KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Ilya Klyuchnikov

Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting

Moscow
2010

Ilya Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting

The paper describes the algorithm implemented in the supercompiler HOSC 1.5, dealing with programs written in a higher-order functional language. The design decisions behind the algorithm are illustrated through a series of examples. Of particular interest are the decisions related to generalization and homeomorphic embedding of expressions with bound variables.

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

Илья Ключников. Суперкомпилятор HOSC 1.5: гомеоморфное вложение и обобщение для выражений высшего порядка

В работе приводится алгоритм экспериментального суперкомпилятора HOSC 1.5, работающего с функциями высших порядков. Детали алгоритма обосновываются на ряде примеров. Особое внимание уделяется обобщению и гомеоморфному вложению выражений со связанными переменными.

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

Contents

1 Introduction

The supercompiler HOSC[1] is meant to be used as a tool for program analysis. For this reason HOSC propagates as mush positive information as possible when constructing a partial process tree.

The ability of HOSC to check Haskell expressions for equivalence was demonstrated in [11]. Another HOSC's feature is its tendency to normalize expressions, that is to transform equivalent expressions to the same syntactic form. Upon a slight modification, HOSC is also able to check if a pair of equivalent expressions forms an *improvement lemma*. This is achieved by annotating residual expressions with extra information (taken from the partial process tree) about the performance of the original program. Such lemmas can be used by a multi-level supercompiler in order to avoid generalizations when building partial process trees.

The supercompiler HOSC 1.0 was described in detail in [8].

In [10] the supercompiler HOSC is described in the form of a transformation relation and the correctness of this relation is proved.

An attempt to construct a termination proof for HOSC 1.0 revealed that it does not terminate for some input programs! Thus HOSC was revised to produce its new version, HOSC 1.1 described in [9]. The proof that HOSC 1.1 does terminate for any input program is given in [9].

It should be noted that the termination of HOSC 1.1 is ensured in a rather "ad hoc" manner: the partition of nodes into trivial and non-trivial ones is based on the sizes of expressions produced by reduction steps. Unfortunately, this change in partition has produced a negative effect on the "normalization power" of HOSC 1.1 (as compared to HOSC 1.0).

Here we describe a new version of the HOSC supercompiler: HOSC 1.5, which is superior to HOSC 1.0 at normalizing expressions, and yet terminates for any input program. The termination of HOSC 1.5 is ensured in a more elegant and general way than in the case of HOSC 1.1.

The explanations given in [8, 9, 10] about the rationale behind the design of HOSC are incomplete. The problem is that, unlike HOSC, the other existing supercompilers for higher-order languages are meant for program optimization. Hence, it was difficult to justify particular decisions because of the lack of material for comparison.

A peculiarity of HOSC 1.0 and HOSC 1.1 is that the definition of the refined homeomorphic relation is tightly coupled with the algorithm for computing a generalization of two expressions, the algorithm being defined for coupled expressions only. Thus in [8, 9] the problems of generalization of expressions with bound variables were just circumvented, rather than really solved, since the algorithm in [8], by construction, *doesn't touch bound variables*.

Here we present a revised version of the aforementioned algorithm, which is applicable to *any* pair of expressions, even if the expressions are not coupled. In

[1] The source code of HOSC is available at `http://hosc.googlecode.com`

Figure 1 SLL syntax

P	::=	$d_1 \ldots d_n$	program
d	::=	$f(v_1, \ldots, v_n) = e;$	f-function
	|	$g(p_1, v_1, \ldots, v_n) = e_1;$	g-function
		$\ldots$	
		$g(p_m, v_1, \ldots, v_n) = e_m;$	
e	::=	v	variable
	|	$C(e_1, \ldots, e_n)$	constructor
	|	$f(e_1, \ldots, e_n)$	call to f-function
	|	$g(e_1, \ldots, e_n)$	call to g-function
p	::=	$C(v_1, \ldots, v_n)$	pattern

addition, if the expressions do not contain bound variables, the new algorithm just degenerates into the "classical" one, which makes it easier to compare HOSC with other supercompilers.

We consider several algorithms of supercompilation for the core Haskell language. A series of examples explains the decisions taken in the design of HOSC 1.5 and demonstrates the advantages of its supercompilation algorithm in the context of program analysis.

2 SLL: embedding and generalization

First of all, we consider generalization and homeomorphic embedding in a first-order setting – for a first-order functional language SLL (i.e. *without bound variables*).

The SLL[2] language is considered in [17, 18]. Its syntax is shown in Fig. 1. An SLL expression is a variable, a constructor or a function call. All constructors and functions have fixed arity. The number of constructors and function symbols is finite.

The notions of a substitution, an instance and a most specific generalization for SLL expressions are defined in [18]. The paper [5, 20] considers a slightly extended version of SLL with `if`, a special construction for equality checking.

We will consider the homeomorphic embedding relation and the algorithm of computing a most specific generalization defined in [17, 18, 20] as a reference point for further refinements in a higher-order setting.

The notation $h(e'_1, \ldots, e'_n)$ is used for denoting either a constructor or a function call.

[2] SLL = Simple Lazy Language

Рис. 2 SLL: iterative algorithm of generalization of e' and e''

Initial trivial generalization

$$(v, \{v := e_1\}, \{v := e_2\})$$

Common functor rule

$$\left(\begin{array}{c} e \\ \{v := h(e'_1,\ \ldots,\ e'_n)\} \cup \theta' \\ \{v := h(e''_1,\ \ldots,\ e''_n)\} \cup \theta'' \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v := h(v_1,\ \ldots,\ v_n)\} \\ \{v_1 := e'_1,\ \ldots,\ v_n := e'_n\} \cup \theta' \\ \{v_1 := e''_1,\ \ldots,\ v_n := e''_n\} \cup \theta'' \end{array}\right)$$

Common subexpression rule

$$\left(\begin{array}{c} e \\ \{v_1 := e',\ v_2 := e'\} \cup \theta' \\ \{v_1 := e'',\ v_2 := e''\} \cup \theta'' \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v_1 := v_2\}) \\ \{v_2 := e'\} \cup \theta' \\ \{v_2 := e''\} \cup \theta'' \end{array}\right)$$

Figure 3 SLL: recursive algorithm of generalization of e' and e''

Most specific generalization

- $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$

Common functor rule

- $v \mathbin{\widetilde{\sqcap}} v = (v, \{\}, \{\})$
- $h(e'_1, \ldots, e'_n) \mathbin{\widetilde{\sqcap}} h(e''_1, \ldots, e''_n) = (h(e_1, \ldots, e_n), \bigcup \theta'_i, \bigcup \theta''_i)$ where
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $e_1 \mathbin{\widetilde{\sqcap}} e_2 = (v, \{v := e_1\}, \{v := e_2\})$

Common subexpression rule

- $s(e, \{\}, \{\}) = s(e, \{\}, \{\})$
- $s(e, \{v_1 := e'\} \cup \theta', \{v_1 := e''\} \cup \theta'') = s'(s(e, \theta', \theta''))$ where
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e\{v_1 := v_2\},\ \theta'_1,\ \theta''_1)$ if $\exists v_2 : \{v_2 := e'\} \in \theta'_1, \{v_2 := e''\} \in \theta''_1$
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e,\ \{v_1 := e'\} \cup \theta'_1,\ \{v_1 := e''\} \cup \theta''_1)$ otherwise

Definition 1 (Iterative generalization algorithm for SLL expressions)**.** A most specific generalization of two expressions e' and e'', $e' \sqcap e''$ is computed by exhaustively applying the rewrite rules in Fig. 2 to the initial trivial generalization $(v, \{v := e'\}, \{v := e''\})$.

The algorithm 1 is presented in an iterative form in the same way as in [19, 20, 18].

Figure 4 SLL: homeomorphic embedding

Embedding	
$e' \trianglelefteq e''$	if $e' \trianglelefteq_d e''$ or $e' \trianglelefteq_c e''$ or $e' \trianglelefteq_v e''$
Variables	
$v' \trianglelefteq_v v''$	
Coupling	
$h(e'_1, \ldots, e'_n) \trianglelefteq_c h(e''_1, \ldots, e''_n)$	if $\forall i : e'_i \trianglelefteq e''_i$
Diving	
$e \trianglelefteq_d h(e'_1, \ldots, e'_n)$	if $\exists i : e \trianglelefteq e'_i$

However, the computation of a most specific generalization can also be defined in terms of recursive functions (big step semantics), rather than rewrite rules (small step semantics), the recursive definition being more convenient for our purposes.

Definition 2 (Recursive generalization algorithm for SLL expressions)**.** $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$, where operations $\widetilde{\sqcap}$ are s defined in Fig. 3.

The computation of MSG is performed in two stages. The first stage corresponds to the application of the common functor rule (the calculation of $e_1 \widetilde{\sqcap} e_2$), while the second stage (the operation s) corresponds to the application of the common subexpression rule.

Fig. 4 presents homeomorphic embedding relation of SLL expressions as defined in [17, 19]. We explicitly distinguish coupling, diving and variable embedding.

An SLL expression e is said to be *derivable* from a program P if all constructor and function names appearing in e also appear in P (and have the same arities). A set of SLL expressions derivable from P is also said to be derivable from P.

Since an SLL program P may only contain a finite number of constructor and function names with fixed arities, the same is true of any set of expressions derivable from P, and $\trianglelefteq$ is a well-quasi-order on this set. This result is due to Higman and Kruskal.

It can be shown that $\trianglelefteq_c$ is also a well-quasi-order for a set of expressions derivable from a program P.

In addition, for any expressions e_1 and e_2, such that $e_1 \trianglelefteq_c e_2$, their generalization $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$ is guaranteed to be non-trivial (in other words, e_g is not a variable). It means that, when constructing a partial process tree, we do not have to split configurations, regardless of the computation history, because, if we use $\trianglelefteq_c$ as a whistle, then there always exists a non-trivial generalization of embedded expressions.

Figure 5 HLL syntax

$tDef$ ::= data $tCon = \overline{dCon_i}$;	type definition
$tCon ::= tn\ \overline{tv_i}$	type constructor
$dCon ::= c\ \overline{type_i}$	data constructor
$type ::= tv \mid tCon \mid type \to type \mid (type)$	type expression
$prog ::= \overline{tDef_i}\ e$ **where** $\overline{f_i = e_i}$;	program
$e ::= v$	variable
$\mid c\ \overline{e_i}$	constructor
$\mid f_g$	global variable
$\mid \lambda\ \overline{v_i} \to e$	λ-abstraction
$\mid e_1\ e_2$	application
$\mid$ **case** e_0 **of** $\{\overline{p_i \to e_i;}\}$	case-expression
$\mid$ **let** $\overline{v_i = e_i;}$ **in** e	let-expression
$\mid (e)$	parenthesized expression
$p ::= c\ \overline{v_i}$	pattern

3 HLL: embedding and generalization

The input language of the supercompiler HOSC is HLL[3] which is quite close to the Haskell Kernel [22, 23]. HLL syntax is shown in Fig. 5. We require that patterns in case expressions be non-overlapping and exhaustive. HLL is statically typed according to the Hindley-Milner polymorphic typing system[4]

Remark 3 (Order of patterns in `case`-expressions)**.** We require that patterns in `case`-expressions be enumerated in the same order as constructors in the declaration of a corresponding data type.

In order to denote that a variable f is defined in a program (that is, there is a definition $f = e$;), we use an index g (global) and write f_g.

Since HLL-expressions may contain bound variables, we need to formalize some related concepts.

Definition 4 (Bound, global and free variables of an expression)**.** Sets $bv[\![e]\!]$, $gv[\![e]\!]$, $fv[\![e]\!]$ of bound variables, global variables and global variables of an expression e are computed according to the rules shown in Fig. 6, 7 and 8, respectively.

Definition 5 (Multi-set of bound variables of an expression)**.** The rules, defining a multi-set $bv'[\![e]\!]$ of bound variables of an expression e, are the same as in Fig. 6, but take into account that the result of the operation is a multi-set.

[3] HLL = Higher-order Lazy Language

[4] Really all HLL supercompilers described in this paper are able to consume programs with more general typings. We use Hindley-Milner type system for simplicity only.

Figure 6 HLL: a set of bound variables of an expression

$bv[\![f_g]\!]$	$= \{\}$
$bv[\![v]\!]$	$= \{\}$
$bv[\![c\ \overline{e_i}]\!]$	$= \bigcup bv[\![e_i]\!]$
$bv[\![\lambda v \to e]\!]$	$= bv[\![e]\!] \ \cup \ \{v\}$
$bv[\![e_1\ e_2]\!]$	$= bv[\![e_1]\!] \cup bv[\![e_2]\!]$
$bv[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$	$= bv[\![e_0]\!] \cup (\bigcup bv[\![e_i]\!]) \cup (\bigcup v_{ik})$
$bv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$	$= bv[\![e]\!] \cup (\bigcup bv[\![e_i]\!]) \ \cup \ (\bigcup v_i)$

Figure 7 HLL: a set of global variables of an expression

$gv[\![f_g]\!]$	$= \{f_g\}$
$gv[\![v]\!]$	$= \{\}$
$gv[\![c\ \overline{e_i}]\!]$	$= \bigcup gv[\![e_i]\!]$
$gv[\![\lambda v \to e]\!]$	$= gv[\![e]\!]$
$gv[\![e_1\ e_2]\!]$	$= gv[\![e_1]\!] \cup gv[\![e_2]\!]$
$gv[\![case\ e\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$	$= gv[\![e]\!] \cup (\bigcup gv[\![e_i]\!])$
$gv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$	$= gv[\![e]\!] \cup (\bigcup gv[\![e_i]\!])$

Figure 8 HLL: a set of free variables of an expression

$fv[\![f_g]\!]$	$= \{\}$
$fv[\![v]\!]$	$= \{v\}$
$fv[\![c\ \overline{e_i}]\!]$	$= \bigcup fv[\![e_i]\!]$
$fv[\![\lambda v \to e]\!]$	$= fv[\![e]\!] \setminus \{v\}$
$fv[\![e_1\ e_2]\!]$	$= fv[\![e_1]\!] \cup fv[\![e_2]\!]$
$fv[\![case\ e\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$	$= fv[\![e]\!] \cup (\bigcup fv[\![e_i]\!] \setminus \{\overline{v_{ik}}\})$
$fv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$	$= (fv[\![e]\!] \setminus (\bigcup v_i)) \cup (\bigcup fv[\![e_i]\!])$

Remark 6 (Variable names)**.** In order to avoid name clashes, we require that for any expression e the sets $bv[\![e]\!]$, $gv[\![e]\!]$, $fv[\![e]\!]$ be mutually disjoint, and there be no repeated elements in the multi-set $bv'[\![e]\!]$.

Definition 7 (Refreshing of bound variables)**.** The refreshing of bound variables of an expression e is a consistent replacement of every bound variable $v \in bv'[\![e]\!]$ in e by a fresh variable. We denote this operation as $fresh[\![e]\!]$.

Definition 8 (HLL-substitution)**.** A substitution is a finite list of pairs $\theta = \{\overline{v_i := e_i}\}$, where a variable v_i is bound to its value e_i. The domain of θ is defined as $domain(\theta) = \{\overline{v_i}\}$. The range of θ is computed as $range(\theta) = \{\overline{e_i}\}$.

HLL language is based on λ-calculus. A substitution is a fundamental operation in λ-calculus. In order to ensure the correctness of a substitution, expressions are considered up to renaming of bound variables, that is up to $\equiv_\alpha$ ([2], 2.1.11). Thus an application of substitution should be correct for $\equiv_\alpha$-classes of expressions

([2], Appendix C). That is:

$$e \equiv_\alpha e',\ e_i \equiv_\alpha e'_i \Rightarrow e\{\overline{v_i := e_i}\} \equiv_\alpha e'\{\overline{v_i := e'_i}\}$$

There exist different approaches to this problem:

1. One may consider "canonical" nameless expressions where bound variables do not have names [4]. The whole class of α-congruent expressions corresponds to one expression in this approach. This approach is good for mechanical processing but is inconvenient for a human.

2. It is possible to refresh bound variables of expressions before applying a substitution in order to avoid name capture [3, 21]. This approach is inconvenient for cases where a substitution is a result of some operation (e.g. generalization).

3. Substitutions may be avoided by means of the mechanism of *explicit substitutions* [1, 14]. But, in the context of supercompilation, this approach would considerably complicate configuration analysis.

4. It is possible to extend Remark 6 to the case of the application of a substitution. That is, to define a notion of an allowed substitution with respect to a given expression e and then consider allowed substitutions only.

The last alternative is the most convenient one in the context of supercompilation (more precisely – for a task of computing the most specific generalization).

Definition 9 (Allowed HLL-substitution)**.** A HLL-substituion θ is allowed with respect to an expression e if

1. $bv[\![e]\!] \cap domain(\theta) = \emptyset$
2. $\forall e_i \in range(\theta) : bv[\![e]\!] \cap fv[\![e_i]\!] = \emptyset$

Remark 10 (Usage of substitutions)**.** In the following all substitutions we consider are required to be allowed (with respect to corresponding expression).

Convention 10 is similar to the Barendregt's convention ([2], 2.1.13). Conventions 6 and 10 ensure the correctness of a simple ("naive") application of substitution:

Definition 11 (Application of HLL-substitution)**.** The result of applying a substitution θ to an expression e, $e\theta$, is computed according to the rules in Fig. 9.

Note that when a substitution is applied to a variable, the variable is replaced with an expression whose bound variables get refreshed!

Figure 9 HLL: application of substitution

$v\theta$	$=$	$fresh[\![e]\!]$	if $v := e \in \theta$
	$=$	v	otherwise
$f_g\theta$	$=$	f_g	
$(c\ \overline{e_i})\theta$	$=$	$c\ \overline{(e_i\theta)}$	
$(\lambda v \to e)\theta$	$=$	$\lambda v \to (e\theta)$	
$(e_1\ e_2)\theta$	$=$	$(e_1\theta)\ (e_2\theta)$	
$(case\ e\ of\ \{\overline{p_i\ \to e_i;}\})\theta$	$=$	$case\ (e\theta)\ of\ \{\overline{p_i\ \to (e_i\theta);}\}$	
$(let\ \overline{v_i = e_i;}\ in\ e)\theta$	$=$	$let\ \overline{v_i = (e_i\theta);}\ in\ (e\theta)$	

Remark 12 (Elimination of `let`-expressions)**.** As in [8], a program p is λ-lifted [6] before supercompilation and replaced with a program p' that does not contain `let`-expressions. The reasons for the elimination of `let`-expressions are as follows: (1) the notions of renaming, instance and generalizations are quite complicated for expressions containing `let`-expressions, since permutations of `let`-bindings are possible and should be taken into account, (2) a `let`-expression in a residual program represents the result of a generalization, so we would need to distinguish the original `let`-expressions from the results of generalizations, (3) since HOSC is intended for program analysis, the elimination of `let`-expressions allows positive information to be propagated in a more aggressive way, which results in a deeper transformation of programs. So in the following we only consider HLL expressions without `let`-expressions.

Definition 13 (Instance of an expression)**.** An expression e_2 is said to be an instance of an expression e_1, or $e_1 \lessdot e_2$, if there is a substitution θ, such that $e_1\theta \equiv e_2$. As far as the language HLL is concerned, the substitution θ is unique and is denoted by $e_1 \lhd e_2$.

Definition 14 (Renaming)**.** An expression e_2 is a *renaming* of an expression e_1, $e_1 \simeq e_2$, if $e_1 \lessdot e_2$ and $e_2 \lessdot e_1$. That is, e_1 and e_2 differ in names of free variables only.

Definition 15 (Generalization)**.** A generalization of 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 $e_g\theta_2 \equiv e_2$. The set of generalizations of expressions e_1 and e_2 is denoted by $e_1 \frown e_2$.

Definition 16 (Most specific generalization)**.** A generalization $(e_g, \theta_1, \theta_2) \in e_1 \frown e_2$ is a *most specific* one, if, for any generalization $(e'_g, \theta'_1, \theta'_2) \in e_1 \frown e_2$, it holds that $e'_g \lessdot e_g$, that is e_g is an instance of e'_g. We suppose that there is defined an operation $\sqcap$, such that $e_1 \sqcap e_2$ is a most specific generalization of e_1 and e_2.

Definition 17 (Incommensurable expressions)**.** Expressions e_1 and e_2 are incommensurable, $e_1 \leftrightarrow e_2$, if $e_1 \sqcap e_2 = (v, \theta_1, \theta_2)$, that is the most specific generalization of given expressions is a variable.

Figure 10 HLL: calculation of the most specific generalization

Most specific generalization

- $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$

Common functor rule

- $v \mathbin{\widetilde{\sqcap}} v = (v, \{\}, \{\})$
- $c\ \overline{e'_i} \mathbin{\widetilde{\sqcap}} c\ \overline{e''_i} = (c\ \overline{e_i},\ \bigcup \theta'_i,\ \bigcup \theta''_i)$, where
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $e'_1\ e'_2 \mathbin{\widetilde{\sqcap}} e''_1\ e''_2 = (e_1\ e_2,\ \theta'_1 \mathbin{\widetilde{\sqcap}} \theta'_2,\ \theta''_1 \mathbin{\widetilde{\sqcap}} \theta''_2)$, where
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $\lambda v' \to e' \sqcap \lambda v'' \to e'' = (e_g, \theta', \theta'')$, <u>if θ' and θ'' correct wrt e_g</u>, where
  - $e_g = \lambda v \to e$
  - $(e, \theta', \theta'') = \underline{e'\{v' := v\}} \mathbin{\widetilde{\sqcap}} \underline{e''\{v'' := v\}}$
- $case\ e'_0\ of\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \mathbin{\widetilde{\sqcap}} case\ e''_0\ of\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\} = (e_g, \bigcup \theta'_i, \bigcup \theta''_i)$, <u>if all θ'_i and θ''_i are correct wrt e_g</u>, where
  - $e_g = case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$
  - $(e_0, \theta'_0, \theta''_0) = e'_0 \mathbin{\widetilde{\sqcap}} e''_0$
  - $(e_i, \theta'_i, \theta''_i) = \underline{e'_i\{\overline{v'_{ik} := v_{ik}}\}} \mathbin{\widetilde{\sqcap}} \underline{e''_i\{\overline{v''_{ik} := v_{ik}}\}}$
- $e_1 \mathbin{\widetilde{\sqcap}} e_2 = (v, \{v := e_1\}, \{v := e_2\})$

Common subexpression rule

- $s(e, \{\}, \{\}) = s(e, \{\}, \{\})$
- $s(e, \{v_1 := e'\} \cup \theta', \{v_1 := e''\} \cup \theta'') = s'(s(e, \theta', \theta''))$ where
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e\{v_1 := v_2\},\ \theta'_1,\ \theta''_1)$
    if $\exists v_2 : \{v_2 := e'\} \in \theta'_1, \{v_2 := e''\} \in \theta''_1$
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e,\ \{v_1 := e'\} \cup \theta'_1,\ \{v_1 := e''\} \cup \theta''_1)$
    otherwise

The substitutions θ_1 and θ_2 in a most specific generalization $(e_g, \theta_1, \theta_2)$ should be allowed ones with respect to the expression e_g. In the context of conventions we have chosen an algorithm of computation of the most specific generalization should ensure the correctness of substitutions it produces.

Figure 11 HLL: pure homeomorphic embedding

Embedding	
$e' \trianglelefteq e''$	if $e' \trianglelefteq_d e''$, $e' \trianglelefteq_c e''$ or $e' \trianglelefteq_v e''$
Variables	
$f_g \trianglelefteq_v f_g$	
$v \trianglelefteq_v v'$	
Coupling	
$c\ \overline{e'_i} \trianglelefteq_c c\ \overline{e''_i}$	if $\forall i : e'_i \trianglelefteq e''_i$
$\lambda v' \to e' \trianglelefteq_c \lambda v'' \to e''$	if $e' \trianglelefteq e''$
$e'_1\ e'_2 \trianglelefteq_c e''_1\ e''_2$	if $\forall i : e'_i \trianglelefteq e''_i$
$case\ e'\ of\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \trianglelefteq_c case\ e''\ of\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\}$	if $e' \trianglelefteq e''$ and $\forall i : e'_i \trianglelefteq e''_i$
Diving	
$e \trianglelefteq_d c\ \overline{e_i}$	if $\exists i : e \trianglelefteq e_i$
$e \trianglelefteq_d \lambda v_0 \to e_0$	if $e \trianglelefteq e_0$
$e \trianglelefteq_d e_1\ e_2$	if $\exists i : e \trianglelefteq e_i$
$e \trianglelefteq_d case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$	if $\exists i : e \trianglelefteq e_i$

Definition 18 (Recursive algorithm of computing the most specific generalization of HLL expressions)**.** $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$, where the operations $\widetilde{\sqcap}$ and s are defined in Fig. 10.

Algorithm 18 is applicable to *any* pair of *HLL expressions* and produces a most specific generalization taking into account Conventions 6 and 10. The rules are applied in order of their enumeration. Variables v and v_{ik} in the 3rd, 4th and 5th common subexpression rules are fresh ones. The most interesting parts of the algorithm, taking into account bound variables, are underlined.

Definition 19 (Pure homeomorphic embedding $\trianglelefteq$)**.** The pure homeomorphic embedding of HLL-expressions is defined inductively according to the rules in Fig. 11.

In the following definition of a refined version of homeomorphic embedding, we use a table recording the correspondence of bound variables:

$$\rho = \{(v'_1,\ v''_1),\ \ldots,\ (v'_n,\ v''_n)\}$$

Definition 20 (Refined homeomorphic embedding $\trianglelefteq^*$)**.** The refined homeomorphic embedding of HLL-expressions is defined inductively according to the rules in Fig. 12.

Recall, that incommensurable SLL expression can not be coupled: if $e_1 \trianglelefteq_c e_2$, then $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$ is a non-trivial generalization (e_g is not a variable). The same is true for the refined embedding $\trianglelefteq^*$. However, this does not hold for the pure embedding $\trianglelefteq$. So, sometimes, when using $\trianglelefteq$ as a whistle, we have to split

Figure 12 HLL: refined homeomorphic embedding

Embedding

$e' \trianglelefteq^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq^* e'' \mid_{\{\}}$

$e' \trianglelefteq_c^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq_c^* e'' \mid_{\{\}}$

$e' \trianglelefteq_d^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq_d^* e'' \mid_{\{\}}$

Embedding wrt table of bound variables

$e' \trianglelefteq^* e'' \mid_\rho$ if $e' \trianglelefteq_d^* e \mid''_\rho$, $e' \trianglelefteq_c^* e'' \mid_\rho$ or $e' \trianglelefteq_v^* e'' \mid_\rho$

Variables

$f_g \trianglelefteq_v^* f_g \mid_\rho$

$v' \trianglelefteq_v^* v'' \mid_\rho$ if $(v', v'') \in \rho$

$v' \trianglelefteq_v^* v'' \mid_\rho$ if $v' \notin domain(\rho)$ and $v'' \notin range(\rho)$

Coupling

$c\ \overline{e'_i} \trianglelefteq_c^* c\ \overline{e''_i} \mid_\rho$ if $\forall i : e'_i \trianglelefteq^* e''_i \mid_\rho$

$\lambda v' \to e' \trianglelefteq_c^* \lambda v'' \to e'' \mid_\rho$ if $e' \trianglelefteq^* e'' \mid_{\rho \cup \{(v', v'')\}}$

$e'_1\ e'_2 \trianglelefteq_c^* e''_1\ e''_2 \mid_\rho$ if $\forall i : e'_i \trianglelefteq^* e''_i \mid_\rho$

$case\ e'\ of\ \{c_i\ \overline{v'_{ik}} \to e'_i;\} \trianglelefteq_c^* case\ e''\ of\ \{c_i\ \overline{v''_{ik}} \to e''_i;\} \mid_\rho$
if $e' \trianglelefteq^* e'' \mid_\rho$ and $\forall i : e'_i \trianglelefteq^* e''_i \mid_{\rho \cup \{\overline{(v'_{ik}, v''_{ik})}\}}$

Diving only if $fv(e) \cap domain(\rho) = \emptyset$

$e \trianglelefteq_d^* c\ \overline{e_i} \mid_\rho$ if $\exists i : e \trianglelefteq^* e_i \mid_\rho$

$e \trianglelefteq_d^* \lambda v_0 \to e_0 \mid_\rho$ if $e \trianglelefteq^* e_0 \mid_{\rho \cup \{(\bullet, v_0)\}}$

$e \trianglelefteq_d^* e_1\ e_2 \mid_\rho$ if $\exists i : e \trianglelefteq^* e_i \mid_\rho$

$e \trianglelefteq_d^* case\ e'\ of\ \{\overline{c_i\ v_{ik} \to e_i;}\} \mid_\rho$ if $e \trianglelefteq^* e' \mid_\rho$ or $\exists i : e \trianglelefteq^* e_i \mid_{\rho \cup \{\overline{(\bullet, v_{ik})}\}}$

down configurations (in order to ensure the termination of supercompilation), regardless of the history of computation.

4 A generic supercompilation algorithm

The notions of driving, partial process tree and operations over partial process tree used in this section are taken from [8]. Residual programs are constructed according to the rules corresponding to the transformation relation HOSC ([10], Fig. 15).

The generic supercompilation algorithm shown in Fig. 13 is a generalization of the algorithm in [20].

Any implementation of the generic algorithm has to give answers to the following questions:

1. *computeRelAncs* – how to compute relevant ancestors for a given leaf (in order to search for repeated configurations, super configurations and embedding configurations).

2. *whistle* – how to check two configurations for embedding.

Figure 13 A scheme of supercompilation algorithm

$t = (e \rightarrow)$
while $unprocessedLeaf(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaf(t)$
  $relAncs = computeRelAncs(\beta)$
  $\alpha = find(relAncs, \beta, whistle)$
  if $\alpha \neq \bullet$ *and* $\alpha.expr \simeq \beta.expr$ **then**
    $t = fold(t,\ \alpha,\ \beta)$
  else if $\alpha \neq \bullet$ *and* $\alpha.expr \lessdot \beta.expr$ **then**
    $t = abstract(t, \beta, \alpha)$
  else if $\alpha \neq \bullet$ **then**
    if $\alpha.expr \leftrightarrow \beta.expr$ **then**
      $t = split(t, \beta, \beta.expr)$
    else
      $t = abstract(t, \alpha, \beta)$
    end
  else
    $t = drive(t, \beta)$
  end
end

Figure 14 HLL: splitting of configuration

$split(t, \beta, e_1 e_2)$	$=$	$replace(t, \beta, e_s)$, where $e_s = let\ v_1 = e_1;\ v_2 = e_2;\ in\ v_1 v_2$
$split(t, \beta, case\ v\ of\ \{\overline{p_i \rightarrow e_i;}\})$	$=$	$addChildren(t, \beta, [v, \overline{e_i}])$
$split(t, \beta, case\ e\ of\ \{\overline{p_i \rightarrow e_i;}\})$	$=$	$replace(t, \beta, e_s)$, where $e_s = let\ v = e\ in\ case\ v\ of\ \{\overline{p_i \rightarrow e_i;}\}$
$split(t, \beta, e)$	$=$	$drive(t, \beta)$

Figure 15 Classes of HLL expressions

$expClass(let\ \overline{v_i = e_i}\ in\ e)$	$=$	0
$expClass(v\ \overline{e_i})$	$=$	0
$expClass(c\ \overline{e_i})$	$=$	0
$expClass(\lambda v \rightarrow\ e)$	$=$	0
$expClass(con\langle(\lambda v \rightarrow e_0)\ e_1\rangle)$	$=$	1
$expClass(con\langle f_g\rangle)$	$=$	2
$expClass(con\langle case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \rightarrow e_i;}\}\rangle)$	$=$	3
$expClass(con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \rightarrow e_i;}\}\rangle)$	$=$	4

If the configurations e' and e'' are incommensurable, we split the lower configuration. In a higher-order setting, this operation is a bit complicated, because

of bound variables in `case`-expressions. In order to ensure termination, it is desirable for the splitting to produce expressions that are smaller in size, than the original expression.

Definition 21 (Splitting of configuration)**.** Splitting of HLL configurations is performed according to the rules in Fig. 14.

A feature of the splitting thus defined is the processing of `case`-expressions. If a selector of a `case`-expression is a variable, then we perform a step similar to a driving step, but *without positive information propagation*, thus ensuring that the size of all expressions decreases. If a selector is not a variable, we extract the selector first.

An obvious disadvantage of the above approach is that the splitting of `case`-expressions is performed without taking into account the structure of the upper configuration (embedded in the lower one). Fortunately, the supercompiler HOSC is designed in such a way that splitting is rarely required.

5 Supercompiler HOSC 1.5

The supercompiler HOSC 1.5 is a parameterized supercompiler $\mathbf{SC_{ijk}}$, which, given different parameters, is able to produce various residual programs. Generating several residual programs may be of interest in cases where supercompilation is used for program analysis, for example for detecting improvement lemmas [12].

Before defining concrete functions *whistle* and *computeRelAncs*, implemented in $\mathbf{SC_{ijk}}$, we have to introduce a few definitions.

Definition 22 (Classes of HLL expressions)**.** All HLL expressions are divided into 5 *classes* according to the rules in Fig. 15.

Definition 23 (Trivial node)**.** A node β is *trivial* if $\beta.expr$ is either an observable or a `let`-expression. $(expClass(\beta.expr) = 0)$

Definition 24 (β-transient node)**.** A node β is *β-transient* if $\beta.expr = con\langle(\lambda v_0 \to e_0)\ e_1\rangle$. $(expClass(\beta.expr) = 1)$

Definition 25 (Global node)**.** A node β is *global* if $\beta.expr = con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$. $(expClass(\beta.expr) = 4)$

Definition 26 (Local node)**.** All nodes except global ones are *local*.

Definition 27 (Candidate for folding)**.** All nodes except for trivial and β-transient ones are candidates for folding. $(expClass(\beta.expr) > 1)$

Remark 28 (Rationale for choosing candidates for folding)**.** If we consider β-transient nodes as candidates for folding, then all supercompilers (defined below) produce poor residual programs. The elimination of β-transient nodes from candidates for folding is safe for termination, since it is impossible to get an infinite path in a partial process tree without a candidate.

Definition 29 (Relevant ancestors)**.** The relevant (with respect to control) ancestors of β are defined as follows:

- All global ancestors $\overline{\alpha_i}$ of β, which are candidates, if β is a global node.
- All local ancestors $\overline{\alpha_i}$ of β, such that there are no global nodes between α_i and β, if β is a local node.

Now let us consider a concrete algorithm SC_{ijk}, which depends on three parameters:

1. i – Which homeomorphic embedding should be used as a whistle: $\trianglelefteq_c^*$ (+) or $\trianglelefteq_c$ (−)?
2. j – Whether to partition the nodes into global and local ones (+) or not (−)?
3. k – Whether to require the candidates for folding to belong to the same expression class (+) or not (−)?

More formally, the operations *whistle* and *computeRelAncs* for supercompiler $\mathbf{SC_{ijk}}$ are defined as follows:

- $whistle(e_1, e_2) =$
  - $e_1 \trianglelefteq_c e_2$, if $j = -$ and $k = -$
  - $e_1 \trianglelefteq_c^* e_2$, if $j = +$ and $k = -$
  - $e_1 \trianglelefteq_c e_2$ if $expClass(e_1) = expClass(e_2)$, if $j = -$ and $k = +$
  - $e_1 \trianglelefteq_c^* e_2$ if $expClass(e_1) = expClass(e_2)$, if $j = +$ and $k = +$
- $computeRelAncs(\beta) =$
  - all candidates that are ancestors of β, if $j = -$
  - all relevant ancestors of β, if $j = +$

So the algorithm $\mathbf{SC_{ijk}}$ defines eight variations of the generic supercompilation algorithm.

Theorem 30 (Correctness)**.** *All supercompilers* $\mathbf{SC_{ijk}}$ *produce residual programs that are equivalent to the source ones.*

Proof. Without the operation of splitting of `case`-expressions, all defined supercompilers satisfy the transformation relation *HOSC* [10]. The operation of splitting of `case`-expressions is used in deforestation, and its correctness is shown in [15]. □

Theorem 31 (Termination)**.** *All supercompilers* $\mathbf{SC_{ijk}}$ *terminate for any well-typed input program.*

Proof. Typing ensures that any sequence of consecutive β-transient nodes is finite, so we can exclude β-transient nodes from our consideration. Relations $\trianglelefteq$ and $\trianglelefteq^*$

Figure 16 Tests

`length (concat xs)`	$\cong$	`sum (map length xs)`	(1)
`map f (append xs ys)`	$\cong$	`append (map f xs) (map f ys)`	(2)
`filter p (map f xs)`	$\cong$	`map f (filter (compose p f) xs)`	(3)
`map f (concat xs)`	$\cong$	`concat (map (map f) xs)`	(4)
`iterate f (f x)`	$\cong$	`map f (iterate f x)`	(5)
`map (compose f g)`	$\cong$	`compose (map f) (map g)`	(6)
`map f xs`	$\cong$	`join xs (compose return f)`	(7)

Figure 17 Comparing supercompilers

	$\mathbf{Sc}_{---}$	$\mathbf{Sc}_{--+}$	$\mathbf{Sc}_{-+-}$	$\mathbf{Sc}_{-++}$	$\mathbf{Sc}_{+--}$	$\mathbf{Sc}_{+-+}$	$\mathbf{Sc}_{++-}$	$\mathbf{Sc}_{+++}$
(1)	–	–	–	–	–	+	–	+
(2)	–	+	+	+	–	+	+	+
(3)	–	+	–	+	–	+	–	+
(4)	–	–	–	–	–	+	–	+
(5)	–	–	+	+	–	–	+	+
(6)	–	+	+	+	–	+	+	+
(7)	+	+	+	+	+	+	+	+

are well-quasi-order on a set of expressions labeling a partial process tree. The termination of a supercompiler without global/local control follows from Lemma 38 in [9]. It is easy to show that the partition of nodes into global and local ones does not affect the termination properties of the supercompiler. Indeed, local control ensures that any infinite path in a partial process tree has only finite portions of consecutive local nodes. So there should be an infinite number of global nodes in this path, but global control does not allow this situation. □

The termination of all supercompilers is ensured in a more elegant way than it was done in the supercompiler HOSC 1.1 [9], where the candidates were defined depending on the size of the expression produced by a reduction step.

The additional requirement that, to be compared, the configurations belong to the same class of expressions is, to some extent, similar to Turchin's "stack" whistle [25]. If configurations are represented as stacks, then the equality of expression classes corresponds to the equality of prefixes of stacks. And the requirement that the configurations be coupled, ensures the equality of suffixes of stacks.

6 Comparing supercompilers

Some recent works on supercompilation have studied the influence of different aspects of supercompilation algorithms on program speedup. However, little attention has been paid to the aspects of supercompilation related to its fitness for program analysis.

Figure 18 Definitions of functions for tests

```
data List a = Nil | Cons a (List a);
data Nat = Z | S Nat;
data Boolean = True | False;
data Pair a b = P a b;

compose = λf g x → f (g x);
outl = λp → case p of { P a b → a;};
outr = λp → case p of { P a b → b;};
uncurry = λf p → case p of {P x y → f x y;};
curry = λf b c → f (P b c);
cond = λp f g a → case (p a) of {True → f a; False → g a;};
foldn = λc h x → case x of {
  Z → c;
  S x1 → h (foldn c h x1);
};
plus = foldn (λx → x) (λf y → S (f y));
foldr = λc h xs → case xs of {
  Nil → c;
  Cons y ys → h y (foldr c h ys);
};
concat = foldr Nil append;
sum = foldr Z plus;
filter = λp → foldr Nil
  (curry (cond (compose p outl) (uncurry (λx xs → Cons x xs)) outr));
iterate = λf x → Cons x (iterate f (f x));
length = foldr Z (λx y → S y);
join = λm k → foldr Nil (compose append k) m;
return = λx → Cons x Nil;
map = λf → foldr Nil (λx xs → Cons (f x) xs);
append = λxs ys → case xs of {
  Nil → ys;
  Cons x1 xs1 → Cons x1 (append xs1 ys);
};
```

In order to compare the 8 instances of the supercompiler $\mathbf{Sc_{ijk}}$, we use the following "benchmark" task: proving the equivalence of expressions [11]. All expressions used in the tests involve higher-order functions and (potentially) infinite data structures. The functions over pairs, lists and numbers used in the tasks are shown in Fig. 18. The results obtained are shown in Fig. 16. A supercompiler is considered to have passed a test if it has been able to transform both sides of the equality to the same syntactic form.

The results of the experiment, shown in Fig. 17, demonstrate that the best combination of options is the refined homeomorphic embedding, the partition of

nodes into local and global ones and the partition of expressions according to their redex. That is, additional tricks (the partition of nodes into local and global ones and the partition of expressions according to their redex) are not as powerful as the use of the refined homeomorphic embedding $\trianglelefteq_c^*$.

In order to get some insight into the peculiarities of the refined homeomorphic embedding $\trianglelefteq^*$, let us compare the behavior of the supercompilers $\mathbf{Sc}_{-++}$ and $\mathbf{Sc}_{+++}$ when they transform the expression `map f (concat xs)`. The whistle blows after a few steps of the supercompiler $\mathbf{Sc}_{-++}$:

```
case
  case xs of {
    Nil → Nil;
    Cons v54 v55 → (append v54) (foldr Nil append v55);
  }
of {
  Nil → Nil;
  Cons v56 v57 →
    (λv58 v59 → Cons (f v58) v59)
      v56 (foldr Nil (λv60 v61 → Cons (f v60) v61) v57);
}
⊴c
  case
    case v54 of {
      Nil → foldr Nil append v55;
      Cons v87 v88 → Cons v87 (append v88 (foldr Nil append v55));
    }
  of {
    Nil → Nil;
    Cons v89 v90 →
      (λv91 v92→  Cons (f v91) v92)
        v89 (foldr Nil (λv93 v94→  Cons (f v93) v94) v90);
  }
```

The embedded expression is generalized by the supercompiler $\mathbf{Sc}_{-++}$ in the following way:

```
let
v100 = case xs of {
  Nil → Nil;
  Cons v54 v55 → (append v54) (foldr Nil append v55);
}
in
case v100 of {
  Nil → Nil;
  Cons v89 v90 →
    (λv91 v92→  Cons (f v91) v92)
    v89 (foldr Nil (λv93 v94→  Cons (f v93) v94) v90);
}
```

Figure 19 `map f (concat xs)`: transformed by $\mathbf{Sc}_{-++}$

```
letrec f1 = λz →
  case z of { Nil → Nil; Cons p q → Cons (f p) (f1 q); }
in
  f1
    (letrec
      g = λx → case x of {
          Nil → Nil;
          Cons k l →
            letrec h = λy → case y of
            { Nil → g l; Cons s t → Cons s (h t); } in h k;
        }
    in g xs)
```

Figure 20 `concat (map (map f) xs)`: transformed by $\mathbf{Sc}_{+++}$

```
letrec
  g = λx →
    case x of {
      Nil → Nil;
      Cons p q →
        letrec h = λy → case y of
        { Nil → (g q); Cons s t → Cons (f s) (h t); }
        in h p;
    }
in g xs
```

However, when using the refined homeomorphic embedding the aforementioned configurations are not coupled and the transformation proceeds without over-generalization.

The residual programs produced by supercompilers $\mathbf{Sc}_{-++}$ and $\mathbf{Sc}_{+++}$ are shown in Fig. 19 and Fig. 20, respectively.

7 Discussion

As a *general idea* [24, 25], supercompilation is based on two sub-ideas: identifying (potentially) infinite branches in a partial process tree and generalizing some configurations (in order to fold the process tree into a finite graph). The main problem of the designer of a *concrete* supercompiler is the development of a reasonable generalization strategy. An aggressive strategy may produce negative effects on the depth of program analysis and transformation. On the other hand, a passive generalization strategy may result in non-termination of the supercom-

piler.

The idea to use homeomorphic embedding relation for detecting infinite branches [19] proved to be extremely fruitful and robust. The next idea is to generalize one of the configurations (either the embedded[5] one or the embedding one) by replacing it with the most specific generalization of the two configurations.

However, a concrete implementation of supercompilation has to rely on a specific homeomorphic embedding relation[6] and an algorithm of generalization.

Sørensen was the first to give a complete and self-contained description of a supercompilation algorithm for a first-order, call-by-name functional language [17]. The expressions in Sørensen's language does not contain bound variables, so that the homeomorphic embedding and the generalization algorithm do not have to deal with bound variables. The same language extended with `case`-expressions was considered in [19, 16]. Since `case`-expressions do introduce bound variables, the homeomorphic embedding had to be adapted for expressions with bound variables. However, bound and free variables are treated in the same way (as in the relation $\trianglelefteq$ considered above). Besides, the paper [19] gives no explicit description of an algorithm of generalization for expressions with bound variables. However, the algorithm of generalization for first-orders expressions is not adaptable in a straightforward way for higher-order expressions.

Recent papers on supercompilation [7, 13] also use the adapted relation $\trianglelefteq$, but do not go into details of generalization.

A difficulty in formalizing the generalization algorithm for higher-order expressions is that, in order to be correct, it must take into account a number of strict syntactic conventions concerning bound and free variables. Unfortunately, such conventions are often formulated in a rather informal and imprecise way.

The current paper presents a formal and complete algorithm for computing a most specific generalization of two HLL expressions. This algorithm is based on syntactic conventions about substitutions allowed with respect to expressions.

The current paper also demonstrates that the use of the pure embedding relation $\trianglelefteq$ for higher-order expression in an "analyzing" supercompiler results in poor residual programs, while the use of the refined homeomorphic embedding relation $\trianglelefteq^*$ results in an evident improvement in the depth of program analysis and transformation [11, 12].

A number of recent works on supercompilation have studied the influence of different aspects of supercompilation algorithms on program speedup. However, little attention has been paid to the aspects of supercompilation related to its fitness for program analysis. In this paper we have compared 8 variants of a supercompiler for a higher-order call-by-name language and have shown that the best combination of options is the use of the refined homeomorphic embedding, the partition of nodes into local and global ones and the partition of expressions according to their redex.

[5] The most popular variant.

[6] Since homeomorphic relation is a quite general notion

However, it would be interesting to study more possible variations of the supercompilation algorithm. In particular, in case of embedding, we have always generalized the upper (embedded) configuration, but it would be interesting to also consider the generalization of the lower (embedding) configuration.

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 and to Natasha for her love and patience.

References

[1] M. Abadi, L. Cardelli, P. Curien, and J. Lévy. Explicit substitutions. *Journal of functional programming*, 1(04):375–416, 1991.

[2] H. Barendregt. *The lambda calculus: its syntax and semantics*. North-Holland, 1984.

[3] H. Curry, R. Feys, and W. Craig. *Combinatory logic*, volume 1. North-Holland, 1958.

[4] N. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[5] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In *Selected Papers From the International Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 137–160, 1996.

[6] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional programming languages and computer architecture*, volume 201 of *LNCS*, pages 190–203, 1985.

[7] P. Jonsson and J. Nordlander. Positive Supercompilation for a higher order call-by-value language. In *IFL 2007*, pages 441–456, 2007.

[8] I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.

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

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

[11] I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.

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

[13] N. Mitchell and C. Runciman. A supercompiler for core haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes In Computer Science*, pages 147–164, 2008.

[14] K. H. Rose. Explicit substitution – tutorial & survey. Technical Report LS-96-3, BRICS, 1996.

[15] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1-2):193–233, 1996.

[16] M. Sørensen, R. Glück, and N. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

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

[18] M. H. Sørensen. Convergence of program transformers in the metric space of trees. In *Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 315–337, 1998.

[19] 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.

[20] M. H. Sørensen and R. Glück. Introduction to supercompilation. In *Partial Evaluation. Practice and Theory*, volume 1706 of *LNCS*, pages 246–270, 1998.

[21] A. Stoughton. Substitution revisited. *Theor. Comput. Sci.*, 59(3):317–325, 1988.

[22] The GHC Team. Haskell 2010 language report. `http://haskell.org/definition/haskell2010.pdf`, 2010.

[23] A. Tolmach, T. Chevalier, and The GHC Team. An external representation for the GHC core language. `http://www.haskell.org/ghc/docs/6.12.2/core.pdf`, 2010.

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

[25] V. F. Turchin. The algorithm of generalization in the supercompiler. In *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop*, 1988.