KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Ilya G. Klyuchnikov

Supercompiler HOSC: proof of correctness

Moscow
2010

Ilya G. Klyuchnikov. Supercompiler HOSC: proof of correctness

The paper presents the proof of correctness of an experimental supercompiler HOSC dealing with higher-order functions.

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

И.Г. Ключников. Суперкомпилятор HOSC: доказательство корректности

В работе приводится доказательство корректности экспериментального суперкомпилятора HOSC, работающего с функциями высших порядков.

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

Содержание

1 Introduction **3**

2 Input language **4**

3 Transformation relation HOSC **6**

4 The theory of improvement **9**

5 Proof of correctness **10**

5.1 Transformation relation $HOSC_0$ 11

5.2 Transformation relation $HOSC_{1/2}$ 15

5.2.1 Example 16

5.3 Transformation relation $HOSC$ 16

5.3.1 Example 18

6 Typing and correctness **19**

7 Discussion **22**

8 Related work **24**

References **25**

1 Introduction

The paper [9] describes the internal structure of HOSC, an experimental supercompiler for a higher-order call-by-name language. In this paper we prove the correctness of transformations performed by HOSC. We show that for any input program, the corresponding residual program is equivalent to an original one.

In the context of supercompilation the problem of correctness is, in a sense, ortogonal to the problem of termination. In this paper we abstract from the problem of termination and reformulate supercompilation of higher-order functions in the form of a *transformation relation*.

In [10] we have already shown that the supercompiler HOSC 1.1 terminates for any input program. Since HOSC 1.1 satisfies the transformation relation formulated in this paper, this implies the *total correctness* of the supercompiler HOSC 1.1.

In order to prove correctness of our transformation relation, we use the theory of improvement [22, 24], which is based on the concept of evaluation cost and the unfolding of a function call is used as a cost unit. Informally

speaking, an expression e_1 is an *improvement* of an expression e_0 if the computation of the expression e_1 is not more expensive than the computation of the expression e_0. An expression e_1 is a *strong improvement* of an expression e_0, if e_1 is an improvement of e_0 and e_1 and e_0 are equivalent.

Higher-order deforestation [14] is a program transformation technique which is a special case of higher-order supercompilation. The correctness of higher-order deforestation was proved in [23] using the theory of improvement. Among other things, the result of deforestation was shown to be a strong improvement over the original program.

The correctness proof for deforestation is based on the fact that, given two configurations c_1 and c_2, folding is performed only if $c_1 = con\langle f \rangle$ (so that the next reduction step is going to be an unfolding of f). The transformation relation HOSC does not impose such a restriction, folding being allowed for any pair of configurations.

Thus we begin by considering the relation $HOSC_0$, which only allows folding for configurations of the form $con\langle f \rangle$, so that $HOSC_0$ and higher-order deforestation differ only in the generalization step. The proof of the fact that, given a program, $HOSC_0$ produces a strong improvement of the original program is rather straightforward, which implies that a residual program produced by $HOSC_0$ is equivalent to the original one.

Then we prove a number of more general statements about the correctness of the transformation relations $HOSC_{1/2}$ and $HOSC$.

Finally, we consider the correctness of supercompilation in the presence of typing.

2 Input language

The supercompiler HOSC transforms programs written in the language HLL. The syntax and semantics of HLL are described in detail in [9]. Here we reproduce only definitions and notation essential for the proof.

The syntax of the untyped version of HLL is shown in Fig. 1. Until Section 6, we will consider the untyped version of HLL, ignoring type declarations and typing issues.

Before the supercompilation is started, the input program is λ-lifted, so that *letrec*-expressions, if any, are transformed into global definitions. In addition, the bindings of *let*-expressions are inlined. Thus, during supercompilation, the transformed program contains neither *letrec*- nor *let*-expressions.

Any HLL expression may be represented as either an observable or a reduction context with a redex placed into the hole of the context. The grammar of such decompositions is shown in Fig. 2.

$$
\begin{array}{ll}
prog ::= e \ \textbf{where} \ \overline{f_i = e_i;} & \text{program} \\
\\
e ::= v & \text{variable} \\
\quad \mid c \ \overline{e_i} & \text{constructor} \\
\quad \mid f & \text{function} \\
\quad \mid \lambda \ \overline{v_i} \to e & \lambda\text{-abstraction} \\
\quad \mid e_1 \ e_2 & \text{application} \\
\quad \mid \textbf{case} \ e_0 \ \textbf{of} \ \{\overline{p_i \to e_i;}\} & \text{case-expression} \\
\quad \mid \textbf{letrec} \ f = e_0 \ \textbf{in} \ e_1 & \text{local definition} \\
\quad \mid \textbf{let} \ \overline{v_i = e_i;} \ \textbf{in} \ e & \text{let-expression} \\
\quad \mid (e) & \text{parenthesized expression} \\
\\
p :: = c \ \overline{v_i} & \text{pattern}
\end{array}
$$

Figure 1: Untyped variant of the HLL language

$$
\begin{array}{lll}
obs & ::= & v \ \overline{e_i} \mid c \ \overline{e_i} \mid (\lambda v \to \ e) \\
con & ::= & \langle\rangle \mid con \ e \mid case \ con \ of \ \{\overline{p_i \to e_i;}\} \\
red & ::= & f \mid (\lambda v \to e_0) \ e_1 \mid case \ v \ \overline{e'_j} \ of \ \{\overline{p_i \to e_i;}\} \\
 & \mid & case \ c \ \overline{e'_j} \ of \ \{\overline{p_i \to e_i;}\}
\end{array}
$$

Figure 2: Decomposition of expressions

$$
\begin{array}{llll}
\mathcal{E}[\![c \ \overline{e_i}]\!] & \Rightarrow & c \ \overline{e_i} & (E_1) \\
\mathcal{E}[\![\lambda v_0 \to e_0]\!] & \Rightarrow & \lambda v_0 \to e_0 & (E_2) \\
\mathcal{E}[\![con\langle f_0\rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle unfold(f_0)\rangle]\!] & (E_3) \\
\mathcal{E}[\![con\langle(\lambda v \to e_0) \ e_1\rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle e_0\{v \ := \ e_1\}\rangle]\!] & (E_4) \\
\mathcal{E}[\![con\langle case \ c_j \ \overline{e'_k} \ of \ \{\overline{c_i \ \overline{v_{ik}} \ \to e_i;}\}\rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle e_j\{\overline{v_{jk} \ := \ e'_k}\}\rangle]\!] & (E_5) \\
\mathcal{E}[\![let \ \overline{v_i = e_i;} \ in \ e_g]\!] & \Rightarrow & \mathcal{E}[\![e_g\{\overline{v_i := e_i}\}]\!] & (E_6)
\end{array}
$$

Figure 3: Operational semantics of HLL

A *substitution* is a finite list of pairs in the form

$$\theta = \{v_1 := e_1, v_2 := e_2, \ldots, v_n := e_n\},$$

each pair binding a variable to an expression. The domain of a substitution θ is denoted as $domain(\theta)$. The application of a substitution θ to free variables of an expression e is written as $e\,\theta$.

In order to represent generalization in partial process trees, *let*-expressions

$$\mathcal{D}[\![v\ \overline{e_i}]\!] \Rightarrow [v,\ \overline{e_i}] \qquad (D_1)$$

$$\mathcal{D}[\![c\ \overline{e_i}]\!] \Rightarrow [\overline{e_i}] \qquad (D_2)$$

$$\mathcal{D}[\![\lambda v_0 \to e_0]\!] \Rightarrow [e_0] \qquad (D_3)$$

$$\mathcal{D}[\![con\langle f_0\rangle]\!] \Rightarrow [con\langle unfold(f_0)\rangle] \qquad (D_4)$$

$$\mathcal{D}[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] \Rightarrow [con\langle e_0\{v_0 := e_1\}\rangle] \qquad (D_5)$$

$$\mathcal{D}[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}\rangle]\!] \Rightarrow \left[con\langle e_j\{\overline{v_{jk} := e'_k}\}\rangle\right] \qquad (D_6)$$

$$\mathcal{D}[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!] \Rightarrow \left[v\ \overline{e'_j},\ \overline{con\langle e_i\{v\ \overline{e'_j} := p_i\}\rangle}\right] \qquad (D_7)$$

$$\mathcal{D}[\![let\ \overline{v_i = e_i;}\ in\ e]\!] \Rightarrow [e,\ \overline{e_i}] \qquad (D_8)$$

Figure 4: Driving

are used. The syntax of *let*-expressions is as follows:

$$let\ \overline{v_i = e_i;}\ in\ e_g,$$

where $\overline{v_i = e_i}$ are bindings of the corresponding substitution. Given a result of generalization represented by a *let*-expression $let\ \overline{v_i = e_i;}\ in\ e_g$, one may reconstruct the original expression by applying the substitution: $e_g\{\overline{v_i := e_i}\}$.

The operational (call-by-name) semantics of the language HLL is shown in Fig. 3. Since *let*-expressions may appear in partial process trees, we also define the operational semantics for them (Rule E_6[1]). The semantics of *letrec*-expression is defined in Section 4. If no rule in Fig. 3 can be applied during evaluation of an expression, a runtime error happens.

An expression e_2 is an *instance* of an expression e_1, $e_1 \lessdot e_2$, if there exists a substitution θ such that $e_1\theta \equiv e_2$. An operation of finding such a substitution is denoted as $\theta = e_1 \otimes e_2$.

A set of all expressions of the language HLL is denoted by $\mathcal{H}$.

3 Transformation relation HOSC

The supercompilation of a program is a two stage process. At the first stage, a partial process tree is constructed. At the second stage, a residual program is extracted from this tree.

A partial process tree is a directed tree (whose edges are denoted by $\to$) supplemented with "return" edges (denoted by $\rightrightarrows$) turning it into a directed graph.

[1]This rule is sufficient, since during supercompilation only top level *let*-expressions may appear.

$t = (e \rightarrow)$
while $unprocessedLeaf(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaf(t)$
  $t = choice\{drive(t,\beta), generalize(t,\beta), fold(t,\beta)\}$
end

Figure 5: $HOSC_{1/2}$, $HOSC$: partial process tree construction

$children(t,\ \alpha)$	Returns an ordered list of child nodes for the node α of the tree t.
$addChildren(t,\ \beta,\ es)$	Adds child nodes to the node β of the tree t and puts expressions es into them.
$replace(t,\ \beta,\ expr)$	Replaces a subtree with the root β by a single node γ such that $\gamma.expr = expr$.
$fold(t,\ \beta)$	If a node β has an ancestor node α, such that $\alpha.expr \simeq \beta.expr$, then adds a "return" edge $\beta \rightrightarrows \alpha$ to form a cycle.
$fold_0(t,\ \beta)$	If $\beta.expr = con\langle f_0\rangle$ and the node β has an ancestor node α, such that $\alpha.expr \simeq \beta.expr$, then adds a "return" edge $\beta \rightrightarrows \alpha$ to form a cycle.
$generalize(t,\ \beta)$	$replace(t,\ \beta,\ e_1)$, where $e_1 = let\ \overline{v_i = e_i;}\ in\ e_g$, $e_g\theta = e_g\{\overline{v_i := e_i}\} = \beta.expr$, θ – *any* correct substitution
$\lceil \alpha \curlywedge t \rceil$	Returns all repeat nodes of the node α, $\lceil \alpha \curlywedge t \rceil = [\overline{\beta_i}] : \beta_i \rightrightarrows \alpha$, or $\bullet$ if α is not a function node.
$\lceil \alpha \upuparrows t \rceil$	Returns a function node for the node α, $\lceil \alpha \upuparrows t \rceil = \beta : \alpha \rightrightarrows \beta$, or $\bullet$ if α is not a repeat node.
$drive(t,\alpha)$	$= addChildren(t,\ \alpha,\ \mathcal{D}[\![\alpha.expr]\!])$ - executes a driving step.
$unprocessedLeaf(t)$	Returns *any* unprocessed leaf α of the tree t, or $\bullet$, if all leaves of t are processed. A leaf is processed if it is labeled with a variable, or there is a "return" edge $\rightrightarrows$ originating from this leaf.

Figure 6: Operations on partial process tree

A process tree is produced by applying the driving rules shown in Fig. 4, which, in a sense, generalize the reduction semantics for the case of expressions with free variables. Some operations on partial process trees are presented and explained in Fig. 6.

Now we consider the transformation relation $HOSC_{1/2}$, which will be used as the basis for defining more sophisticated relations, $HOSC_0$ and $HOSC$,

$$
\begin{array}{lll}
\mathcal{C}\,[\![\alpha]\!]_{t,\Sigma} & & \\
\quad \Rightarrow \mathit{letrec}\ f' = \lambda\overline{v_i} \rightarrow (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma'})\ \mathit{in}\ f'\overline{v_i} & \text{if } \lceil \alpha \curlywedge t \rceil \neq \bullet & (C_1) \\
\qquad \text{where} & & \\
\qquad\quad \overline{v_i} = fv(\alpha.expr) & & \\
\qquad\quad \Sigma' = \Sigma \cup (\alpha,\ \{f' := \lambda\overline{v_i} \rightarrow \alpha.expr\}),\ f' \text{ is fresh} & & \\
\quad \Rightarrow f'\overline{v_i} & \text{if } \lceil \alpha \upuparrows t \rceil \neq \bullet & (C_2) \\
\qquad \text{where} & & \\
\qquad\quad f' = domain(\Sigma(\lceil \alpha \upuparrows t \rceil)),\ \overline{v_i} = fv(\alpha.expr) & & \\
\quad \Rightarrow \mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma} & \text{otherwise} & (C_3)
\end{array}
$$

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

In order to make the rules less cumbersome, the following abbreviations are used. If γ_i appears in the right-hand side, we assume that $[\overline{\gamma_i}] = children(t,\ \alpha)$, and all child nodes are processed in the same way. If γ' and γ_i' appear in the right-hand side, we assume that $[\gamma',\ \overline{\gamma_i'}] = children(t,\ \alpha)$, and either there is exactly one child node or the first child node requires special treatment.

Figure 7: $HOSC_0$, $HOSC_{1/2}$: extraction of a residual program from a tree

matching certain "design decisions" taken in the development of the supercompiler HOSC. The general idea of presenting supercompilation in terms of a program specialisation relation is due to [8].

A nondeterministic algorithm constructing partial process trees for an expression e is presented in Fig. 5. The operator *choice* nondeterministically selects and evaluates one of its arguments.

A partial process tree t is transformed into a residual program $prog'$ according to the rules given in Fig. 7.

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

Definition 1 (Transformation relation $HOSC_{1/2}$)**.** *An input program p and a residual program p' are related by the transformation relation $HOSC_{1/2}$ ($p\ HOSC_{1/2}\ p'$), if there is a partial process tree t for p that may be produced by some execution of the algorithm from Fig. 5, such that $p' = \mathcal{C}\ [\![t.root]\!]_{t,\{\}}$.*

The transformation relations $HOSC_0$ and $HOSC$ are defined later in a similar way. Sometimes, instead of $prog\ HOSC_{1/2}\ prog'$, we will write $prog' = \mathcal{SC}_{1/2}[\![prog]\!]$ (and, analogously, for $\mathcal{SC}_0[\![prog]\!]$ and $\mathcal{SC}[\![prog]\!]$).

Sometimes we will write $\mathcal{SC}_{1/2}[\![e]\!]$ instead of $\mathcal{SC}_{1/2}[\![prog]\!]$, where e is the target expression of a program $prog$.

4 The theory of improvement

Definition 2 (Context). *A context C is an expression with a hole $[\,]$ in the place of a subexpression. $C[e]$ is the expression produced by replacing the hole with the expression e.*

Definition 3 (Weak head normal form). *An HLL expression e is in weak head normal form if it is a constructor ($e = c\ \overline{e_i}$) or a λ-abstraction ($e = \lambda v \rightarrow\ e_1$).*

Definition 4 (One-step reduction). *One-step reduction $\mapsto$ is the least relation on closed expressions satisfying the rules given in the Figure 3.*

The transitive reflexive closure of $\mapsto$ is denoted as $\mapsto^*$.

Definition 5 (Convergence). *A closed expression e converges to weak head normal form w, $e \Downarrow w$, if and only if $e \mapsto^* w$.*

$e \Downarrow$ means that there exists w such that $e \Downarrow w$.

Definition 6 (Approximation). *An expression e_1 operationally approximates e_2, $e_1 \sqsubseteq e_2$, if for all contexts C, such that $C[e_1]$ and $C[e_2]$ are closed, if $C[e_1] \Downarrow$ then $C[e_2] \Downarrow$.*

Definition 7 (Equivalence). *An expression e_1 is operationally equivalent to e_2, $e_1 \cong e_2$, if $e_1 \sqsubseteq e_2$ and $e_2 \sqsubseteq e_1$.*

Definition 8 (Improvement). *An expression e_2 is an improvement of e_1, $e_1 \trianglerighteq e_2$, if for all contexts C, such that $C[e_1]$ and $C[e_2]$ are closed, if computation of $C[e_1]$ terminates using n function calls, then computation of $C[e_2]$ terminates using no more than n function calls.*

Definition 9 (Strong improvement). *An expression e_2 is a strong improvement of an expression e_1, $e_1 \trianglerighteq_s e_2$, if $e_1 \trianglerighteq e_2$ and $e_1 \cong e_2$.*

Definition 10 (Cost-equivalence). *Expressions e_1 and e_2 are cost-equivalent $e_1 \trianglelefteq\trianglerighteq e_2$, if $e_1 \trianglerighteq e_2$ and $e_2 \trianglerighteq e_1$.*

Definition 11 (Fix point operator). *The operator fix is defined as:*

$$fix = \lambda f \rightarrow f(fix\ f)$$

Definition 12 (Letrec)**.** *Letrec-expressions are translated using fix:*

$$letrec\ f = e\ in\ e' \stackrel{def}{=} (\lambda f \to e')(fix\ (\lambda f \to e))$$

Definition 13 (Transformation of definitions)**.** *Let $\{f_i = e_i;\}$ be a set of functions and $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ be a set of new functions, such that $\overline{e'_i}$ do not depend on $\overline{g_i}$, and $\overline{g_i}$ do not depend on $\overline{f_i}$. Then $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ is a transformation of $\{f_i = e_i;\}$.*

Theorem 14 (Improvement theorem [23])**.** *If $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ is a transformation of $\{f_i = e_i;\}$, such that $e_i \trianglerighteq e'_i$, then $f_i \trianglerighteq g_i$.*

Theorem 15 (Strong improvement theorem)**.** *If $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ is a transformation of $\{f_i = e_i;\}$, such that $e_i \trianglerighteq_s e'_i$, then $f_i \trianglerighteq_s g_i$.*

Theorem 16 (Local improvement theorem [23])**.** *If variables h and $\overline{x_i}$ include all free variables of both e_0 and e_1, then if*

$$letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e_0 \trianglerighteq_s letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e_1$$

then for all expressions e

$$letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e \trianglerighteq_s letrec\ h = \lambda\overline{x_i} \to e_1\ in\ e$$

Proposition 17 (Improvements [23])**.** *The following improvements hold:*

1. *If $e \trianglerighteq_s e'$, then $C[e] \trianglerighteq_s C[e']$*
2. *$con\langle case\ e\ of\ \{\overline{p_i \to e_i;}\}\rangle \trianglelefteq\!\trianglerighteq case\ e\ of\ \{\overline{p_i \to con\langle e_i\rangle;}\}$*
3. *$con\langle case\ e\ of\ \{\overline{p_i \to e_i\{z := e\};}\}\rangle \trianglerighteq_s con\langle case\ e\ of\ \{\overline{p_i \to e_i\{z := p_i\};}\}\rangle$*
4. *If $e \mapsto e'$, then $C[e] \trianglerighteq_s C[e']$*
5. *For all expressions e and substitutions θ, such that $h \notin domain(\theta)$, if $e_0 \mapsto t$, then*

   $$letrec\ h = \lambda\overline{y_i} \to t\ in\ e\{z := e_0\theta\} \trianglelefteq\!\trianglerighteq letrec\ h = \lambda\overline{y_i} \to t\ in\ e\{z := (h\ \overline{y_i})\theta\}$$

5 Proof of correctness

Let e and e' be target expressions of programs $prog$ and $prog'$. We will write $prog \trianglerighteq_s prog'$ if $e \trianglerighteq_s e'$, and $prog \cong prog'$ if $e \cong e'$.

We consider a transformation relation T to be correct if

$$prog\ T\ prog' \Rightarrow prog \cong prog'$$

$t = (e \rightarrow)$
while $unprocessedLeaf(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaf(t)$
  $t = choice\{drive(t,\beta), generalize(t,\beta), fold_0(t,\beta)\}$
end

Figure 8: $HOSC_0$: the construction of partial process trees

5.1 Transformation relation $HOSC_0$

We start by proving the correctness of the transformation relation $HOSC_0$ that only allows folding for nodes of the form $con\langle f\rangle$. This transformation relation is shown in Fig. 8 and Fig. 7. In order to extract a residual program, a partial process tree is traversed and a correspondence of function nodes and signatures of new functions is registered in the table Σ.

We assume that $\Sigma(\bullet) = \{\}$ (i.e. the signature $\Sigma(\bullet)$ is empty). We also assume that ρ_Σ is defined and

$$\rho_\Sigma = range(\Sigma)$$

Theorem 18. *For a partial process tree built according to the rules in Fig. 8, in the process of constructing a residual program according to the rules in Fig. 7, at any step the following holds:*

$$\alpha.expr \underset{\sim s}{\trianglerighteq} (\mathcal{C}[\![\alpha.expr]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

Proof. By induction on the structure of the expression $\alpha.expr$. First, consider Rule (C_3). We need to prove that

$$\alpha.expr \underset{\sim s}{\trianglerighteq} (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

We proceed by case analysis of the operation $\mathcal{C}'$.

- (C'_1) We have to show that

$$e\{\overline{v_i = e_i;}\} \underset{\sim s}{\trianglerighteq} (\mathcal{C}'[\![let\ \overline{v_i = e_i;}\ in\ e]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

  By Rule (C'_1):

$$(\mathcal{C}'[\![let\ \overline{v_i = e_i;}\ in\ e]\!]_{t,\alpha,\Sigma})\rho_\Sigma = (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}}\})\rho_\Sigma$$

  By construction of ρ_Σ:

$$(\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}}\})\rho_\Sigma = (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)(\{\overline{v_i = (\mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma)}\})$$

Thus we need to show:

$$e\{\overline{v_i = e_i;}\} \trianglerighteq_s (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)(\{\overline{v_i = (\mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma)}\})$$

By construction of the partial process tree (Rule D_8 in Fig. 4): $e = \gamma'.expr$, $e_i = \gamma'_i.expr$. By the induction hypothesis:

$$e \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma, \quad e_i \trianglerighteq_s \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma$$

Thus from Proposition 17(1) it follows:

$$e\{\overline{v_i = e_i;}\} \trianglerighteq_s (\mathcal{C}'[\![let\ \overline{v_i = e_i;}\ in\ e]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

- (C'_2) We have to show that

$$v\ \overline{e_i} \trianglerighteq_s (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}})\rho_\Sigma = (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma})$$

  By the induction hypothesis:

$$e_i \trianglerighteq_s \mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma$$

  Thus the statement to be proved follows from Proposition 17(1).

- (C'_3) We have to show that

$$c\ \overline{e_i} \trianglerighteq_s (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}})\rho_\Sigma = (c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma})$$

  By the induction hypothesis:

$$e_i \trianglerighteq_s \mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma$$

  Thus the statement to be proved follows from Proposition 17(1).

- (C'_4) We have to show that

$$\lambda v_0 \to e_0 \trianglerighteq_s (\lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma})\rho_\Sigma = \lambda v_0 \to (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)$$

  By the induction hypothesis:

$$e_0 \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma$$

  Thus the statement to be proved follows from Proposition 17(1).

- (C'_5) We have to show that

$$con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle \trianglerighteq_s (case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma} of \{\overline{p_i \to \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma};}\})\rho_\Sigma =$$

$$= case\ (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma) of \{\overline{p_i \to (\mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma);}\}$$

By construction of partial process tree (Rule D_7 in Fig. 4):

$$\gamma'.expr = v\ \overline{e'_j}, \quad \gamma'_i.expr = con\langle e_i\{v\ \overline{e'_j} := p_i\}\rangle$$

By the induction hypothesis:

$$v\ \overline{e'_j} \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma, \quad con\langle e_i\{v\ \overline{e'_j} := p_i\}\rangle \trianglerighteq_s \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma$$

Thus the statement to be proved follows from Lemmas 17(1, 2, 3).

- $(C'_6), (C'_7), (C'_8)$. In these cases $\alpha.expr \mapsto \gamma'.expr$. Thus the statement to be proved follows from the induction hypothesis and Lemma 17(4).

Now consider Rule (C_2). We heed to show that

$$\alpha.expr \trianglerighteq_s (\mathcal{C}[\![\alpha.expr]\!]_{t,\Sigma})\rho_\Sigma = (f'\overline{v_i})\rho_\Sigma$$

By construction (the operation $fold_0$ in Fig. 6):

$$(f'\overline{v_i})\rho_\Sigma = (\lambda\overline{v_i} \to \alpha.expr)\overline{v_i}$$

The statement to be proved follows from the fact:

$$(\lambda\overline{v_i} \to \alpha.expr)\overline{v_i} \trianglelefteq\trianglerighteq \alpha.expr$$

It remains to consider Rule (C_1). We need to show that

$$\alpha.expr \trianglerighteq_s (letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\ in\ f'\overline{v_i})\rho_\Sigma$$

This is equivalent to

$$\alpha.expr \trianglerighteq_s letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\ in\ f'\overline{v_i}$$

From the fact that $\alpha.expr \mapsto \gamma'.expr$ and from Lemma 17(5) it follows:

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \alpha.expr \trianglelefteq\trianglerighteq letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i}$$

Since $f' \notin fv(\alpha.expr)$:

$$\alpha.expr \trianglelefteq\trianglerighteq letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i}$$

Thus it suffices to show that

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i} \trianglerighteq_s$$

$$\trianglerighteq_s letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\ in\ f'\overline{v_i}$$

Lemma 19.

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'} \mathrel{\underset{\sim}{\lhd\rhd}} letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

Proof. Consider the expression

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'}$$

According to Rule (C_2) all free occurrences f' are in subexpressions $f'\overline{v_i'}$, Suppose there are n such occurrences – $\overline{(f'\overline{v_i})\theta_k}$, where θ_k is a renaming of variables $\overline{v_i}$. Then

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'}) = e'\{\overline{z_k := (f'\overline{v_i})\theta_k}\}$$

where $f' \notin fv(e')$. Thus,

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'} \equiv (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\{f' := \lambda\overline{v_i} \to \alpha.expr\}$$

$$\mathrel{\underset{\sim}{\lhd\rhd}} e'\{\overline{z_k := f'\overline{v_i}\theta_k}\}\rho_\Sigma\{f' := \lambda\overline{v_i} \to \alpha.expr\} \mathrel{\underset{\sim}{\lhd\rhd}} e'\{\overline{z_k := (\alpha.expr)\theta_k}\}\rho_\Sigma$$

Since $\alpha.expr \mapsto \gamma'.expr$, then from Lemma 17(5) it follows:

$$e'\{\overline{z_k := (\alpha.expr)\theta_k}\}\rho_\Sigma$$

$$\mathrel{\underset{\sim}{\lhd\rhd}} letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ e'\{\overline{z_k := f'\overline{v_i}\theta_k}\}\rho_\Sigma$$

$$\equiv letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

□

Figure 9: Auxiliary lemma

By Theorem 16 it is sufficient to show that

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \gamma'.expr \mathrel{\underset{\sim}{\rhd}}_s$$

$$\mathrel{\underset{\sim}{\rhd}}_s letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

Since $letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \gamma'.expr = \gamma'.expr$, it follows from Auxiliary Lemma 19 that it suffices to show

$$\gamma'.expr \mathrel{\underset{\sim}{\rhd}}_s (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'}$$

but this is the induction hypothesis. □

Corollary 20 (Correctness of the transformation relation $HOSC_0$)**.**

$$e \mathrel{\underset{\sim}{\rhd}}_s \mathcal{SC}_0[\![e]\!]$$

Proof. It follows from Theorem 18, since $\mathcal{SC}_0[\![e]\!] = (\mathcal{C}[\![t.root]\!]_{t,\{\}})\{\}$ where t is a partial process tree constructed according to the rules in Fig. 8. □

5.2 Transformation relation $HOSC_{1/2}$

The relation $HOSC_0$ only allows folding for configurations of the form $con\langle f\rangle$. In this section we show the correctness of the transformation relation $HOSC_{1/2}$. $HOSC_{1/2}$ allows folding for *all* configurations.

Theorem 21 (Correctness of the transformation relation $HOSC_{1/2}$)**.** *Let t be a partial process tree of an expression e built according to the rules in Fig. 5. Let $\mathcal{SC}_{1/2}[\![e]\!]$ be a residual program extracted from the tree t according to the rules in Fig. 7. Then*

$$e \cong \mathcal{SC}_{1/2}[\![e]\!]$$

Proof. In order to prove correctness of the transformation relation $HOSC_{1/2}$, we use the following trick. Let us consider *folded* configurations c_1, c_2, ... in the tree t, such that c_1, c_2, ... do not have the form $con\langle f\rangle$. For any c_i we insert a new node labeled with c'_i directly above c_i. A configuration c'_i has the form $con\langle g\rangle$, where g is a new function such that $c'_i \to c_i$. Then we replace the folding of the configurations c_i with the folding of the configurations c'_i. We look into the traces of the configurations c_1, c_2, ... and find subexpressions e_1, e_2, ... in the input program which resulted into c_1, c_2, ... We replace expressions e_1, e_2, ... with expressions $g\ \overline{e'_i}$. This may result in some parts of the input program being rewritten as supercombinators with maximal free expressions abstracted [20] (see an example in Section 5.2.1).

We repeat these steps while there are "folded" configurations not in the form $con\langle f\rangle$. Finally we get a "new" partial process tree t' and a "new" input program $prog'$.

By construction, residual programs extracted from the trees t and t' will be the same. Let us denote those programs as $prog''$. Note that, by construction, the tree complies with the transformation relation $HOSC_0$. By definition e_i is a strong improvement over $g\ \overline{e'_i}$ (since $g\ \overline{e'_i} \to e_i$), that is $prog$ is a strong improvement over $prog'$ (by Theorem 15). Thus:

$$prog' \trianglerighteq_s prog,\ prog'' = \mathcal{SC}_0[\![prog']\!]$$

Since $HOSC_0$ is an improving transformation relation, it follows:

$$prog' \trianglerighteq_s prog''$$

By the definition of strong improvement:

$$prog \cong prog''$$

On the other hand:

$$prog'' = \mathcal{SC}_{1/2}[\![prog]\!]$$

Thus

$$prog \cong \mathcal{SC}_{1/2}[\![prog]\!]$$

□

```
data Stream = S Stream;

case x of {S y1 → (S (id1 y1));} where

id1 = λx → case (id x) of { S y → S (id1 y);};
id = λx → x;
```

Figure 10: Program *prog*

```
data Stream = S Stream;

g x where

id1 = λx → g (id x);
g = λx → case x of {S y → S (id1 y);};
id = λx → x;
```

Figure 11: Program *prog*′

5.2.1 Example

Consider the simple program *prog* in Fig. 10. A partial process tree for *prog* satisfying $HOSC_{1/2}$ is shown in Fig. 13. Note that $HOSC_{1/2}$ allows folding for case-expressions. The residual program *prog*″ is shown in Fig. 12.

We build the program *prog*′ (Fig. 11) using the trick described earlier. *prog* is a strong improvement over *prog*′. When constructing *prog*′ we had to abstract subexpression `id x` from the body of the function `id1`.

The combination of t and a partial process tree t' built by $HOSC_0$ for *prog*′ is shown in Fig. 13. Repeat edges in t are dashed, return edges in t' are solid. Obviously, the residuals programs extracted from these trees are the same. So $prog'' \cong prog$.

5.3 Transformation relation $HOSC$

The rules in Fig. 7 may result in constructing recursive functions with too many arguments ("arity over-raising", see [21]). An example will be presented in the next section.

According to the rules in Fig. 7, the arity of a residual function is determined by the number of free variables in the base configuration (labeling the base node).

However, the arity of a residual function may be reduced by not taking into account the free variables in its base configuration e whose values do not

```
data Stream = S Stream;
letrec f = λp → case p of {S y → S (f y);} in f x
```

Figure 12: Program $prog''$

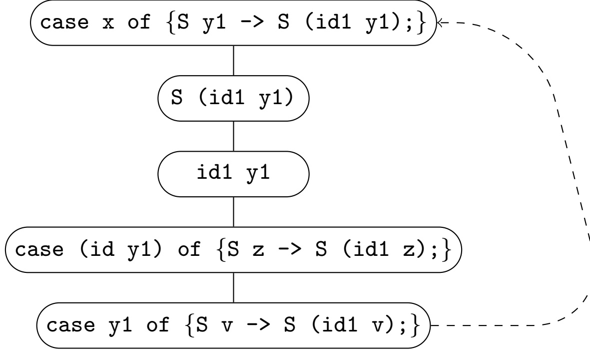

Figure 13: $HOSC_{1/2}$ partial process tree for *prog*

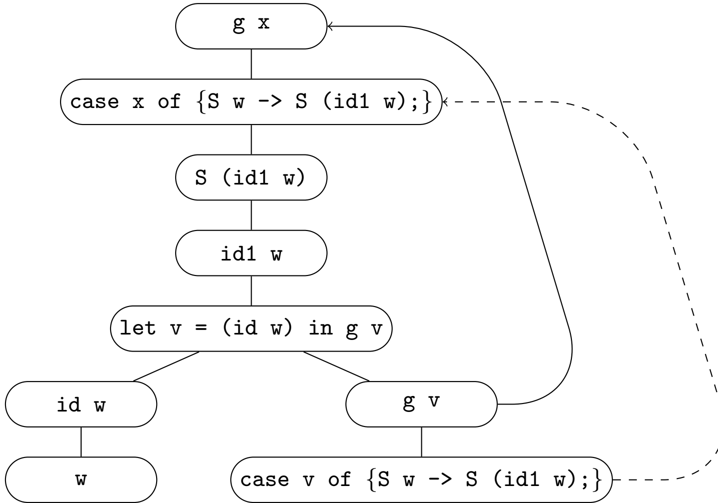

Figure 14: Transforming $HOSC_{1/2}$ tree into $HOSC_0$ tree

change in the repeat configurations. This is done in the rules in Fig 15.

The transformation relation *HOSC* is defined by the rules in Fig. 5 describing the construction of a partial process tree and by the rules in Fig. 15

$$\mathcal{C}\,[\![\alpha]\!]_{t,\Sigma}$$

$$\Rightarrow letrec\ f' = \lambda\overline{v_i} \rightarrow (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma'})\theta'\ in\ f'\overline{v_i'} \qquad \text{if } \lceil \alpha \curlywedge t \rceil \neq \bullet \quad (C_1^*)$$

where

$$\overline{[\beta_i]} = \lceil \alpha \curlywedge t \rceil,\ \theta_i = \alpha.expr \oslash \beta_i.expr,$$
$$\overline{v_i'} = domain(\bigcup \overline{\theta_i}),\ \theta' = \{\overline{v_i' := v_i}\},$$
$$\Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i}),\ f' \text{ and } \overline{v_i} \text{ are fresh}$$

$$\Rightarrow f'_{sig}\theta \qquad \text{if } \lceil \alpha \upuparrows t \rceil \neq \bullet \quad (C_2^*)$$

where

$$f'_{sig} = \Sigma(\lceil \alpha \upuparrows t \rceil),\ \theta = \lceil \alpha \upuparrows t \rceil.expr \oslash \alpha.expr$$

$$\Rightarrow \mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma} \qquad \text{otherwise} \qquad (C_3^*)$$

The operator $\mathcal{C}'$ is the same as the operator $\mathcal{C}'$ in the Fig. 7.

Figure 15: *HOSC*: extraction of a residual program from a tree

```
data List a = Nil | Cons a (List a);

app x (Cons y z)
where

app = λxs ys → case xs of {
    Nil → ys;
    Cons z zs → Cons z (app zs ys);
};
```

Figure 16: `app x (Cons y z)`: input program

describing the extraction of the residual program from the process tree.

So $\mathcal{SC}_{1/2}[\![prog]\!]$ and $\mathcal{SC}[\![prog]\!]$ are related by λ-dropping [2]:

$$\mathcal{SC}_{1/2}[\![prog]\!] \xRightarrow{\lambda-dropping} \mathcal{SC}[\![prog]\!]$$

The correctness of λ-dropping is shown in [1, 25]. That is

$$prog \cong \mathcal{SC}_{1/2}[\![prog]\!] \quad \text{and} \quad \mathcal{SC}[\![prog]\!] \cong \mathcal{SC}_{1/2}[\![prog]\!]$$

This implies the correctness of the transformation relation *HOSC*:

$$prog \cong \mathcal{SC}[\![prog]\!]$$

5.3.1 Example

Consider the concatenation of a list `x` and a non-empty list `Cons y z` (Fig. 16). A partial process tree built by $HOSC_{1/2}$ (as well as by *HOSC*) is shown

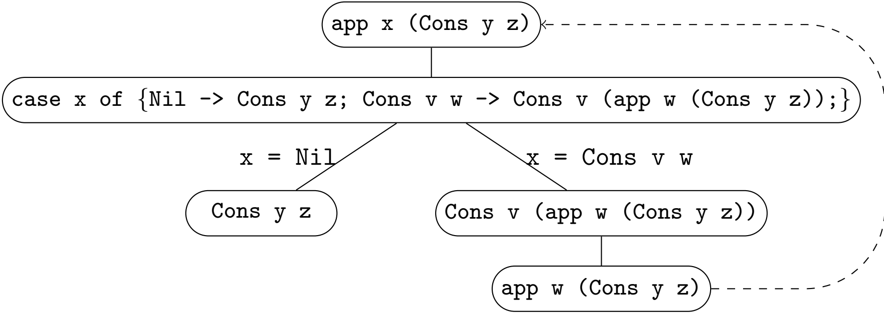

Figure 17: `app x (Cons y z)`: partial process tree

```
data List a = Nil | Cons a (List a);

letrec f = λxs v zs → case xs of {
    Nil → Cons v zs;
    Cons x1 xs1 → Cons x1 (f xs1 v zs);
}
in f x y z
```

Figure 18: `app x (Cons y z)`: residual program extracted by $HOSC_{1/2}$

in Fig. 17. The configurations `app x (Cons y z)` and `app w (Cons y z)` are folded. The extraction of the residual program from this tree (satisfying $HOSC_{1/2}$) produces a new recursive function of tree arguments, two arguments being passed without changes in the recursive calls (Fig. 18). Supercompiling the expression `app x (Cons y (Cons v w))` by $HOSC_{1/2}$ may produce a new function of four arguments, etc. By extracting the residual program from the tree in Fig. 17 according to $HOSC$, we get a function of one argument (Fig. 19).

6 Typing and correctness

Until now, we have treated HLL as an untyped language and ignored issues related to typing. All errors have been supposed to be dealt with at run-time.

But, actually, HLL is a statically typed language using the Hindley-Milner polymorphic typing system (see Fig. 20). Note that HLL expressions may contain explicit type constraints.

The semantics of the typed HLL differs from that of the untyped HLL (see

```
data List a = Nil | Cons a (List a);

letrec f = λxs → case xs of {
    Nil → Cons y z;
    Cons x1 xs1 → Cons x1 (f xs1);
}
in f x
```

Figure 19: `app x (Cons y z)`: residual program extracted by *HOSC*

$tDef ::= \text{data } tCon = \overline{dCon_i};$	type definition
$tCon ::= tn\ \overline{tv_i}$	type constructor
$dCon ::= c\ \overline{type_i}$	data constructor
$type ::= tv \mid tCon \mid type \rightarrow type \mid (type)$	type expression
$prog ::= \overline{tDef_i}\ e\ \textbf{where}\ \overline{f_i = e_i;}$	program
$e ::= e'$	implicitly typed
$\quad \mid e' :: type$	explicitly typed
$e' ::= v$	variable
$\quad \mid c\ \overline{e_i}$	constructor
$\quad \mid f$	function
$\quad \mid \lambda\ \overline{v_i} \rightarrow e$	λ-abstraction
$\quad \mid e_1\ e_2$	application
$\quad \mid \textbf{case}\ e_0\ \textbf{of}\ \{\overline{p_i \rightarrow e_i;}\}$	case-expression
$\quad \mid \textbf{letrec}\ f = e_0\ \textbf{in}\ e_1$	local definition
$\quad \mid \textbf{let}\ \overline{v_i = e_i;}\ \textbf{in}\ e$	let-expression
$\quad \mid (e)$	parenthesized expression
$p ::= c\ \overline{v_i}$	pattern

Figure 20: Typed variant of the language HLL

Fig. 3) only in one point: a program to be executed is required to be typable. Before executing a program, the interpreter type-checks it. If the program contains typing errors, the interpreter returns an error at once.

The subtle point is that, if we consider only implicitly typed programs (i.e. without explicit type constraints), the transformation relation *HOSC* is *not* correct.

Some types inferred for a residual program may be more general than the corresponding types inferred for the input program. Thus, supercompilation

```
data Bool = True | False;
data U = MkU (U → Bool);

russel (MkU russel) where

russel = λu → case u of {MkU p → p u;};
```

Figure 21: `russel (MkU russel)`: input program

```
data Bool = True | False;
data U = MkU (U → Bool);
(letrec f=f in f)
```

Figure 22: `russel (MkU russel)`: implicitly typed residual program

may happen to extend the domain of a program being transformed, because a residual program may accept inputs rejected by the original program as ill-typed.

Consider the program in Fig. 21. The inferred type of the target expression is

```
(russel (MkU russel)) :: Bool
```

This program may be supercompiled into the program shown in Fig. 22. The inferred type for the residual expression is

```
(letrec f=f in f) :: a
```

Obviously, the inferred type of the residual expression is more general than that of the original expression. It also means that there are contexts in which the original expression is ill-typed, while the residual expression is well-typed.

The fact that the Hindley-Milner type system with implicit typing does not preserve types in the case of β-reduction has been taken notice of in the literature [18].

Since supercompilation involves β-reduction, special care should be taken to prevent it from changing the typing properties of programs under transformation. Fortunately, the problem can be solved by inserting explicit type constraints into residual programs.

Consider a residual expression e' and its original expression e. Suppose $x_1, \ldots, x_n$ are the free variables in e. Let t be the type inferred for e, and t_i be the type inferred for x_i.

Now, let us replace each free occurrence of x_i in e' with $x_i :: t$ to produce e'', and add the explicit type constraint to the whole expression: $e'' :: t$. The

```
data Bool = True | False;
data U = MkU (U → Bool);
(letrec f=f in f) :: Bool
```

Figure 23: `russel (MkU russel)`: explicitly typed residual program

```
data Stream = S Stream;

case (id x) of {S y1 → (S (id1 y1));} where

id1 = λx → case (id x) of { S y → S (id1 y);};
id = λx → x;
```

Figure 24: Program $prog_1$

result is that e and $e'' :: t$ are interchangeable in any context. Namely, in any context C, $C[e]$ and $C[e'' : t]$ are both either well-typed or ill-typed.

More exactly, there remains a subtle case in which some free variables appearing in the original expression e may disappear from the residual expression e'. However, to ensure correctness, we may reintroduce the missing free variables into e' by artificially inserting into e' some λ-abstractions and applications, just to pass typing information about these variables.

For example, Fig. 23 presents an explicitly typed residual program, which is strictly equivalent to the original program shown in Fig. 21.

7 Discussion

For the sake of brevity, let H denote the *supercompiler* described in [9, 10], let $H_{1/2}$ denote the supercompiler that is the same as H, except for it does not perform λ-dropping, and let H_0 denote the supercompiler that is the same as $H_{1/2}$, except for it only folds configurations of the form $con\langle f \rangle$.

As compared to H_0, the supercompiler $H_{1/2}$ is more powerful at "normalizing" expressions, which is essential if supercompilation is used for proving the equivalence of expressions [11]. Consider the program $prog_1$ in Fig. 24. The target expression of the program $prog_1$ is reduced in two steps into the target expression of the program *prog*. The supercompiler $HOSC_{1/2}$ will transform both *prog* and $prog_1$ into the same program $prog''$ (Fig. 12). In contrast, the supercompiler $HOSC_0$ produces different programs: $prog''$ (Fig. 12) for $prog_1$ and $prog'_1$ (Fig. 25) for *prog*.

Note that $\mathcal{H}_{1/2}[\![e]\!]$ and $\mathcal{H}[\![e]\!]$ are not guaranteed to be improvements over

```
data Stream = S Stream;

case x of {
    S z → S (letrec f = λp → case p of {S y → S (f y);} in f z);
}
```

Figure 25: Program $prog'_1$

```
data List a = Nil | Cons a (List a);

letrec f = λxs vs → case xs of {
    Nil → vs;
    Cons x1 xs1 → Cons x1 (f xs1 vs);
}
in f x ys
```

Figure 26: $\mathcal{H}_{1/2}[\![app\ x\ ys]\!]$

e. This is not good for program optimization, but may be useful for program analysis ([11]).

As compared to $H_{1/2}$, the supercompiler H has a greater tendency for preserving "isomorphism" between two programs.

Consider the expressions `app x ys` and `app x (Cons y z)` (the function `app` is defined in Fig. 16). These expressions are related via a substitution:

$$app\ x\ (Cons\ y\ z) = (app\ x\ ys)\{ys := (Cons\ y\ z)\}$$

The corresponding residual expressions produced by the supercompiler H (Fig. 19 and Fig. 27) are related via the same substitution:

$$\mathcal{H}[\![app\ x\ (Cons\ y\ z)]\!] = \mathcal{H}[\![app\ x\ ys]\!]\{ys := (Cons\ y\ z)\}$$

However, the supercompiler $H_{1/2}$ does not preserve this relation between the two expressions (the results of supercompilation by $H_{1/2}$ are in Fig. 18 and Fig. 26):

$$\mathcal{H}_{1/2}[\![app\ x\ (Cons\ y\ z)]\!] \neq \mathcal{H}_{1/2}[\![app\ x\ ys]\!]\{ys := (Cons\ y\ z)\}$$

The two aforementioned properties (the tendency towards normalization and the tendency for preserving "isomorphisms") are of importance when the supercompiler *HOSC* is used for proving improvement lemmas in a higher-level supercompiler [12].

```
data List a = Nil | Cons a (List a);

letrec f = λxs → case xs of {
    Nil → ys;
    Cons x1 xs1 → Cons x1 (f xs1);
}
in f x
```

Figure 27: $\mathcal{H}[\![app\ x\ ys]\!]$

8 Related work

A fundamental part of supercompilation (driving) was originally described as a system of equivalent transformations for REFAL [27], a first-order call-by-value functional language. Other components of supercompilation were formulated later [28, 29]. At present, the most advanced REFAL supercompiler is SCP4 [19].

SCP4 may extend the domain of programs it transforms, which may be useful for the purposes of program optimization. More specifically, in some cases a residual program produced by SCP4 terminates and returns a result, while the original program does not terminate (or terminates abnormally without returning a result). This is due to the fact that REFAL is an applicative (call-by-value) language, while driving in SCP4 interprets programs in a lazy manner. A proof of partial correctness of SCP4 has not been published yet.

Sørensen considers a supercompiler for a simpler *first-order* call-by-name language in his Master's work [26] and proves correctness of the described supercompiler.

Mitchell described a supercompiler Supero for a subset of Haskell [17, 16]. A proof of correctness of Supero has not been published yet.

A supercompiler for a higher-order call-by-value language is described by Jonsson and Nordlander [4]. The correctness of this supercompiler has been proved [6].

Recently a supercompilation for a variant of system F was presented by Mendel-Gleason and Hamilton [15] and proved to be correct using bisimulation [3].

Krustev verified a simple supercompiler mechanically [13].

The contribution of this work is as follows:

- The supercompiler *HOSC* is described at an abstract level: as a transformation relation, and the correctness proof of this transformation relation is given. Usually, supercompilation is considered at a more concrete and deterministic level [17, 4, 15].

- Unlike supercompilers [17, 4], *HOSC* is allowed to perform folding of configurations of any kind. The correctness of such folding is proved. The ability to fold any configurations increases the “normalization poser” of supercompilation. And this is of importance, when supercompilation is used for program analysis.

- As was pointed out in [7], the static argument transformation (generating *letrec*-expressions with free variables) may be useful for program optimization, but correctness of such transformation was not proved. We have shown that generation of *letrec*-expressions with free variables is also useful when supercompilation is used for program analysis, and proved that, in the case of *HOSC*, such generation is correct.

- Some supercompilers assume the source programs to obey the Hindley-Milner typing discipline [17, 5]. We have shown that a supercompiler may extend the domain of programs, as the inferred types in a residual program may be more general than those in the source program. However, typing properties can be preserved by introducing explicit type constraints in residual programs.

- The correctness of the transformation relations $HOSC_0$, $HOSC_{1/2}$ and *HOSC* was shown by using the theory of improvements. So we were able to show that a residual program defined by the relation $HOSC_0$ is always more efficient (in terms of the number of function calls) than the original one. In contrast, in [15] supercompilation is considered from the perspective of the bisimulation theory, so it is difficult to compare the efficiency of the original and residual programs. Also, while we have dealt with a sound type system (Hindley-Milner), [15] considers a potentially unsound variant of system F.

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] O. Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In *FLOPS ’99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *LNCS*, pages 241–250, London, UK, 1999. Springer-Verlag.

[2] O. Danvy and U. P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theor. Comput. Sci.*, 248(1-2):243–287, 2000.

[3] A. D. Gordon. Bisimilarity as a theory of functional programming. Minicourse. Technical Report NS-95-3, BRICS, University of Cambridge Computer Laboratory, 1995.

[4] P. Jonsson and J. Nordlander. Positive Supercompilation for a higher order call-by-value language. *ACM SIGPLAN Notices*, 44(1):277–288, 2009.

[5] P. A. Jonsson. Positive supercompilation for a higher-order call-by-value language. Master's thesis, Luleå University of Technology, 2008.

[6] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher order call-by-value language. extended proofs. Technical Report 17, Luleå University of Technology, 2008.

[7] P. A. Jonsson and J. Nordlander. Supercompiling overloaded functions. submitted to ICFP 2009, 2009.

[8] A. V. Klimov. A program specialization relation based on supercompilation and its properties. In *First International Workshop on Metacomputation in Russia*, 2008.

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

[10] I. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. Preprint 21, 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] D. Krustev. A simple supercompiler formally verified in Coq. In *Second International Workshop on Metacomputation in Russia*, 2010.

[14] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 154–165. Springer, 1992.

[15] G. E. Mendel-Gleason and G. W. Hamilton. Equivalence in supercompilation and normalisation by evaluation. In *Second International Workshop on Metacomputation in Russia*, 2010.

[16] N. Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, 2008.

[17] 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, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] B. Monsuez. Polymorphic typing for call-by-name semantics. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*, pages 156–169, London, UK, 1993. Springer-Verlag.

[19] A. P. Nemytykh. The supercompiler SCP4: General structure. In *PSI 2003*, volume 2890 of *LNCS*, pages 162–170. Springer, 2003.

[20] S. L. Peyton Jones. An introduction to fully-lazy supercombinators. In *Combinators and Functional Programming Languages*, volume 242 of *LNCS*, pages 175–206. Springer, 1986.

[21] S. A. Romanenko. Arity raiser and its use in program specialization. In *ESOP '90*, volume 432 of *LNCS*, pages 341–360. Springer, 1990.

[22] D. Sands. Operational theories of improvement in functional languages. In *Proceedings of the Fourth Glasgow Workshop on Functional Programming*, 1991.

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

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

[25] U. P. Schultz. Implicit and explicit aspects of scope and block structure. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, 1997.

[26] M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Københavns Universitet, Datalogisk Institut, 1996.

[27] V. Turchin. Equivalent transformations of recursive functions defined in Refal. In *Proceedings of the symposium on Theory of Languages and Methods of Constructing of Programming Systems*, pages 31–42, 1972.

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

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