KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Ilya G. Klyuchnikov

Supercompiler HOSC 1.0: under the hood

Moscow
2009

Ilya G. Klyuchnikov. Supercompiler HOSC 1.0: under the hood

The paper describes the internal structure of HOSC, an experimental supercompiler dealing with programs written in a higher-order functional language. A detailed and formal account is given of the concepts and algorithms the supercompiler is based upon.

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

Илья Г. Ключников. Суперкомпилятор HOSC 1.0: внутренняя структура

В работе описана внутренняя структура экспериментального суперкомпилятора HOSC. Дано полное описание всех существенных понятий и алгоритмов суперкомпилятора, работающего с функциональным языком высшего порядка.

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

Contents

1 Introduction

Supercompilation is a program transformation technique, suggested by V.F. Turchin in the early 1970-s [21, 22].

This paper describes the specializer HOSC, an experimental supercompiler dealing with programs written in a simple higher-order functional language.

A number of supercompilers for higher-order functional languages have already being described in the literature [8, 10, 11, 12, 15, 14]. However, the published descriptions of concepts and algorithms related to higher-order supercompilation suffer from being incomplete: a number of essential details are either omitted or presented in sketchy and informal style.

The goal of the present work is to provide a *detailed* and *formal* description of the concepts and algorithms the supercompiler HOSC is based upon. The source code of HOSC is publicly available and can be found at `http://hosc.googlecode.com`.

2 Notation

For the sake of conciseness and readability we use overlining to denote constructs that may be repeated, so that $\overline{A_i}$ means $A_0 A_1 \dots A_n$, if A_0 is defined, or $A_1 \dots A_n$ otherwise, where n is determined by the context. Thus, an overlined construct should contain parts with subscripts and, in a sense, is similar to a list comprehension. The doubly overlined text is unfolded "inside out" with respect to overlines and "from left to right" with respect to indices. If the overlined text is found inside the parentheses denoting a set or a list ({}

$tDef$ ::= 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
$\mid f$	function
$\mid \lambda\ \overline{v_i} \to e$	λ-abstraction
$\mid e_1\ e_2$	application
$\mid$ **case** e_0 **of** $\{\overline{p_i \to e_i;}\}$	case-expression
$\mid$ **letrec** $f = \lambda\ \overline{v_i} \to e_0$ **in** e_1	local function
$\mid (e)$	parenthesized expression
$p ::= c\ \overline{v_i}$	pattern

Figure 1: HLL grammar

or [], respectively), we assume that delimiters (commas) are inserted correctly during unfolding. For example:

$$
\begin{array}{l}
let\ \overline{v_i = e_i};\ in\ e \Rightarrow let\ v_1 = e_1;\ v_2 = e_2;\ \ldots;\ v_n = e_n;\ in\ e \\
[\gamma,\ \overline{\gamma_i}] \Rightarrow [\gamma,\ \gamma_1,\ \gamma_2,\ \ldots,\ \gamma_n] \\
case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} \Rightarrow \\
case\ e_0\ of\ \{c_1\ \overline{v_{1k}} \to e_1;\ \ldots;\ c_n\ \overline{v_{nk}} \to e_n;\} \Rightarrow \\
case\ e_0\ of\ \{c_1\ v_{11}\ \ldots\ v_{1m_1} \to e_1;\ \ldots;\ c_n\ v_{n1}\ \ldots\ v_{nm_n} \to e_n;\}
\end{array}
$$

3 Input language

HOSC transforms programs written in HLL, a simple higher-order lazy language. HLL is statically typed using the Hindley-Milner polymorphic typing system [4].

A program in HLL consists of a number of data type definitions, an expression to be evaluated and a set of function definitions.

A left-hand side of a data type definition is a type name (more precisely, a type constructor name) followed by a list of type variables. The right-hand side consists of one or more constructor declarations (delimited by a vertical bar, $\overline{dCon_i} \Rightarrow dCon_1 \mid \ldots \mid dCon_n$).

The grammar of HLL is shown in Fig. 1. An expression is either a variable, a constructor, a λ-abstraction, an application, a case-expression, a local func-

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

(compose (map f)(map g)) xs where

compose = λf g x → f (g x);

map = λf xs →
    case xs of {
        Nil → Nil;
        Cons x1 xs1 → Cons (f x1) (map f xs1);
    };
```

Figure 2: A simple HLL program

tion definition or an expression in parenthesis. A function definition binds a *global* variable to a λ-abstraction.

The target expression e in a program *prog* may contain free variables. All local variables in global function definitions should be bound.

An expression e_0 within a case expression is called a selector, and expressions $\overline{e_i}$ are called branches. We require *all* constructors of a corresponding data type to be enumerated within a case expression, so that patterns in the case expression are *non-overlapping* and *exhaustive*.

We will use two different notations for applications: $e_1\ e_2$ and $e_0\ \overline{e_i}$. In the first case, an expression e_1 may be any well-typed expression. In the second case, we require the list of arguments to be non-empty and the expression e_0 not to be an application.

An example of a simple HLL program is shown in Fig. 2.

3.1 Basic definitions

The following definitions will be used throughout the whole paper.

3.1.1 Free and bound variables

A variable is bound if it is either (1) the argument of an enclosing λ-abstraction or (2) is defined in the pattern of an enclosing branch of a case-expression. Variables that are not bound are considered to be free. To avoid problems caused by possible name clashes, we require all variable names to be unique in the following sense: for any variable v and expression e *all* occurrences of the variable v in the expression e must be either free or bound.

3.1.2 Equal expressions

Expressions e_1 and e_2 are considered to be equal, if they differ only in the names of bound variables and there is a one-to-one correspondence between the bound variables of e_1 and e_2. The equality of expressions e_1 and e_2 is denoted as $e_1 \equiv e_2$.

3.1.3 Free variables

$fv[\![e]\!]$ denotes the set of the free variables of the expression e and is defined inductively as follows:

$$
\begin{array}{ll}
fv[\![v]\!] & = \{v\} \\
fv[\![c\ \overline{e_i}]\!] & = \bigcup fv[\![e_i]\!] \\
fv[\![\lambda v \rightarrow e]\!] & = fv[\![e]\!] \setminus \{v\} \\
fv[\![e_1\ e_2]\!] & = fv[\![e_1]\!] \cup fv[\![e_2]\!] \\
fv[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \rightarrow e_i;}\}]\!] & = fv[\![e_0]\!] \cup (\bigcup fv[\![e_i]\!] \setminus \{\overline{v_{ik}}\}) \\
fv[\![letrec\ f = e_1\ in\ e_2]\!] & = fv[\![e_1]\!] \cup fv[\![e_2]\!] \setminus \{f\}
\end{array}
$$

3.1.4 Bound variables

$bv[\![e]\!]$ denotes the set of the bound variables of the expression e and is defined inductively as follows:

$$
\begin{array}{ll}
bv[\![v]\!] & = \{\} \\
bv[\![c\ \overline{e_i}]\!] & = \bigcup bv[\![e_i]\!] \\
bv[\![\lambda v \rightarrow e]\!] & = bv[\![e]\!]\ \cup\ \{v\} \\
bv[\![e_1\ e_2]\!] & = bv[\![e_1]\!] \cup bv[\![e_2]\!] \\
bv[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \rightarrow e_i;}\}]\!] & = bv[\![e_0]\!] \cup \bigcup bv[\![e_i]\!] \cup \bigcup v_{ik} \\
bv[\![letrec\ f = e_1\ in\ e_2]\!] & = bv[\![e_1]\!] \cup bv[\![e_2]\!]\ \cup\ \{f\}
\end{array}
$$

3.1.5 Substitution

A substitution is a finite list of pairs of the form

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

each pair binding a variable v_i to its value e_i.

The domain of a substitution θ is denoted by $domain(\theta)$ and is defined as follows:

$$domain(\{v_1 := e_1, v_2 := e_2, \ldots, v_n := e_n\}) = \{v_1,\ v_2,\ \ldots,\ v_n\}$$

3.1.6 Application of a substitution

$e\ \theta$ denotes the result of applying a substitution θ to an expression e, which, informally speaking, is obtained in the following way.

- All free variables of expression e, which are in $domain(\theta)$, are replaced by their values from θ.
- All other parts of the expression e remain unchanged.

To avoid problems with variable name clashes, we require the domain of θ not to contain bound variables of e, i.e.

$$domain(\theta) \cap bv[\![e]\!] = \emptyset$$

More formally, $e\ \theta$ is defined as follows:

$$\begin{array}{lcll}
v\theta & = & e & \text{if } v := e \in \theta \\
 & = & v & \text{otherwise} \\
(c\ \overline{e_i})\theta & = & c\ \overline{(e_i\theta)} & \\
(\lambda v \to e)\theta & = & \lambda v \to (e\theta) & \\
(e_1\ e_2)\theta & = & (e_1\theta)\ (e_2\theta) & \\
(case\ e_0\ of\ \{\overline{p_i\ \to e_i;}\})\theta & = & case\ (e_0\theta)\ of\ \{\overline{p_i\ \to (e_i\theta);}\} & \\
(letrec\ f = e_1\ in\ e_2)\theta & = & letrec\ f = (e_1\theta)\ in\ (e_2\theta) &
\end{array}$$

3.1.7 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 $e_1 \oslash e_2$.

3.1.8 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.

3.1.9 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 $e_1 \frown e_2$.

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

Figure 3: Observables, contexts, redexes

3.1.10 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 $(e'_g, \theta'_1, \theta'_2) \in e_1 \frown e_2$ 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.

3.2 λ-lifting

Note that the construction `where` is no more than syntactic sugar, since the global function definitions may always be transformed into `letrec`-s and moved to the target expression. Thus, any program written in HLL may always be replaced by an equivalent program consisting of type definitions and a *self-sufficient* target expression. In some cases it is more convenient to deal with such programs. For example, the problem of checking the equivalence of two programs is reduced to checking the equivalence of two expressions.

However, in the process of supercompilation, global function definitions are easier to deal with than `letrec`-expressions. For this reason, at the first stage of supercompilation, HOSC removes all `letrec`-expressions using the well-known method of λ-lifting [9].

Thus, in the following, we assume that source HLL programs do not contain `letrec`-expressions. (Ironically, residual programs constructed by HOSC never contain global function definitions.)

3.3 Observables, contexts, redexes

The syntax of observables, redexes and contexts is shown in Fig. 3.

An *observable* is either:

- a local variable,
- a local variable application,
- a constructor,
- a λ-abstraction.

$$
\begin{array}{lcl}
\mathcal{E}[\![c\ \overline{e_i}]\!] & \Rightarrow & c\ \overline{e_i} \\
\mathcal{E}[\![\lambda v_0 \to e_0]\!] & \Rightarrow & \lambda v_0 \to e_0 \\
\mathcal{E}[\![con\langle f_0 \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle unfold(f_0) \rangle]\!] \\
\mathcal{E}[\![con\langle (\lambda v \to e_0)\ e_1 \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle e_0\{v\ :=\ e_1\} \rangle]\!] \\
\mathcal{E}[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\} \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle e_j\{\overline{v_{jk}\ :=\ e'_k}\} \rangle]\!]
\end{array}
$$

Figure 4: Operational semantics of HLL (call by name)

A *redex* is either:

- a global variable,
- a λ-abstraction application,
- a case-expression with an observable as a selector.

A *context* is an "expression with a hole". It is either:

- a hole (empty context) $\langle\rangle$,
- a context application,
- a case-expression with a context as a selector.

If *con* is a context, then $con\langle e \rangle$ denotes the result of replacing the hole in *con* with e. Essentially, the hole $\langle\rangle$ is used as a meta-variable.

It can be shown that any HLL-expression e is either an observable or can be represented as a context *con* whose hole is replaced with a redex r, so that $e = con\langle r \rangle$. This fact is known as the *unique decomposition* property.

3.4 Operational semantics

HLL operational semantics is defined as normal-order reduction to a weak-head normal form [3] (Fig. 4). Note that operational semantics is only defined for expressions *without* free variables.

4 Homeomorphic embedding

In the definition of the "classical" homeomorphic embedding [13, 6, 17, 18] all variables are supposed to be free and considered to be embedded into each other. However, bound variables may appear in HLL expressions, for which reason HOSC takes a more subtle approach to the embedding of variables:

- a *free* variable is embedded into a *free* variable,
- a *bound* variable is embedded into a *corresponding bound* variable,
- a *global* variable v is embedded into *itself* only.

The correspondence between bound variables is recorded in a table ρ:

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

When two λ-abstractions are coupled, the correspondence between their arguments is recorded in ρ. When two case-expressions are coupled, the correspondence between their pattern variables is also recorded in ρ.

When checking whether an expression e can dive into the body of a λ-abstraction $\lambda v \to e_1$, the bound variable v does not correspond to any variables in e. So the pair $(\bullet, v)$ is put into ρ, where $\bullet$ is a special pseudo-variable (placeholder) for a missing variable.

Checking for an expression e diving into a branch of a case-expression is performed in the same manner - we remember that pattern variables do not correspond to any variables in e.

4.1 Extended homeomorphic embedding

Let ρ be a table recording the correspondence of bound variables. Then *extended homeomorphic embedding* relation $\trianglelefteq|_\rho$ for expressions e' and e'' (constrained by ρ) is defined inductively as follows:

$$e' \trianglelefteq e'' \,|_\rho \qquad \text{if } e' \trianglelefteq_v e'' \,|_\rho \text{ or } e' \trianglelefteq_d e'' \,|_\rho \text{ or } e' \trianglelefteq_c e'' \,|_\rho$$

Thus we distinguish the following kinds of embedding:

- $e' \trianglelefteq_v e'' \,|_\rho$ – embedding of variables
- $e' \trianglelefteq_d e'' \,|_\rho$ – diving
- $e' \trianglelefteq_c e'' \,|_\rho$ – coupling

We will also use the following abbreviations:

$$\begin{array}{lcl} e' \trianglelefteq e'' & \stackrel{\text{def}}{=} & e' \trianglelefteq e'' \,|_{\{\}} \\ e' \trianglelefteq_v e'' & \stackrel{\text{def}}{=} & e' \trianglelefteq_v e'' \,|_{\{\}} \\ e' \trianglelefteq_d e'' & \stackrel{\text{def}}{=} & e' \trianglelefteq_d e'' \,|_{\{\}} \\ e' \trianglelefteq_c e'' & \stackrel{\text{def}}{=} & e' \trianglelefteq_c e'' \,|_{\{\}} \end{array}$$

4.2 Embedding of variables

$$
\begin{array}{rll}
f & \trianglelefteq_v f \mid_\rho & \\
v_1 & \trianglelefteq_v v_2 \mid_\rho & \text{if } (v_1,\ v_2) \in \rho \\
v_1 & \trianglelefteq_v v_2 \mid_\rho & \text{if } v_1 \notin \mathit{domain}(\rho) \text{ and } v_2 \notin \mathit{range}(\rho)
\end{array}
$$

4.3 Diving

For an expression e to be able to dive into another expression e' under constraints ρ, we require that

$$\forall v \in fv(e) : v \notin \mathit{domain}(\rho)$$

The rationale behind this requirement will be given in the next section. The diving relation is defined as follows:

$$
\begin{array}{rll}
e & \trianglelefteq_d c\ \overline{e_i} \mid_\rho & \text{if } e \trianglelefteq e_i \mid_\rho \text{ for some } i \\
e & \trianglelefteq_d \lambda v_0 \to e_0 \mid_\rho & \text{if } e \trianglelefteq e_0 \mid_{\rho \cup \{(\bullet, v_0)\}} \\
e & \trianglelefteq_d \overline{e_i} \mid_\rho & \text{if } e \trianglelefteq e_i \mid_\rho \text{ for some } i \\
e & \trianglelefteq_d \mathit{case}\ e_0\ \mathit{of}\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} \mid_\rho & \text{if } e \trianglelefteq e_0 \mid_\rho \\
e & \trianglelefteq_d \mathit{case}\ e_0\ \mathit{of}\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} \mid_\rho & \text{if } e \trianglelefteq e_i \mid_{\rho \cup \{\overline{(\bullet, v_{ik})}\}} \text{ for some } i
\end{array}
$$

4.4 Coupling

$$
\begin{array}{ll}
c\ \overline{e'_i} \trianglelefteq_c c\ \overline{e''_i} \mid_\rho & \text{if } \forall i : e'_i \trianglelefteq e''_i \mid_\rho \\
\lambda v_1 \to e_1 \trianglelefteq_c \lambda v_2 \to e_2 \mid_\rho & \text{if } e_1 \trianglelefteq e_1 \mid_{\rho \cup \{(v_1, v_2)\}} \\
e'\ \overline{e'_i} \trianglelefteq_c e''\ \overline{e''_i} \mid_\rho & \text{if } e' \trianglelefteq e'' \mid_\rho \text{ and } \forall i : e'_i \trianglelefteq e''_i \mid_\rho \\
\mathit{case}\ e'\ \mathit{of}\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \trianglelefteq_c \mathit{case}\ e''\ \mathit{of}\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\} \mid_\rho & \\
\qquad \text{if } e' \trianglelefteq e'' \mid_\rho \text{ and } \forall i : e'_i \trianglelefteq e''_i \mid_{\rho \cup \{\overline{(v'_{ik}, v''_{ik})}\}} &
\end{array}
$$

Note that in the third rule (diving of applications) e' and e'' themselves are not applications! This ensures that a global variable can be embedded into itself only.

4.5 Examples

Here are some examples of embedding and non-embedding (data types and global functions are taken from the programs presented in Fig. 2 and Fig. 9):

- $x \trianglelefteq y$
- $\mathit{map} \not\trianglelefteq \mathit{compose}$

- $map\ f \not\trianglelefteq compose\ map\ f$
- $map\ f \trianglelefteq_c map\ (compose\ f\ g)$
- $\lambda x \to Nil \trianglelefteq_c \lambda x \to Cons\ a\ Nil$
- $\lambda x \to Nil \not\trianglelefteq \lambda x \to Cons\ x\ Nil$
- $case\ x\ of\ \{Z \to x;\ S\ b \to f\ b;\} \trianglelefteq_c case\ x\ of\ \{Z \to x;\ S\ b \to (f\ g)\ b;\}$
- $case\ x\ of\ \{Z \to x;\ S\ b \to f\ b;\} \not\trianglelefteq case\ x\ of\ \{Z \to x;\ S\ b \to f\ (g\ b);\}$

5 Generalization

The specializer HOSC described in the paper never tries to find a generalization for two expressions e_1 and e_2, unless one of them is embedded into another by coupling. For this reason, we describe an algorithm for finding a most specific generalization (MSG) dealing only with expressions e_1 and e_2, such that $e_1 \trianglelefteq_c e_2$.

5.1 Most specific generalization of coupled expressions

A most specific generalization of coupled expressions e_1 and e_2 (that is $e_1 \trianglelefteq_c e_2$) is computed by exhaustively applying the following rewrite rules (common functor rule and common sub-expressions rule) to the initial trivial generalization $(v, \{v := e_1\}, \{v := e_2\})$. The requirement from the subsection 4.3 comes from the fact that θ_1 and θ_2 should be correct substitutions - that is, substitutions operating on free variables. This requirement guarantees that substitutions will be correct, if $e_1 \trianglelefteq_c e_2$.

5.2 Common functor rule

5.2.1 Variable

$$\left(\begin{array}{c} e \\ \{v_1 := v_2\} \cup \theta_1 \\ \{v_1 := v_2\} \cup \theta_2 \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v_1 := v_2\} \\ \theta_1 \\ \theta_2 \end{array}\right)$$

5.2.2 Constructor

$$\left(\begin{array}{c} e \\ \{v := c\ \overline{e'_1}\} \cup \theta_1 \\ \{v := c\ \overline{e''_1}\} \cup \theta_2 \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v := c\ v_1\ \ldots\ v_n\} \\ \{\overline{v_i := e'_1}\} \cup \theta_1 \\ \{\overline{v_i := e''_1}\} \cup \theta_2 \end{array}\right)$$

5.2.3 λ-abstraction

$$\left(\begin{array}{c} e \\ \{v := \lambda v' \to e'\} \cup \theta_1 \\ \{v := \lambda v'' \to e''\} \cup \theta_2 \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v := \lambda v_0 \to v_1\} \\ \{v_1 := e'\{v' := v_0\}\} \cup \theta_1 \\ \{v_1 := e''\{v'' := v_0\}\} \cup \theta_2 \end{array}\right)$$

5.2.4 Application

$$\left(\begin{array}{c} e \\ \{v := e'_0\ \overline{e'_i}\} \cup \theta_1 \\ \{v := e''_0\ \overline{e''_i}\} \cup \theta_2 \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v := v_0\ \overline{v_i}\} \\ \{v_0 := e'_0,\ \overline{v_i := e'_i}\} \cup \theta_1 \\ \{v_0 := e''_0,\ \overline{v_i := e''_i}\} \cup \theta_2 \end{array}\right)$$

5.2.5 Case-expression

$$\left(\begin{array}{c} e \\ \{v := case\ e'_0\ of\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\}\} \cup \theta_1 \\ \{v := case\ e''_0\ of\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\}\} \cup \theta_2 \end{array}\right)$$
$$\Downarrow$$
$$\left(\begin{array}{c} e\{v := case\ v_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to v_i;}\}\} \\ \{v_0 := e'_0,\ \overline{v_i := e'_i\{\overline{v'_{ik} := v_{ik}}\}}\} \cup \theta_1 \\ \{v_0 := e''_0,\ \overline{v_i := e''_i\{\overline{v''_{ik} := v_{ik}}\}}\} \cup \theta_2 \end{array}\right)$$

5.3 Common sub-expression rule

$$\left(\begin{array}{c} e \\ \{v_1 := e',\ v_2 := e'\} \cup \theta_1 \\ \{v_1 := e'',\ v_2 := e''\} \cup \theta_2 \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v_1 := v_2\} \\ \{v_2 := e'\} \cup \theta_1 \\ \{v_2 := e''\} \cup \theta_2 \end{array}\right)$$

5.4 Let-expression

Let-expressions are used for representing results of generalizing expressions. The syntax of a let-expression has the form:

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

where variables v_i may appear in e_g, the value of v_i being obtained by evaluating e_i. Since HLL is a lazy language, a let-expression is semantically equivalent to $e_g\{\overline{v_i := e_i}\}$.

Let-expressions are not allowed in input programs, but may be generated by HOSC when constructing a partial process tree.

5.5 Examples

Here are examples of generalization of expressions embedded by coupling (see examples in 4.5):

- $map\ f \sqcap map\ (compose\ f\ g) =$
  $(map\ h, \{h := f\}, \{h := (compose\ f\ g)\})$
- $\lambda x \to Nil \sqcap \lambda x \to Cons\ a\ Nil =$
  $(\lambda x \to y, \{y := Nil\}, \{y := Cons\ a\ Nil\})$
- $case\ x\ of\ \{Z \to x;\ S\ b \to f\ b;\} \sqcap case\ x\ of\ \{Z \to x;\ S\ b \to (f\ g)\ b;\} =$
  $(case\ x\ of\ \{Z \to x;\ S\ b \to h\ b;\}, \{h := f\}, \{h := f\ g\})$

6 Metacomputation

A process tree is a directed (possibly infinite) tree whose nodes are labeled with HLL expressions. A process tree is built by HOSC in the process of *metacomputation* ([5, 6, 1, 2]), which is performed by symbolically evaluating expressions that, unlike the case of ordinary evaluation, may contain free variables.

The expression a node n is labeled with will be denoted by $n.expr$.

Given a source program, the construction of its process tree starts with creating a single-node tree whose root is labeled with the program's target expression. Then the process tree is incrementally built by adding children to its leaves according to the *driving* rules presented below.

6.1 Driving step

The operator $\mathcal{D}$ (Fig. 5) takes as argument an expression in a leaf node and returns zero or more expressions $e_1, \ldots, e_k$. Then the supercompiler adds k child nodes to the leaf labeling them with the expressions $e_1, \ldots, e_k$.

The last rule applies to let-expressions which are not present in source programs but may appear as a result of generalization.

7 Partial process tree

In the general case, metacomputation may result in constructing an infinite process tree. The task of supercompilation is to turn this tree into a finite (possibly cyclic) graph, which is referred to as a *partial process tree*. A partial process tree differs from a process tree in the following points:

- Nodes of a partial process tree may contain let-expressions.

$$
\begin{array}{lcl}
\mathcal{D}[\![v\ \overline{e_i}]\!] & \Rightarrow & [v,\ \overline{e_i}] \\
\mathcal{D}[\![c\ \overline{e_i}]\!] & \Rightarrow & [\overline{e_i}] \\
\mathcal{D}[\![\lambda v_0 \to e_0]\!] & \Rightarrow & [e_0] \\
\mathcal{D}[\![con\langle f_0 \rangle]\!] & \Rightarrow & [con\langle unfold(f_0) \rangle] \\
\mathcal{D}[\![con\langle (\lambda v_0 \to e_0)\ e_1 \rangle]\!] & \Rightarrow & [con\langle e_0\{v_0 := e_1\} \rangle] \\
\mathcal{D}[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} \rangle]\!] & \Rightarrow & \left[con\langle e_j\{\overline{v_{jk} := e'_k}\} \rangle\right] \\
\mathcal{D}[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\} \rangle]\!] & \Rightarrow & \left[e,\ \overline{con\langle e_i\{v\ \overline{e'_j} := p_i\} \rangle}\right] \\
\mathcal{D}[\![let\ \overline{v_i = e_i;}\ in\ e]\!] & \Rightarrow & [e,\ \overline{e_i}]
\end{array}
$$

Figure 5: Driving step

- A partial process tree may have *cycles*. Let a node α be an ancestor of a node β and $\beta.expr$ a renaming of $\alpha.expr$ ($\beta.expr \simeq \alpha.expr$). Then it is possible to create a "return" edge $\beta \rightrightarrows \alpha$ from the *repeat* (or recursive) node β to the *function* (or base) node α.

Thus, 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.

The goal of supercompilation is to construct a partial process tree of a reasonable size.

Fig. 6 shows a number of operations used by HOSC for constructing partial process trees and generating residual programs.

A node β is referred to as *processed*, if one of the following conditions holds:

- β is the source of a "return" edge $\beta \rightrightarrows \alpha$,
- $\beta.expr$ is a nullary constructor,
- $\beta.expr$ is a local variable.

A node β is *non-trivial*, if $\beta.expr$ can be decomposed as either

$$con\langle f \rangle \qquad \text{or} \qquad con\langle case\ v\ \overline{e_j}\ of\ \{\overline{p_i \to e_i;}\} \rangle$$

Otherwise, the node β is referred to as *trivial*.

$processed(node)$	Returns $true$ or $false$ depending on whether the node is processed or not.
$trivial(node)$	Returns $true$ or $false$ depending on whether the node is trivial or not.
$children(t,\ \alpha)$	Returns an ordered list of child nodes for the node α of a tree t.
$addChildren(t,\ \beta,\ es)$	Adds child nodes to the node β of the tree t and puts expressions es into them.
$replace(t,\ \alpha,\ expr)$	Replaces a subtree with root α by a single node β such that $\beta.expr = expr$.
$ancestor(t,\ \beta,\ \simeq)$	Returns a node α - the closest ancestor of β, such that $\alpha.expr \simeq \beta.expr$, or $\bullet$ if β doesn't have such ancestor.
$ancestor(t,\ \beta,\ \lessdot)$	Returns a node α - the closest ancestor of β, such that $\alpha.expr \lessdot \beta.expr$, or $\bullet$ if β doesn't have such ancestor.
$ancestor(t,\ \beta,\ \trianglelefteq_c)$	Returns a node α - the closest ancestor of β, such that $\alpha.expr \trianglelefteq_c \beta.expr$, or $\bullet$ if β doesn't have such ancestor.
$fold(t,\ \alpha,\ \beta)$	Adds a "return" edge $\beta \rightrightarrows \alpha$ to form a cycle.
$abstract(t,\ \alpha,\ \beta)$	$= replace(t,\ \alpha,\ let\ \overline{v_i = e_i;}\ in\ e_g)$, where $(e_g, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $e_g\theta_1 = e_g\{\overline{v_i := e_i}\} = let\ \overline{v_i = e_i;}\ in\ e_g$.
$\lceil \alpha \not\downarrow t \rceil$	Returns all repeat nodes of a node α, $\lceil \alpha \not\downarrow t \rceil = [\overline{\beta_i}] : \beta_i \rightrightarrows \alpha$, or $\bullet$ if α is not a function node.
$\lceil \alpha \uparrow\uparrow t \rceil$	Returns a function node for a node α, $\lceil \alpha \uparrow\uparrow t \rceil = \beta : \alpha \rightrightarrows \beta$, or $\bullet$ if α is not a repeat node.
$drive(t, \alpha)$	$= addChildren(t,\ \alpha,\ \mathcal{D}[\![\alpha.expr]\!])$ - executes driving step.
$unprocessedLeaves(t)$	Returns a list of all unprocessed leaves of a tree t or $\bullet$ if all leaves are processed.

Figure 6: Operations on partial process tree

8 Partial process tree construction

Now we describe the core of the supercompiler HOSC: the algorithm of constructing a partial process tree (see Fig. 7).

Given a source program, the algorithm starts with the creation of a single-node tree whose root is labeled with the program's target expression. Then, while there is at least one unprocessed leaf β:

1. If β is trivial, $\beta.expr$ is "metaevaluated" by performing a driving step.

2. If β has an ancestor α, such that $\alpha.expr \simeq \beta.expr$, then α becomes a function node for β, i.e. a "return" edge $\beta \rightrightarrows \alpha$ is added to the tree.

```
while unprocessedLeaves(t) ≠ • do
    β = unprocessedLeaves(t).head
    if trivial(β) then
        t = drive(t, β)
    else if ancestor(t, β, ≃) ≠ • then
        α = ancestor(t, β, ≃)
        fold(t, α, β)
    else if ancestor(t, β, ◁) ≠ • then
        α = ancestor(t, β, ◁)
        t = abstract(t, β, α)
    else if ancestor(t, β, ⊴_c) ≠ • then
        α = ancestor(t, β, ⊴_c)
        t = abstract(t, α, β)
    else
        t = drive(t, β)
end
```

Figure 7: Algorithm of partial process tree construction

3. If β has an ancestor α, such that $\alpha.expr \lhd \beta.expr$, then $\beta.expr$ is generalized.

4. If β has an ancestor α, such that $\alpha.expr \trianglelefteq_c \beta.expr$, then $\alpha.expr$ is generalized.

5. Otherwise, $\beta.expr$ is "metaevaluated" by performing a driving step.

The fourth case is the central point of the algorithm: removing it would cause the supercompiler not to terminate for a lot of source programs!

However, the test for the homeomorphic embedding works as a "whistle" signalling that two expressions have "too much in common", but a "return" edge still cannot be added to the tree. So the supercompiler generalizes the expression labeling the "function" (base) node, thereby preventing the the infinite growth of the process tree.

9 Residual program construction

Here we describe an algorithm for transforming a partial process tree t into a program in the language HLL.

The algorithm (presented in Fig. 8) is defined via two mutually recursive operators (functions): $\mathcal{C}$ and $\mathcal{C}'$. A residual program can be extracted from a

$$\mathcal{C}\ [\![\alpha]\!]_{t,\Sigma} = \mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma} \qquad (C_0)$$

$$\mathcal{C}'[\![let\ \overline{v_i = e_i};\ in\ e]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]\{\overline{v_i = \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma}}\} \qquad (C_1)$$

$$\mathcal{C}'[\![v\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} \qquad (C_2)$$

$$\mathcal{C}'[\![c\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} \qquad (C_3)$$

$$\mathcal{C}'[\![\lambda v_0 \to e_0]\!]_{t,\alpha,\Sigma} \Rightarrow \lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma} \qquad (C_4)$$

$$\mathcal{C}'[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} \qquad (C_5)$$

$$\mathcal{C}'[\![con\langle case\ c\ \overline{e_j'}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} \qquad (C_6)$$

$$\mathcal{C}'[\![con\langle f\rangle]\!]_{t,\alpha,\Sigma}$$

$$\Rightarrow letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma']\!]_{t,\Sigma'})\theta' \quad \text{if } \lceil \alpha \not\downarrow_{\searrow} t \rceil \neq \bullet \qquad (C_7^1)$$

$$in\ f'\overline{v_i'}$$

where

$$\overline{[\beta_i]} = \lceil \alpha \not\downarrow_{\searrow} t \rceil,\ \theta_i = \alpha.expr \circledcirc \beta_i.expr,$$

$$\overline{v_i'} = domain(\bigcup \overline{\theta_i}),\ \theta' = \{\overline{v_i' := v_i}\},$$

$$\Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i}),\ f' \text{ and } \overline{v_i} \text{ are fresh}$$

$$\Rightarrow f'_{sig}\theta \quad \text{if } \lceil \alpha \uparrow\uparrow t \rceil \neq \bullet \qquad (C_7^2)$$

where

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

$$\Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} \quad \text{otherwise} \qquad (C_7^3)$$

$$\mathcal{C}'[\![con\langle case\ v\ \overline{e_j'}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma}$$

$$\Rightarrow letrec\ f' = \lambda\overline{v_i} \to (case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'}of\{\overline{p_i \to \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma'};}\})\theta' \quad \text{if } \lceil \alpha \not\downarrow_{\searrow} t \rceil \neq \bullet \qquad (C_8^1)$$

$$in\ f'\overline{v_i'}$$

where

$$\overline{[\beta_i]} = \lceil \alpha \not\downarrow_{\searrow} t \rceil,\ \theta_i = \alpha.expr \circledcirc \beta_i.expr,$$

$$\overline{v_i'} = domain(\bigcup \overline{\theta_i}),\ \theta' = \{\overline{v_i' := v_i}\},$$

$$\Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i}),\ f' \text{ and } \overline{v_i} \text{ are fresh}$$

$$\Rightarrow f'_{sig}\theta \quad \text{if } \lceil \alpha \uparrow\uparrow t \rceil \neq \bullet \qquad (C_8^2)$$

where

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

$$\Rightarrow case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'}of\{\overline{p_i \to \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma'};}\} \quad \text{otherwise} \qquad (C_8^3)$$

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 8: Rules for constructing residual program

partial process tree by evaluating:

$$\mathcal{C}\ [\![t.root]\!]_{t,\{\}}$$

In order to construct a residual program we traverse a partial process tree top-down starting from the root node. Traversing a function node produces the definition of a recursive function. The correspondence between a function node and the signature of a new function in recorded in Σ. Later Σ is used for constructing a recursive function call.

Let us consider the most important details of the algorithm.

- The rule C_0 defines $\mathcal{C}$ in terms of $\mathcal{C}'$.
- The rules C_1, C_2 and C_3 define how to combine supercompiled parts of a let-expression, a call to an unknown function or a constructor expression in order to produce the corresponding residual expression.
- The rule C_4 defines how to construct a residual λ-abstraction from its supercompiled body.
- The rules C_5 and C_6 correspond to the cases $con\langle(\lambda v_0 \to e_0)\ e_1\rangle$ and $con\langle case\ c\ \overline{e_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$, in which a reduction step was completely performed by driving. Thus no actions need to be inserted into the residual program.
- The rules C_7 and C_8 are used when traversing non-trivial nodes:
  - A recursive function definition is generated when traversing a function node (the rules C_7^1 and C_8^1). Note that only those free variables which were not preserved in repeat nodes become the arguments of λ-abstractions. This trick allows the arity of residual functions to be reduced. The correspondence between a function node and the signature of a new function is recorded in Σ and passed down the tree.
  - A call to a recursive function defined in a base node is generated when traversing a repeat node (the rules C_7^2 and C_8^2).
  - The rules C_7^3 and C_8^3 are used when traversing a node which is neither a "function" nor "repeat" one.

10 Related work

Originally supercompilation was formulated for the programming language Refal [20]. Currently, the most advanced Refal supercompiler is SCP4 [16].

In 1990-s supercompilation was reformulated for a number of simpler languages. The main goal of that work was to understand which parts of supercompilation were due to Refal and which ones were of a fundamental character.

The first *complete* and *formal* description of the concepts and algorithms essential for any supercompiler (driving, generalization, residual program construction) can be found in Sørensen's Master's Thesis [17] and in the later papers by Sørensen and Glück [6, 19], presenting a supercompiler for a simple first-order functional language (with the call-by-name semantics).

A full description of a supercompiler for the language TSG (a simple first-order funcional call-by-name language) is given by Abramov [2].

The supercompilers by Abramov, Sørensen and Glück construct a partial process tree which is then transformed into a residual program.

In recent years there has been a growing interest in supercompilation for higher-order functional languages [8, 10, 11, 12, 15, 14].

The techniques used by Jonsson [10, 11] and Mitchell [15, 14] are similar to that of deforestation [23, 7], since they do not generate a partial process tree at a separate stage, so that the driving (symbolic evaluation) and the residual program generation are closely interleaved.

On the contrary, Hamilton [8] and Kabir [12] prefer to deal with partial process trees.

While the papers by Hamilton, Jonsson, Kabir and Mitchell pay much attention to driving and residual program generation, unfortunately, the problems related to homeomorphic embedding and generalization are not given an account they deserve.

Homeomorphic embedding relation (used as a whistle for generalization) is described in detail in [6, 17, 18] for a first-order functional language without case-expressions. Case-expressions are discussed in [19], however a formal description of homeomorphic embedding of case-expressions is missing. The same situation is with the higher-order languages with case-expressions dealt with in [8, 10, 11, 12, 15, 14].

An algorithm for finding a most specific generalization for a first-order language without case-expressions (that is, without bound variables) is described in already mentioned papers [6, 17, 18]. Bound variables are considered in [19, 10, 8, 15], and the algorithm for finding an MSG is said to be a modification of algorithms from [6, 17, 18], but it is not described in detail.

11 Conclusions

We have described the internal structure of an experimental supercompiler HOSC dealing with programs written in a higher-order functional language.

A *complete* and *formal* account has been given of all essential concepts and algorithms the supercompiler HOSC is based upon. Of particular interest are the questions related to homeomorphic embedding and generalization for expressions containing bound variables.

12 Acknowledgements

The author expresses his gratitude to Sergei Romanenko and all participants of Refal seminar at Keldysh Institute for useful comments and fruitful discussions of this work.

References

[1] S.M. Abramov. *Metacomputation and its applications.* "Science" Publish., Moscow, 1995.

[2] S.M. Abramov and L.V. Parmyonova. *Metacomputation and its applications. Supercompilation.* Ailamazyan University of Pereslavl, 2006.

[3] S. Abramsky. The lazy lambda calculus. In *Research topics in functional programming*, pages 65–116. Addison-Wesley Longman, 1990.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM New York, NY, USA, 1982.

[5] R. Glück and A.V. Klimov. Occam's razor in metacompuation: the notion of a perfect process tree. In *WSA '93: Proceedings of the Third International Workshop on Static Analysis*, volume 724 of *LNCS*, pages 112–123. Springer, 1993.

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

[7] G.W. Hamilton. Higher order deforestation. *Fundamenta Informaticae*, 69(1-2):39–61, 2006.

[8] G.W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70. ACM Press New York, NY, USA, 2007.

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

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

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

[12] M.H. Kabir. *Automatic Inductive Theorem Proving and Program Construction Methods Using Program Transformation.* PhD thesis, Dublin City University, Faculty of Engineering and Computing, School of Computing, 2007.

[13] M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The essence of computation*, volume 2566 of *LNCS*, pages 379–403. Springer, 2002.

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

[15] N. Mitchell and C. Runciman. A supercompiler for core haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *LNCS*, pages 147–164. Springer, 2008.

[16] A.P. Nemytykh. The supercompiler scp4: General structure. In *PSI 2003*, volume 2890 of *LNCS*, pages 162–170. Springer, 2003.

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

[18] M.H. Sørensen. Convergence of program transformers in the metric space of trees. In *Proceedings of the Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 315–337. Springer, 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] V.F. Turchin. A supercompiler system based on the language REFAL. *SIGPLAN Not.*, 14(2):46–54, 1979.

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

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

[23] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88*, volume 300 of *LNCS*, pages 344–358. Springer, 1988.

A Examples

This Appendix contains two examples of supercompilation performed by HOSC supercompiler. Both examples deal with higher-order functions.

A.1 Iterate

```
data List a = Nil | Cons a (List a);
data Nat = Z | S Nat;

iterate (λn → S n) Z where

iterate = λf x → Cons x (iterate f (f x));
```

Figure 9: Iterate: input

The language HLL allows infinite data structures to be processed. Given a function `f` and an initial value `x`, the function `iterate` produces an infinite list (stream) of repeated applications of `f` to `x`:

```
iterate f x = Cons x (Cons (f x) (Cons (f (f x)) (Cons ...)))
```

The target expression in the program in Fig. 9 defines the infinite list (stream) of natural numbers by means of the function `iterate`.

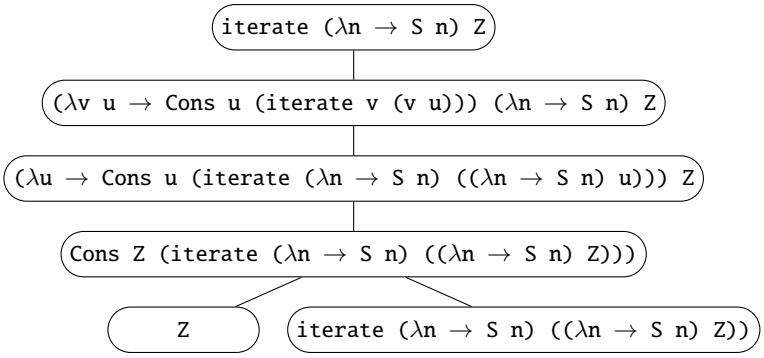

Figure 10: Iterate: metacomputation before generalization

Driving this target expression, HOSC produces the process tree shown in Fig. 10. At this step, it finds out that the expression in the root node is embedded into the expression in the rightmost leaf:

`iterate (λn → S n) Z` $\trianglelefteq_c$ `iterate (λn → S n) ((λn → S n) Z)`

However, the second expression is not an instance of the first one! For this reason, HOSC generalizes the expression in the root node, replacing

`iterate (λn → S n) Z`

with the let-expression

`let z = Z in iterate (λn → S n) z`

and deleting the whole subtree. Further driving (after this generalization and till the next whistle) is shown in Fig. 11:

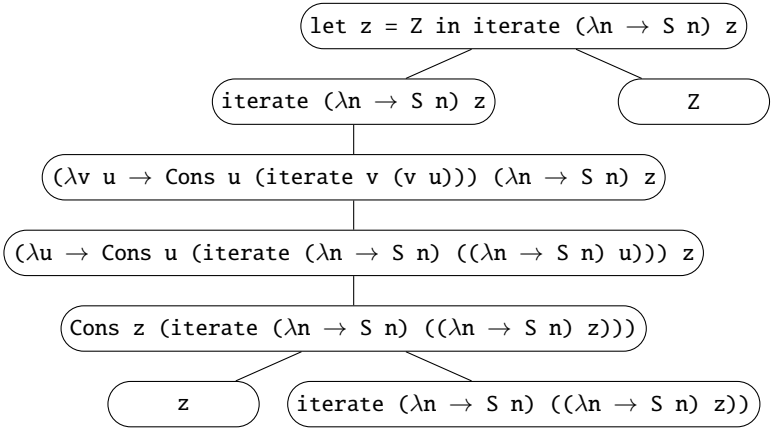

Figure 11: Iterate: metacomputation after the first generalization

At this step an embedding is detected:

`iterate (λn → S n) z` $\trianglelefteq_c$ `iterate (λn → S n) ((λn → S n) z)`

This time the second expression is an instance of the first one:

`iterate (λn → S n) z` $\lhd$ `iterate (λn → S n) ((λn → S n) z)`

Thus the second expression is generalized by replacing the expression

`iterate (λn → S n) ((λn → S n) z)`

with the let-expression

`let y = (λn → S n) z in iterate (λn → S n) ((λn → S n) z)`

Further metacomputation results in a complete partial process tree shown in Fig. 12. The residual program constructed from this tree is presented in Fig. 13. It is worth to note that the function

`(λn → S n)`

was inlined into the body of the recursive function `f`.

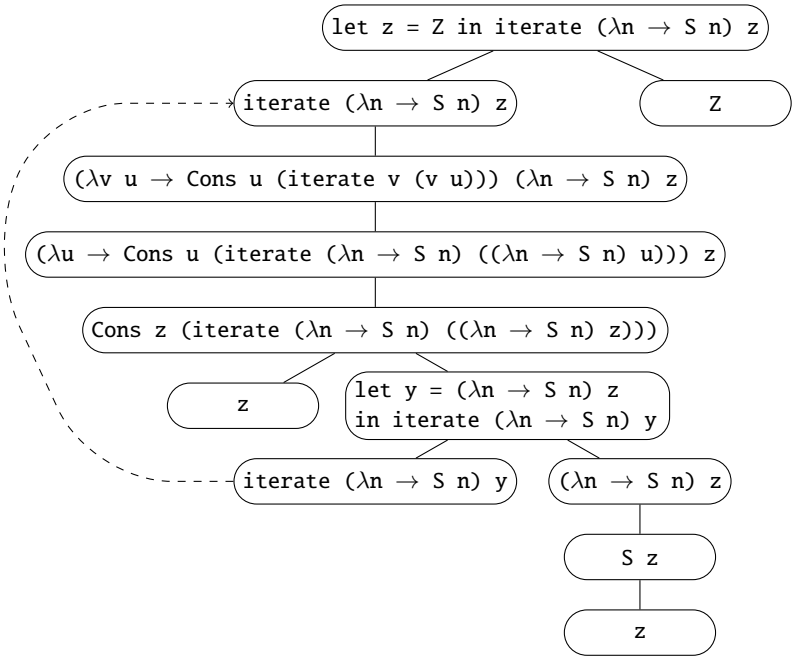

Figure 12: Iterate: metacomputation after the second generalization

```
data List a = Nil | Cons a (List a);
data Nat = Z | S Nat;

letrec
    f = λw → Cons w (f (S w))
in f Z
```

Figure 13: Iterate: output

A.2 Church numerals

Church numerals are the representations of natural numbers under Church encoding. The higher-order function that represents a natural number n is the function that maps any other function f to its n-fold composition. In simpler terms, the "value" of the numeral is equivalent to the number of times the function encapsulates its argument.

$$f^n = f \circ f \circ f \circ \ldots \circ f$$

Church numerals 0, 1, 2, ..., n are defined as follows in the lambda calculus:

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

eq (add x y) (unchurch(churchAdd (church x) (church y))) where

eq = λm y → case m of {
  Z → case n of {Z → True; S n1 → False; } ;
  S m1 → case n of {Z → False; S n1 → eq m1 n1;} ;
};
church = λn → case n of {
  Z → λf t → t;
  S n1 → λf t → f (church n1 f t);
};
unchurch = λn → n (λt → S t) Z;
churchAdd = λm n → (λf t → m f (n f t));
add = λm n → case m of {
  Z → n;
  S m1 → S (add m1 n);
};
```

Figure 14: Church addition: input

$$0 = \lambda f\ x \to x$$
$$1 = \lambda f\ x \to f\ x$$
$$2 = \lambda f\ x \to f\ (f\ x)$$
$$\ldots$$
$$n = \lambda f\ x \to f^n\ x$$

The addition function $churchAdd\ m\ n = m + n$ is based on the identity $f^{(m+n)}\ x = f^m\ (f^n\ x)$.

The program shown in Fig. 14 contains definitions of `Boolean` and `Nat` datatypes. `Nat` represents numbers in Peano encoding by means of constructors `Z` (zero) and `S` (successor). The function `eq` tests Peano numbers for equality. The `church` and `unchurch` functions convert between Peano numbers and Church numerals. The `churchAdd` and `add` functions perform addition of Church numerals and Peano numbers correspondingly.

The target expression of the program tests whether the sum of two Peano numbers `x` and `y` is equal to the decoded sum of corresponding Church numerals `church x` and `church x`. Note that the target expression may return `False`, since this constructor appears in the definition of the function `eq`.

The result of supercompiling this program is shown in Fig. 15. Now it is evident that the residual program cannot return `False` since this constructor never appears in the body of the program.

The structure of the original program has been simplified: it is almost

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

case x of {
  Z → case y of {
     Z → True;
     S y1 → letrec f= λz →
               case z of { Z → True; S z1 → f z1;}
            in f y1;
     };
  S x1 →
    letrec g = λv →
       case v of {
         Z → case y of {
             Z → True;
             S y2 → letrec h= λw →
                       case w of { Z → True; S w1 → h w1; }
                    in h y2;
         };
         S v1 → g v1;
       }
    in g x1;
}
```

Figure 15: Church addition: output

evident that the residual program just destructs `x` and `y` and finally returns `True`.

It can also be shown that this program returns `True`, if `x` and `y` are "good" values (finite Peano numbers), since the result of program execution depends on successful destruction of `x` and `y` only.