True;
}
in f m
```

(b) $SC^1_{+--}[\![e_2]\!]$

```
letrec f = ✓λx →
  case (x) of {
    S v64 → ✓case v64 of {
      S v67 → f v67;
      Z → ✓✓letrec g = ✓λy →
                case y of {
                  S v76 → ✓case v76 of { S v79 → g v79; Z → True; };
                  Z → False;
                }
              in g n;
      };
    Z → True;
}
in f n
```

Let us come back to the example discussed for far (and to the concrete e_1 and e_2). The corresponding residual expressions $e_1' = SC^1_{+--}[\![e_1]\!]$ and $e_2' = SC^1_{+--}[\![e_2]\!]$ are shown in Fig. 6a and 6b. In this case the supercompiled expressions are renaming of each other:

$$SC^1_{+--}[\![e_2]\!] \equiv SC^1_{+--}[\![e_1]\!]\{m := n\}$$

Hence, it is shown that $(e_2, e_1\{m := n\})$ is an improvement lemma. So, it means that e_2 can be replaced by $e_1\{m := n\}$. Then there is an immediate possibility for folding.

The result of two-level supercompilation is shown in Fig. 7.

Figure 7 `or (even m) (odd m)`: the result of two-level supercompilation

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

Figure 8 `even (dAcc m Z)`: input program

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

even (dAcc m Z) where

even = λx → case x of { Z → True; S x1 → odd x1; };
odd = λx → case x of { Z → False; S x1 → even x1; };
dAcc = λx y → case x of { Z → y; S x1 → dAcc x1 (S (S y)); };
```

Figure 9 $e_3 \trianglelefteq_c^* e_4$

(a) e_3

```
case dAcc m Z of { S z → odd z; Z → True; }
```

(b) e_4

```
case dAcc n (S (S Z)) of { S z → odd z; Z → True; }
```

Despite its simplicity, the technique described in this section enables us to construct a two-level supercompiler, which, on the one hand, is quite efficient (in terms of performance), and, on the other hand, is capable of performing deeper transformations, than a one-level supercompiler.

4 Normalization of ticks

Let us consider another example from [15]. The source program is shown in Fig. 8. Again, we use the supercompiler SC^2_{+--}.

After a few steps, an embedding of configurations is detected:

$$e_3 \trianglelefteq_c^* e_4,$$

where expressions e_3 and e_4 are represented in Fig. 9a and Fig. 9b, correspondingly.

Figure 10 Supercompiled expressions

(a) $SC^1_{+--}[\![e_3]\!]$

```
letrec f = ✓λx y →
  case x of {
    S v33 → f v33 (S (S y));
    Z → letrec g = λz →
            case z of {
              S v36 → ✓case v36 of { S v39 → ✓(g v39); Z → False; };
              Z → True;
            }
          in g y;
   }
in f m Z
```

(b) $SC^1_{+--}[\![e_4]\!]$

```
letrec f = ✓λx y →
  case x of {
    S v70 → f v70 (S (S y));
    Z → ✓letrec g = ✓λz →
             case z of {
               S v79 → ✓case v79 of { S v82 → (g v82); Z → False; };
               Z → True;
             }
           in g y;
  }
in f n Z
```

Let us try to apply the technique from the previous section in this situation. The corresponding residual expressions $SC^1_{+--}[\![e_3]\!]$ and $SC^1_{+--}[\![e_4]\!]$ are shown in Fig. 10a and 10b. Again, the supercompiled expressions are renamings of each other:

$$SC^1_{+--}[\![e_4]\!] \equiv SC^1_{+--}[\![e_3]\!]\{m := n\}$$

So, we conclude that $e_3\{m := n\} \cong e_4$. Unfortunately, checking tick annotations of residual programs does not enable us to conclude that $(e_4, e_3\{m := n\})$ is an improvement lemma, since the following does not hold:

$$SC^1_{+--}[\![e_4]\!] \trianglerighteq^{\checkmark}_{\sim} SC^1_{+--}[\![e_3\{m := n\}]\!]$$

However, as can be proven by hand, $(e_4, e_3\{m := n\})$ *is* an improvement lemma.

The problem is that tick annotations are, in a sense, "too exact". What really matters for the detection of improvement lemmas is the number of function calls performed during evaluation, rather than the moments in which they take place. The positions of tick annotations, however, do contain some information irrelevant

Figure 11 Normalization of ticks

$$
\begin{array}{lll}
\checkmark case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} & \Rightarrow case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to \checkmark e_i;}\} & (N_1) \\
case\ \checkmark e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} & \Rightarrow case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to \checkmark e_i;}\} & (N_2) \\
(\checkmark \lambda x_1 \dots x_n \to e)\ e_1 \dots e_n & \Rightarrow (\lambda x_1 \dots x_n \to \checkmark e)\ e_1 \dots e_n & (N_3) \\
letrec\ f = \checkmark \lambda x_1 \dots x_n \to e_0\ in\ e_1 & \Rightarrow letrec\ f = \lambda x_1 \dots x_n \to \checkmark e_0\ in\ e_1 & (N_4)
\end{array}
$$

The rule N_4 is applicable only if all occurrences of f in expressions e_0 and e_1 are applications with at least n arguments

to the estimation of the evaluation cost, so that two cost-equivalent expressions can be annotated in different ways.

Fortunately, tick annotations can often be moved around the expression without violating the correctness of tick counting. Hence, in some cases, two annotated expressions may be transformed to the form in which they become comparable (in terms of improvement).

The key idea is that the exact number of ticks matters only in cases where the evaluation terminates. If the evaluation of an expression does not terminate, the number of ticks is assumed to be "infinite", so that adding to it a finite number of ticks does not change the grand total.

Consider an arbitrary `case`-expressions annotated with a tick:

$$\checkmark case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$$

It turns out that this tick can be pushed inside the expression by placing a tick at the start of each branch:

$$case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to \checkmark e_i;}\}$$

The correctness of this transformation is justified as follows. Suppose the evaluation of the selector e_0 terminates and costs n ticks. Then, before evaluating a branch, we have to take into account $1 + n$ ticks for the original `case`-expression, and $n + 1$ ticks for the transformed one. Since $1 + n = n + 1$, the grand total of ticks in the both cases is the same. Now suppose that the evaluation of e_0 does not terminate. In this case no branch is evaluated, so that we have $1+\infty$ ticks for the original expression and ∞ ticks for the transformed expression. But again, the grand total of ticks is the same, because $1 + \infty = \infty$.

In [17] David Sands gives a number of "tick laws". But, in order to construct a tick normalization algorithm, we need a set of rewriting rules, rather than an equivalence relation. A set of such rewrite rules is presented in Fig. 11. The first three rules correspond to laws described by Sands in [17]. The last rule (the most interesting one) corresponds to a new law.

It can be shown that the tick normalization procedure based on these rules always terminates, the result not depending on the order in which the rules are applied.

Figure 12 Result of ticks normalization

(a) $\mathcal{N}_{\checkmark}(SC^1_{+--}[\![e_3]\!])$

```
letrec f = λx y →
  case x of {
    S v33 → ✓(f v33 (S (S y)));
    Z → ✓letrec g = λz →
            case z of {
              S v36 → case v36 of { S v39 → ✓✓(g v39); Z → ✓False; };
              Z → (True);
            }
          in g y;
   }
in f m Z
```

(b) $\mathcal{N}_{\checkmark}(SC^1_{+--}[\![e_4]\!])$

```
letrec f = λx y →
  case x of {
    S v70 → ✓f v70 (S (S y));
    Z → ✓✓letrec g = λz →
             case z of {
               S v79 → case v79 of { S v82 → ✓✓(g v82); Z → ✓✓False; };
               Z → ✓True;
             }
           in g y;
  }
in f n Z
```

Figure 13 `even (double m Z)`: the result of two-level supercompilation

```
letrec f = λx → case x of { S y → f y; Z → True; } in f m
```

Normalizing ticks in the expressions in Fig. 10a and 10b, we obtain the expressions shown in Fig. 12a and 12b, correspondingly. Now we can conclude that $e_3\{m := n\}$ is an improvement of e_4.

The result of two-level supercompilation is shown in Fig. 13.

The tick normalization technique has something in common with the idea of normalizing some intermediate results of supercompilation, for example, with the syntactic normalization of expressions performed by Neil Mitchell [16].

Figure 14 `match doubleA word`: a source program

```
data Symbol = A;
data List a = Nil | Cons a (List a);
data Maybe a = Just a | Nothing;

match doubleA word where

doubleA = alt nil (seq a (seq doubleA a));
match = λp i → p (eof return) i;
return = λx → Just x;
nil = λnext w → next w;
seq = λp1 p2 next w → p1 (p2 next) w;
alt = λp1 p2 next w → case p1 next w of {
    Just w1 → Just w1;
    Nothing → p2 next w;
};
eof = λnext w → case w of {
    Cons s w1 → Nothing;
    Nil → next Nil;
};
a = λnext w → case w of {
    Cons s w1 → case s of { A → next w1; };
    Nil → Nothing;
};
```

5 Multi-result supercompilation

Consider the program shown in Fig. 14, containing definitions for a set of basic combinators meant for constructing parsers. The implementation of the combinators is based on backtracking. The basic parsers are used in the program to define an additional parser, `doubleA`, corresponding to the following context-free grammar:

```
doubleA = ϵ | A doubleA A
```

The target expression in the program is an application of the parser `doubleA` to an arbitrary string `word`.

It is clear that the parser coded in this way, when applied to a string s, has a complexity $O(|s|^2)$.

Let us try to transform this parser by the two-level supercompiler SC^2_{+--} using the techniques described in the previous sections.

After a few (about 30) steps an embedding is detected:

```
doubleA (eof return) word ⊴*c doubleA (a (eof return)) v33
```

Figure 15 $e_5 \trianglelefteq_c^* e_6$

(a) e_5

```
case
  case word of {
    Cons v32 v33 → Nothing;
    Nil → return Nil;
  } of {
  Nothing → seq a (seq doubleA a) (eof return) word;
  Just v34 → Just v34;
}
```

(b) e_6

```
case
  case v97 of {
    Cons v149 v150 → Nothing;
    Nil → return Nil;
  } of {
  Nothing → seq a (seq doubleA a) (a (eof return)) (Cons A v97);
  Just v151 → (Just v151);
}
```

However, an attempt to silence the whistle fails: the supercompiled expression are not a renaming of each other.

So, we can try to use another version of the supercompiler: the supercompiler SC^2_{++-}, which divides all nodes into global and local ones. After a few steps, an embedding is detected:

$$e_5 \trianglelefteq_c^* e_6,$$

where the expressions e_5 and e_6 are shown in Fig. 15a and Fig. 15b.

Unfortunately, an attempt to silence the whistle using the supercompiler SC^1_{++-} also fails, as the supercompiled expressions are not a renaming of each other:

$$SC^1_{++-}[\![e_5]\!]\theta \not\equiv SC^1_{++-}[\![e_6]\!]$$

(Note that the upper and lower supercompilers are assumed to differ in one detail: in how they handle the embedding of configurations, but we might have tried other options.)

However, nothing prevents us from using a different lower supercompiler. Let us try the supercompiler SC^1_{+--} (without the division of nodes into global and local ones). The result is that:

$$SC^1_{+--}[\![e_5]\!]\{word := v97\} \equiv SC^1_{+--}[\![e_6]\!]$$

Moreover, by examining the tick annotations, the upper supercompiler is able to prove that $(e_6, e_5\{word := v97\})$ is an improvement lemma. Thus, it replaces

Figure 16 `match doubleA word`: the result of two-level supercompilation

```
letrec f = λx →
    case x of {
      Cons v32 v33 →
        case v32 of {
          A → case v33 of {
                  Cons v96 v97 → case v96 of { A → (f v97); };
                  Nil → Nothing;
                };
          };
      Nil → Just Nil;
    }
in f word
```

the configuration e_6 with $e_5\{word := v97\}$, and performs a fold.

The result of two-level supercompilation is shown in Fig. 16.

As was noted above, the source program corresponds to the following context-free grammar:

```
doubleA = ϵ | A doubleA A
```

and the application of this parser to a string s has complexity $O(|s|^2)$.

The result of two-level supercompilation corresponds to the following context-free grammar:

```
doubleA = ϵ | A A doubleA
```

Note that now the application of the second parser to a string s has complexity $O(|s|)$.

This example demonstrates that two-level supercompilation is capable of achieving asymptotic runtime improvements of programs (written in a lazy functional language), while single-level supercompilation can only achieve linear speedups [18].

Another conclusion to be drawn from the above example is that the results of two-level supercompilation can be improved by selecting different combinations of options for the upper and lower supercompilers. This may be regarded as a particular example of *multi-result supercompilation*, which amounts to producing a set of residual programs, rather than a single program.

An interesting observation is that the best results are often produced by the combination of an upper supercompiler, dividing nodes into global and local ones, and a lower supercompiler, not doing so. For example, this combination is able to produce the results described in sections 3 and 4.

Figure 17 Improvement lemma (e_8, e_7)

(a) e_7

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

(b) e_8

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

6 Extraction of more general lemmas

Sometimes, the lemmas discovered by the techniques presented in the previous sections turn out to be cumbersome and difficult to understand (from the human viewpoint). However, as will be shown below, the improvement lemmas discovered by a two-level supercompiler may be used as the source material for extracting more general lemmas that are more abstract and good-looking.

Definition 3 (Consequence of a lemma)**.** Let (e_1, e_2) and (e'_1, e'_2) be two improvement lemmas, such that:

$$e_1 \trianglerighteq_s e_2 \Rightarrow e'_1 \trianglerighteq_s e'_2 \qquad e'_1 \trianglerighteq_s e'_2 \not\Rightarrow e_1 \trianglerighteq_s e_2$$

Then the lemma (e'_1, e'_2) is said to be a consequence of the lemma (e_1, e_2), and the lemma (e_1, e_2) is said to be more general than the lemma (e'_1, e'_2).

A lemma discovered by a two-level supercompiler often happens to be a consequence of a more general lemma.

6.1 Removing the common context

Let us consider an improvement lemma (e_8, e_7) (consisting of expressions shown in Fig. 17), which is discovered by a two-level supercompilation of the program in Fig. 4.

The meaning of this lemma is not obvious, since the expressions e_7 and e_8 are rather cumbersome. However, e_7 and e_8 are almost identical, differing only in the subexpressions highlighted in Fig. 17. Hence, there comes the following idea: let us try to prove the equivalence of the subexpressions! We are lucky: the pair

```
(odd (S (S n)), odd n)
```

really turns out to be an improvement lemma. So the "big" lemma in Fig. 17 is just a consequence of this lemma.

The above example is an instance of an implication of the following form:

$$e' \trianglerighteq_s e'' \Rightarrow e\{v := e'\} \trianglerighteq_s e\{v := e''\}$$

Figure 18 Improvement lemma (e_9, e_{10})

(a) e_9

```
case
  case v97 of {
    Cons v32 v33 → Nothing;
    Nil → return Nil;
  } of {
  Nothing → seq a (seq doubleA a) (eof return) v97 ;
  Just v34 → Just v34;
}
```

(b) e_{10}

```
case
  case v97 of {
    Cons v149 v150 → Nothing;
    Nil → return Nil;
  } of {
  Nothing → seq a (seq doubleA a) (a (eof return)) (Cons A v97) ;
  Just v151 → (Just v151);
}
```

How should the expressions e' and e'' be selected? Since we are interested in finding a lemma (e', e'') of a small size, it seems reasonable to try expressions e' and e'' of minimal (possible) size. However, the smallest e' and e'' usually do not form an improvement lemma! For example, the following pair

```
(S (S n), n)
```

consists of minimal differing subexpressions of e_7 and e_8, nevertheless, it *is not* an improvement lemma (just because the expressions `n` and `S (S n)` are not even equivalent). So, when searching for a lemma, we have to also try subexpressions e' and e'' that are small, but not the smallest ones.

In order to formalize the idea in general terms, we have to define what is a generalization of two expressions.

Definition 4 (Generalization)**.** A generalization of two expressions e_1 and e_2 is a triple $(e_g, \theta_1, \theta_2)$, where e_g is an expression and θ_1 and θ_2 are substitutions, such that $e_g\theta_1 \equiv e_1$ and $e_g\theta_2 \equiv e_2$. A set of all generalizations of expressions e_1 and e_2 is denoted as $e_1 \frown e_2$.

Let (e', e'') be an improvement lemma discovered by a two-level supercompiler. Then we may consider generalizations of the form $(e_g, \{\overline{v_i := e'_i}\}, \{\overline{v_i := e''_i}\})$ belonging the set $e' \frown e''$ and check the pairs (e'_i, e''_i) for being improvement lemmas.

Figure 19 Improvement lemma (e_{12}, e_{11})

(a) e_{11}

```
case dAcc m Z of { S z → odd z; Z → True; }
```

(b) e_{12}

```
case dAcc n (S (S Z)) of { S z → odd z; Z → True; }
```

For example, when transforming the program shown in Fig. 14, a two-level supercompiler discovers an improvement lemma shown in Fig. 18. The lemma seems to be quite cumbersome, but there is a smaller improvement lemma formed by the highlighted subexpressions:

```
(seq a (seq doubleA a) (a (eof return)) (Cons A v97),
  seq a (seq doubleA a) (eof return) v97)
```

6.2 Generalization by abstracting common subexpressions

The form of implication considered in this section is:

$$e_1 \mathrel{\trianglerighteq\!\!\!\raise-0.6ex\hbox{$\sim$}}_s e_2 \Rightarrow e_1\{v := e\} \mathrel{\trianglerighteq\!\!\!\raise-0.6ex\hbox{$\sim$}}_s e_2\{v := e\}$$

When transforming the program in Fig. 8, a two-level supercompiler discovers the lemma shown in Fig. 19. An attempt to extract a more general lemma by removing the common context is unsuccessful. However, abstracting the identical highlighted subexpressions leads to the following lemma:

```
(case dAcc m (S (S x)) of { S z → odd z; Z → True; },
  case dAcc m x of { S z → odd z; Z → True; })
```

Note that this lemma is not smaller in size than the original one (since the common subexpression is just a nullary constant `Z`). But it still may be useful, because it is applicable in more cases than the original one.

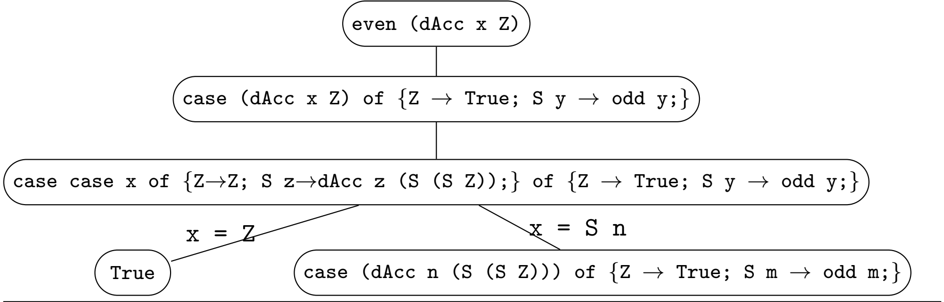
To formalize this idea we use the notion of "inverse generalization".

Definition 5 (Inverse generalization)**.** An inverse generalization of expressions e' and e'' is a triple (e'_g, e''_g, θ), where e'_g and e''_g are expressions, and θ a substitution, such that $e'_g\theta \equiv e'$ and $e''_g\theta \equiv e''$. The set of all inverse generalizations of expressions e' and e'' is denoted as $e' \smile e''$.

Let (e', e'') be an improvement lemma discovered by a two-level supercompiler. Then we may consider the inverse generalizations (e'_g, e''_g, θ) belonging to the set $e' \frown e''$ and try to find a more general improvement lemma among the pairs (e'_g, e''_g).

As regards the improvement lemma in Fig. 19, the inverse generalization corresponding to the highlighted expressions leads to a more general lemma considered above.

Figure 20 `even (double x Z)`: partial process tree



6.3 On the power of homeomorphic embedding for lemma extraction

Why the whistle based on the refined homeomorphic embedding [12], when used in a two-level supercompiler, produces reasonable results? It seems to be due to the fact that the refined embedding captures the notion of generalization, as well as the notion of inverse generalization.

On the other hand, the whistle based on the adaptation of the pure homeomorphic embedding [12] seems to be rather inadequate for the use in a two-level supercompiler. It blows too often, and does not ensure the existence of a meaningful generalization for embedded expressions. In particular, it produces unsatisfactory results for the examples considered in this paper. At the same time, the use of the refined homeomorphic embedding drastically improves the behavior of a two-level supercompiler.

7 Simplification of lemma syntax

A supercompiler may be used as a part of another tool (for example, a theorem prover), in which case it should produce proofs readable for humans.

Let us consider the reduction rules, together with the function definitions appearing in a program, as a term rewrite system. Let $\overset{*}{\longleftrightarrow}$ denote equality by rewriting (closed under context and substitution) [1].

Definition 6 (Lite version of lemma)**.** A lemma (e'_1, e'_2) is a lite version of a lemma (e_1, e_2), if corresponding expressions are equal by rewriting and are of a smaller size: $e_1 \overset{*}{\longleftrightarrow} e'_1$, $e_2 \overset{*}{\longleftrightarrow} e'_2$, $|e'_1| \leq |e_1|$, $|e'_2| \leq |e_2|$.

It is obvious that a lite lemma (due to its smaller size) in many cases will be more readable that the corresponding "technical" lemma.

Consider, for example, the improvement lemma shown in Fig. 19 and the corresponding partial process tree (Fig. 20). This lemma is automatically discovered

when transforming the expression `even (double x Z)` by a two-level supercompiler.

A lite version of this improvement lemma is not difficult to find. Indeed:

```
even (dAcc m Z) →
  case dAcc m Z of { S z → odd z; Z → True; }

even (dAcc m (S (S Z))) →
  case dAcc m (S (S Z)) of { S z → odd z; Z → True; }
```

So the following lemma is extracted:

```
(even (dAcc x (S (S Z))), even (dAcc x Z))
```

This lemma can be rewritten:

```
even (dAcc (S (S m)) Z) →
  even (dAcc m (S (S Z)))
```

so that one more lemma is extracted:

```
(even (dAcc (S (S x)) Z), even (dAcc x Z))
```

This technique of searching for lite lemmas is rather heuristic, hence the pairs of expressions thus found has to be checked for being improvement lemmas by means of supercompilation.

8 Parameterized two-level supercompiler

In this section we give a "recipe" for remaking an ordinary supercompiler into a two-level one, presenting a parameterized two-level supercompilation algorithm[2] shown in Fig. 24. This algorithm is an extension of the single-level supercompiler from [12].

The parameterized parts of algorithm are highlighted. There is one new parameterization (in comparison with a single-level supercompiler): the lower supercompiler SC^1 (used in lines 11, 12 and 14). All example residual programs presented in the paper were produced by the supercompiler SC^2 with a corresponding parameterization.

Theorem 7. *The supercompiler SC^2 terminates.*

Proof. Follows from the termination of the corresponding supercompiler SC_{ijk} [11, 12] and from the fact that after the application of an improvement lemma a folding step can be performed immediately. □

[2] This recipe is the first (but, hopefully, not the last one) in the emerging cookbook of multi-level supercompilation.

Figure 21 Parameterized supercompiler SC^2

1 $t = (e \rightarrow)$
2 **while** $unprocessedLeaf(t) \neq \bullet$ **do**
3 $\quad \beta = unprocessedLeaf(t)$
4 $\quad relAncs = \boxed{computeRelAncs(\beta)}$
5 $\quad \alpha = find(relAncs, \beta, \boxed{whistle})$
6 $\quad$ **if** $\alpha \neq \bullet$ *and* $\alpha.expr \simeq \beta.expr$ **then**
7 $\quad\quad t = fold(t,\ \alpha,\ \beta)$
8 $\quad$ **else if** $\alpha \neq \bullet$ *and* $\alpha.expr \lessdot \beta.expr$ **then**
9 $\quad\quad t = abstract(t, \beta, \alpha)$
10 $\quad$ **else if** $\alpha \neq \bullet$ **then**
11 $\quad\quad e_1 = \boxed{SC_1} [\![\alpha.expr]\!]$
12 $\quad\quad e_2 = \boxed{SC_1} [\![\beta.expr]\!]$
13 $\quad\quad \theta = e_2 \olessthan e_1$
14 $\quad\quad$ **if** $e_1 \simeq e_2$ *and* $\mathcal{N}_{\checkmark}(\boxed{SC_1} [\![\beta.expr]\!]) \trianglerighteq^{\checkmark} \mathcal{N}_{\checkmark}(\boxed{SC_1} [\![\alpha.expr\ \theta]\!])$ **then**
15 $\quad\quad\quad t = replace(t, \beta, \alpha.expr\ \theta)$
16 $\quad\quad$ **else if** $\alpha.expr \leftrightarrow \beta.expr$ **then**
17 $\quad\quad\quad t = split(t, \beta, \beta.expr)$
18 $\quad\quad$ **else**
19 $\quad\quad\quad t = abstract(t, \alpha, \beta)$
20 $\quad\quad$ **end**
21 $\quad$ **else**
22 $\quad\quad t = drive(t, \beta)$
23 $\quad$ **end**
24 **end**

The following details are parameterizable $\boxed{computeRelAncs}$, $\boxed{whistle}$, $\boxed{SC_1}$.

Almost any supercompiler (constructed in a classical style) can be remade into a two-level one on the bases of the techniques described here. Consider an arbitrary supercompiler SC'^1 (guaranteed to terminate). Suppose that the generalization of a configuration is performed in the following way:

1 $t = generalize(t, \alpha, \beta)$

This code can be refined in the following way:

1 $e_1 = SC_1 [\![\alpha.expr]\!]$
2 $e_2 = SC_1 [\![\beta.expr]\!]$
3 $\theta = e_2 \otimes e_1$
4 **if** $e_1 \simeq e_2$ *and* $\mathcal{N}_{\checkmark}(SC_1 [\![\beta.expr]\!]) \trianglerighteq^{\checkmark}_{\sim} \mathcal{N}_{\checkmark}(SC_1 [\![\alpha.expr\ \theta]\!])$ **then**
5 | $t = replace(t, \beta, \alpha.expr\ \theta)$
6 **else**
7 | $t = generalize(t, \alpha, \beta)$
8 **end**

where SC^1 is another arbitrary supercompiler (which is also guaranteed to terminate). Let $SC'^1_{SC_1}$ denote this extension. It is easy to show that this extension terminates.

Whether such an extension will be able to produce useful and/or non-trivial results, depends on implementation details of the supercompilers SC^1 and SC'^1. However, this scheme of extending a single-level supercompiler to a two-level one is formally applicable to a variety of supercompilers.

9 Conclusion

Constructing a multi-level supercompiler on the bases of a single-level one can be regarded a classic case of metasystem transition (using V.F. Turchin's terminology [22, 20]).

In this section we give a brief overview of the goals and methods of two-level supercompilation, and compare our approach with related techniques in other fields, as well as with other works on supercompilation.

9.1 Goals and methods of two-level supercompilation

The supercompiler HOSC was designed as a tool for program analysis by transformation, whose essence can be briefly explained as follows:

> Let p be a program we have to analyze in order to infer or prove its properties. Instead of directly analyzing the program p, we may transform it to a program p', which is *equivalent* to p, but is *easier to analyze.*

But, what does it mean that a transformed program p' is easier to analyze than the original program p? The answer to this question depends on what properties are to be dealt with and what analysis tools are available. For example in the paper [23] a program is transformed to a more modular and structured form for the purpose of extracting a specification from the transformed program.

Usually, a program is easier to analyze if it does not contain "dead" (i.e. unreachable) code. Such code may be clearly localized in a program (for example in the form of function definitions that are never called), or it may also be scattered

throughout the "live" code in small pieces. In the latter case the tool wastes its time analyzing dead or redundant code. *A program without "superfluous" code is likely to be more suitable for mechanical analysis.*

How to determine whether a program contains some "dead" code? In general the problem is undecidable. However, if a program p' is constructed from a perfect process tree produced by supercompilation, the program p' is certain not to contain any dead code [6, 21]. A partial process tree is a perfect one if its construction involves no generalizations. So the task of two-level supercompilation, when constructing a partial process tree, is to avoid generalizations as much as possible, in order for the residual program to contain as little redundant code as possible.

In the algorithm presented in this paper a generalization step involves two nodes: the upper node α and the lower node β, two different cases being considered [12].

1. If $\alpha.expr \trianglelefteq_c^* \beta.expr$, $\alpha.expr \not\lessdot \beta.expr$ and $\alpha.expr \lessdot \beta.expr$, then the lower configuration $\beta.expr$ is generalized.

2. If $\alpha.expr \trianglelefteq_c^* \beta.expr$ and $\alpha.expr \not\lessdot \beta.expr$, then the upper configuration $\alpha.expr$ is generalized.

All techniques of silencing the whistle considered in the paper are only related to the second case (i.e. to avoiding generalizations of the upper configurations). It would be interesting, however, to investigate the problems related to avoiding generalizations of the lower configurations.

9.2 Related work and further development

To some extent, this work is on the junction of program transformation and program analysis. The literature on each of the topics is abundant and diversified. It would be virtually impossible even to list all related works. Thus we confine ourselves to considering a number of classic works and to recent activities in related areas.

For a long time the developments of supercompilation occurred in isolation, with little cross-fertilization with other research areas. Perhaps this was due to the fact that supercompilation was primarily regarded as a means of program optimization. Many optimization techniques are rather language-specific, and supercompilation was initially developed for the language Refal, which was quite different from other programming languages.

However, since we are interested in supercompilation as a means of program analysis by transformation, we believe the ideas and method from other areas to be potentially useful also in the field of supercompilation, such cross-fertilization being able to produce a synergetic effect.

Let us enumerate some ideas and techniques related, or potentially applicable, to supercompilation.

The transformation system by Burstall and Darlington [4] uses a set of predefined lemmas (such as associativity, commutativity of operations) and relies on the use of "eurekas", additional function definition provided by the user. In many cases only the use of a eureka enables a non-trivial result of a transformation to be achieved. Usually a eureka formalizes an assumption about how an expression should be represented (syntactically) to be transformed by the system. It would be useful to enable the user of the supercompiler HOSC to provide "eurekas" helping the supercompiler in avoiding generalizations.

It would be also useful to enable the user to provide lemmas to be used by the supercompiler. The correctness of the lemmas could be automatically checked by supercompilation. These lemmas, again, could be used for avoiding generalizations.

The respect in which our system is similar to the theorem prover by Boyer-Moore [2] is that both systems can benefit from lemmas which are not discoverable (in reasonable time), but are provable by the system.

It should be noted that the Boyer-Moore system proves only partial correctness: all functions are assumed to be total recursive ones. Our system, however, guarantees total correctness.

A program transformation system based on generalized partial computation [5] uses an external theorem prover. Our system is able to discover and prove auxiliary lemmas on its own. However, it should be possible to integrate our system with some third-party theorem provers.

It seems reasonable to try to make use of some techniques of rippling [3]. Rippling is able to permit both orientations of a rewrite rule without the threat of non-termination (bi-directionality). On the other hand, it is possible to reduce corresponding parts of user-defined lemmas (checking that the improvement is preserved) and try to apply reduced variants of lemmas. Rippling [3], in a sense, is the opposite to what we did in Sections 6 and 7: we need to infer an improvement lemma applicable in the current context from a set of user-defined lemmas.

Distillation [7, 8], as well as two-level supercompilation, is a metasystem with respect to classic supercompilation. Instead of comparing configurations, distillation always compares supercompiled configurations. The differences between our system and distillation are: (1) in distillation configurations are always supercompiled which results in possible performance issues (2) distillation is a monolithic algorithm which is difficult to extend or customize. In our system the upper supercompiler tries to use the lower supercompiler only in cases where the whistle blows, thereby reducing possible performance issues. Also our system is modular by design: it allows both layers to be modified or refined: the upper supercompiler and the lower one. Moreover, copying with variations and duplication of lower-level subsystems is a typical scheme of metasystem transition [20, 22].

The performance issues of supercompilation are addressed by Jonsson and Nordlander in [9]: they use zippers (stacks) for representing configuration and use the corresponding (stacked) variation of homeomorphic embedding. We hope to investigate whether this stack representation can be used in two-level super-

compilation.

It was shown in Section 5 that two-level supercompilation is capable of achieving asymptotic runtime improvements of programs. Improvement lemmas discovered by two-level supercompiler can be used for program optimization. A method of generating optimizations from proofs of equivalence is describes in [19]. Improvement lemmas may be used as such proofs of equivalence.

Acknowledgements

The author expresses his gratitude to Sergei Romanenko, to all participants of Refal seminar at Keldysh Institute for useful comments and fruitful discussions of this work. Extra special thanks are to Sergei Abramov for his attention and support of this work.

References

[1] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.

[2] R. S. Boyer and J. S. Moore. Proving theorems about lisp functions. *Journal of the ACM (JACM)*, 22(1):129–144, 1975.

[3] A. Bundy, D. Basin, and D. Hutter. *Rippling: meta-level guidance for mathematical reasoning.* Cambridge University Press, 2005.

[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] Y. Futamura, Z. Konishi, and R. Glück. Program transformation system based on generalized partial computation. *New Gen. Comput.*, 20:75–99, January 2002.

[6] R. Glück and A. Klimov. Occam's razor in metacompuation: the notion of a perfect process tree. In *WSA '93: Proceedings of the Third International Workshop on Static Analysis*, pages 112–123, London, UK, 1993. Springer-Verlag.

[7] 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.

[8] G. W. Hamilton. A graph-based definition of distillation. In *Second International Workshop on Metacomputation in Russia*, 2010.

[9] P. Jonsson and J. Nordlander. Taming code explosion in supercompilation. In *Proceedings of the 2011 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM Press New York, NY, USA, 2011.

[10] I. Klyuchnikov. Higher-order supercompilation. *Program systems: theory and applications*, 4(4):37–71, 2010. `http://psta.psiras.ru/2010/03(003)/r3/r3-3.html` (in Russian).

[11] I. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[12] I. Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. Preprint 62, Keldysh Institute of Applied Mathematics, 2010.

[13] I. Klyuchnikov. Supercompiler HOSC: proof of correctness. Preprint 31, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[14] 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.

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

[16] N. Mitchell. Rethinking supercompilation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 309–320. ACM, 2010.

[17] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.

[18] M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Københavns Universitet, Datalogisk Institut, 1994.

[19] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. *SIGPLAN Not.*, 45:389–402, January 2010.

[20] V. F. Turchin. *The phenomen of science. A cybernetic approach to human evolution.* Columbia University Press, New York, 1977.

[21] V. F. Turchin. *The Language Refal: The Theory of Compilation and Metasystem Analysis.* Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.

[22] V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 481–509. Springer, 1996.

[23] M. Ward. Program analysis by formal transformation. *The Computer Journal*, 39(7):598, 1996.