KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

Ilya G. Klyuchnikov

Supercompiler HOSC 1.1: proof of termination

Moscow
2010

Ilya G. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination

The paper contributes the proof of termination 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 1.1: доказательство завершаемости

В работе приводится доказательство завершаемости экпериментального суперкомпилятора HOSC, работающего с функциями высших порядков.

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

Contents

1 Introduction

The purpose of this paper is to show that the supercompiler HOSC [6, 7] terminates for any input program.

The paper [6] describes the internal structure of HOSC 1.0, an experimental supercompiler for a higher-order call-by-name language[1].

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

An attempt to proof that HOSC 1.0 terminates for any input program failed, and resulted in constructing a counter-example (see Section 6). However, the source of non-termination having being identified, the supercompiler was modified to produce the version HOSC 1.1, which, unlike HOSC 1.0 (see Appendix B), is guaranteed to terminate. In the following, unless otherwise stated, "HOSC" will mean "HOSC 1.1".

We use the framework for proving termination of *abstract program transformers* developed by Sørensen [14], since HOSC can be considered as an abstract program transformer of partial process trees.

Informally speaking, given a source program, HOSC constructs its partial process tree in the following way [6]. First, it creates a single-node tree, whose root is labeled with the program's target expression. Then HOSC proceeds by adding children to the leaves of the tree, until all nodes in the tree become *processed*.

Let β be an unprocessed node.

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 special "return" edge $\beta \rightrightarrows \alpha$ is added to the tree, thereby making the node β a processed one.
3. If β has an ancestor α, such that $\alpha.expr \lessdot \beta.expr$, 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.

Performing a driving step is followed by adding to β child nodes labeled by the expressions produced by metaevaluating $\beta.expr$.

Theorem 1 (HOSC termination)**.** *The supercompiler HOSC 1.1 terminates for any source program.*

Following is the outline of the proof.

- Step 1 cannot be repeated indefinitely, since the expressions in the child nodes produced by driving a trivial node are strictly smaller in size than the expression in the parent node.
- Step 2 cannot lead to non-termination, since it only adds a "return" edge to the process tree, and, for a finite tree, this cannot be repeated indefinitely.

- Therefore, any sequence of steps 1 and 2 is finite. Suppose that HOSC performs either step 3 or 4, generalizing an expression labeling a node. We will show that an expression cannot be generalized infinitely many times, since each generalization reduces the "size" of the expression.

- Finally, we have to show that step 5 cannot be repeated indefinitely. This follows from the fact that $\trianglelefteq|_\rho$ (based on $\trianglelefteq_c$) is a well-quasi-order relation, for which reason any infinite sequence of steps 5 includes a step 4.

The rest of the paper is organized as follows.

Section 2 presents the part of Sørensen's framework that is essential for our proof of termination.

Section 3 explains why additional requirements were imposed in the definition of the homeomorphic embedding relation $\trianglelefteq|_\rho$ (used as a whistle), and why they are essential if the language is a higher-order one.

Section 4 shows that $\trianglelefteq|_\rho$ is a well-quasi-order on any set of expressions *labeling nodes of a process tree*. (Although, it is not true for an arbitrary set of expressions.)

Finally, section 5 shows that HOSC terminates for any source program, since it meets Sørensen's conditions *sufficient* for an abstract program transformer to terminate.

2 Abstract program transformers

In general, we use the same notation as in [6]. However, we need to distinguish HLL expressions (which may appear in source HLL programs as well as in process trees) from let-expressions (which may only appear in process trees).

- $\mathcal{E}$ denotes the set of all HLL expressions,

- $\mathcal{L}$ denotes the set $\mathcal{E}$ extended with the set of all let-expressions.

Note that an HLL expression $e \in \mathcal{E}$ is considered to be equivalent to *let in* e).

Definition 2 (Quasi-order)**.** *Let S be a set with a relation $\leq$. Then $(S, \leq)$ is a quasi-order if $\leq$ is reflexive and transitive. We write $s < s'$ if $s \leq s'$ and $s' \not\leq s$.*

Definition 3 (Well-founded quasi-order)**.** *Let $(S, \leq)$ be a quasi-order. Then $(S, \leq)$ is a well-founded quasi-order if there is no infinite sequence $s_0, s_1, \ldots \in S$ with $s_0 > s_1 > \ldots$.*

Definition 4 (Well-quasi-order)**.** *Let $(S, \leq)$ be a quasi-order. Then $(S, \leq)$ is a well-quasi-order if, for every infinite sequence $s_0, s_1, \ldots \in S$, there are $i < j$ with $s_i \leq s_j$.*

Definition 5 (Trees over sets of expressions)**.** *Let $\mathcal{L}$ be a set of expressions. A (partial) process tree t (see [6]) is a tree over $\mathcal{L}$ if $\forall \gamma \in t : \gamma.expr \in \mathcal{L}$.*

The set of all trees over $\mathcal{L}$ will be denoted as $T(\mathcal{L})$

Definition 6 (Abstract program transformers)**.** *An abstract program transformer on $\mathcal{L}$ is a map M: $T(\mathcal{L}) \to T(\mathcal{L})$.*

The supercompiler HOSC is a program transformer on $\mathcal{L}$, where $\mathcal{L}$ is the set of all HLL expressions extended with let-expressions.

Definition 7 (Termination of program transformers)**.** *(1) An abstract program transformer M on $\mathcal{L}$ terminates on $t \in T(\mathcal{L})$ if $M^i(t) = M^{i+1}(t)$ for some i. (2) An abstract program transformer M on $\mathcal{L}$ terminates if M terminates on all singletons $t \in T(\mathcal{L})$.*

In the following, for the sake of brevity, the initial tree will be denoted by t_0, and the tree produced after the i-th transformation step by t_i.

Proposition 8 (Cauchy transformers)**.** *Let $(\mathcal{L}, \leq)$ be a well-founded quasi-order. An abstract program transformer M on $\mathcal{L}$ is a Cauchy transformer if*

$$t_{i+1} = t_i\{\gamma := t'\}$$

for a node γ, and one of the following conditions is satisfied:

- $\gamma \in leaves(t_i)$ *and* $\gamma.expr = t'.root.expr$
- $\gamma.expr > t'.root.expr$

Proposition 9 (Finitary continuous predicates)**.** *Let $\{\mathcal{L}_1, \mathcal{L}_2\}$ be a partition of $\mathcal{L}$, $(\mathcal{L}_1, \leq_1)$ a well-quasi-order, and $(\mathcal{L}_2, \leq_2)$ a well-founded quasi-order. Let $p : T(\mathcal{L}) \to \mathbb{B}$ be defined as*

$$p(t) = \begin{cases} 0 & \text{if } \exists \alpha, \beta : \alpha \text{ is an ancestor of } \beta, \\ & \alpha.expr, \beta.expr \in \mathcal{L}_1 \text{ and } \alpha.expr \leq_1 \beta.expr \\ 0 & \text{if } \exists \alpha, \beta : \alpha \to \beta, \alpha.expr,\ \beta.expr \in \mathcal{L}_2 \text{ and } \alpha.expr \not> _2 \beta.expr \\ 1 & \text{otherwise} \end{cases}$$

Then p is a finitary continuous predicate.

Definition 10 (Interior of a tree)**.** *A set of nodes is the* interior *of a tree t if it consists of the root and all non-leaf nodes of t.*

The interior of a tree t will be denoted as t^0.

Proposition 11. *Let $p : T(\mathcal{L}) \to \mathbb{B}$ be a finitary continuous predicate. Then q, defined as $q(t) = p(t^0)$ is also a finitary continuous predicate.*

$$e ::= v \mid c(e_1, \ldots, e_n) \mid f(e_1, \ldots, e_n)$$

Figure 1: First order language: the syntax of expressions

Theorem 12 (Sørensen)**.** *Let abstract program transformer* $M : T(\mathcal{L}) \to T(\mathcal{L})$ *maintain predicate* $p : T(\mathcal{L}) \to \mathbb{B}$*. If*

1. *M is Cauchy, and*
2. *p is finitary and continuous,*

then M terminates.

3 Homeomorphic embedding $\trianglelefteq|_\rho$

The most sophisticated part of a supercompiler is the algorithm of generalization, whose purpose is to ensure termination of supercompilation for any input program by preventing the construction of an infinite process tree. The most difficult problem is to decide which expression is worth to be generalized. For historical reasons, this part of the generalization algorithm is called a *whistle* [15, 16].

An approach to ensuring termination that has gained popularity in supercompilation and similar program transformation techniques is the use of the homeomorphic embedding relation [8, 9]. A whistle based on the homeomorphic embedding relation compares the expression labeling the current node with expressions in ancestor nodes. If the whistle finds out that the two expressions are *syntactically* similar, the supercompiler generalizes one of the expressions to avoid constructing an infinite path in the partial process tree. Hence, the *essential* property of the whistle is that it eventually blows for a sequence of expressions *produced by driving*. In technical terms, it means that the whistle is based on a *well-quasi-order*.

3.1 The basis: a first-order language

Let us consider a first-order expression language [13, 14], whose syntax is shown in Fig. 1, and the arity of functors (function names and constructors) is fixed and finite. The well-known definition of the homeomorphic embedding relation for this language is given in Fig. 2. The subtle point is that all variables are assumed to be free and are not distinguished from each other.

Let E_0 be the set of expressions defined by the grammar in Fig. 1.

Variables	Diving	Coupling
$v_1 \trianglelefteq v_2$	$\dfrac{\exists i : e \trianglelefteq e'_i}{e \trianglelefteq \phi(e'_1, \ldots, e'_k)}$	$\dfrac{\forall i : e_i \trianglelefteq e'_i}{\phi(e_1, \ldots, e_k) \trianglelefteq \phi(e'_1, \ldots, e'_k)}$

Figure 2: First-order language: embedding

Theorem 13 (Kruskal)**.** $(E_0, \trianglelefteq)$ *is a well-quasi-order provided that the set of functors (function names and constructors) is finite.*

Proof. Replacing all variables with a fixed constant (a nullary constructor) that is different from all other functors, we get expressions without variables and can reuse the proof given in [3]. □

The relation defined in Fig. 2 possesses the following property, essential for supercompilation: if an expression is embedded into another one by coupling, there can be found a *non-trivial generalization* of the expressions.

3.2 Bound variables

Unlike the first-order supercompiler considered by Sørensen, the supercompiler HOSC deals with HLL, a higher-order language, whose expressions may contain *bound variables* appearing in λ-abstractions and case-expressions. Conceptually, bound variables introduced by case-expressions are not different from bound variables introduced by λ-abstractions. For this reason, in the following, the problems related to bound variables will only be explained using λ-abstractions.

During the check for homeomorphic embedding, a λ-abstraction may be treated as a "special" constructor. However, this simplistic approach would cause problems during supercompilation, since embedding by coupling might not imply the existence of a non-trivial generalization. For example, if bound and free variables were not distinguished, the following expressions would be "naïvely" considered as embedded by coupling, despite there being no non-trivial generalization:

```
λx y → Pair x y
λx y → Pair y x
```

However, if bound variables are distinguished from each other, the above expressions are not considered to be embedded. Indeed, despite the *syntactic* similarity of the expressions, the nameless functions they denote are entirely different *semantically*. For this reason, the definition of the extended homeomorphic embedding in [6] involves a table ρ , which is used to record the

correspondence between bound variables. Let $\trianglelefteq^1$ denote the embedding relation "enhanced" by taking into account the correspondence between bound variables.

Unfortunately, $\trianglelefteq^1$ is not subtle enough, because for two expressions related by coupling there still may not exist a non-trivial generalization. For example, for the expressions

```
λx → x
λx → (S x)
```

For this reason, Section 4.3 in [6] introduces an additional requirement: an expression whose free variables have been already registered in the table ρ is not allowed to dive into another expression. Let $\trianglelefteq^2$ denote the embedding relation $\trianglelefteq^1$ refined with the requirement of Section 4.3.

3.3 Higher-orderness and the arity of application

Kruskal's theorem is applicable to the first-order language defined in Fig 1) due to the fact that the arity of functors (function names and constructors) is fixed, the set of functors appearing in an input program is finite, and driving never produces functors not appearing in the input program.

However, in the case of a higher-order language, a function can be curried, so that, potentially, driving is able to produce function calls of arbitrary "arity": `f`, `f x`, `f x y`, ... In addition, variable can take functional values. Thus it seems natural to treat function names as variables, rather than functors, in which case function applications may be represented by special constructors. But this can still be done in two different ways.

1. We may introduce a single binary constructor `App`, its first argument being the head of the application (which can also be an application), and its second argument being the argument of the application. Let us denote this encoding of application as $\mathcal{A}_2$.

2. We may not allow the head of an application to be an application again by introducing a set of *different* constructors `App2`, `App3` ... having different arities. Let us denote this encoding as $\mathcal{A}_*$.

Thus the expression `f x y` can be represented in one of the following ways:

1. `f x y = App(App(f, x), y)`

2. `f x y = App3(f, x, y)`

In both cases, if e_1 and e_2 are related via $\trianglelefteq^2$ by coupling, there exists a non-trivial generalization. It is also clear that the encoding $\mathcal{A}_2$ causes the whistle to blow more often than the encoding $\mathcal{A}_*$.

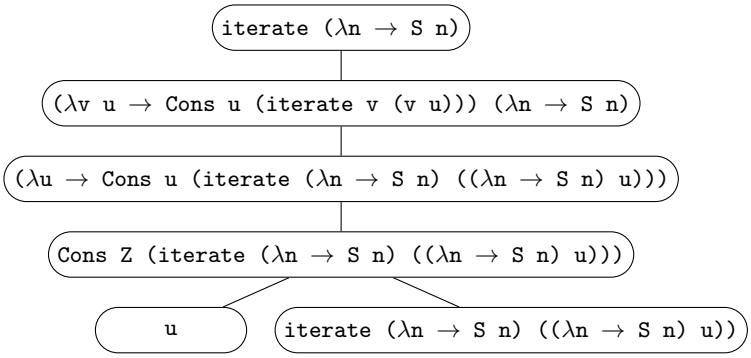

Figure 3: Driving the expression `iterate (λn →Sn)`

In principle, both encodings might be used in a supercompiler. However, the encoding $\mathcal{A}_2$ is not suitable for the supercompiler HOSC, because HOSC, upon encountering a λ-abstraction, tries to transform the body of the λ-abstraction.

For example, let us consider the target expression `iterate (λn →S n)`. A few driving steps produce the tree shown in Fig. 3.

Using the encoding $\mathcal{A}_2$ results in the root expression being embedded by coupling into the right lower one:

`App(iterate, λn→ S n)` $\leq_c$ `App(App(iterate, λn→ S n), App(λn→ S n, u))`

So the root expression gets generalized to

```
let f = iterate in f (λn→ S n)
```

and the function `iterate` does not get specialized with respect to its first argument.

However, the use of the encoding $\mathcal{A}_*$ does cause the function `iterate` to be specialized, and, in general, provides more opportunities for specialization. For this reason, the definition of the homeomorphic embedding relation given in [6] is (implicitly) based on the encoding $\mathcal{A}_*$. Let the relation defined in [6] be denoted as $\trianglelefteq^3$.

4 Well-quasi-order $\trianglelefteq|_\rho$

The purpose of this section is to show that $\trianglelefteq|_\rho$ (which is the same as $\trianglelefteq^3$) is a well-quasi-order relation on any sequence of expressions produced by driving in the process of supercompilation. Recall that $\trianglelefteq^3$ is $\trianglelefteq$ refined with additional

requirements. In the following subsections we will show that, despite additional requirements, $\trianglelefteq$ is still a well-quasi-order on a sequence of expressions *produced by metacomputation.*

Definition 14 (Expressions reachable by metacomputation)**.** *Given an input program prog, an expression e is said to be* reachable by metacomputation *if it appears in the process tree produced by driving prog. The set of expressions reachable by metacomputation of prog will be denoted by* $\mathcal{M}(prog)$*.*

In the following sections we show that $\trianglelefteq^1$ is a well-quasi-order on $\mathcal{M}(prog)$. This is done by changing the representation of bound variables so that Kruskal's theorem will be applicable.

First of all, we have to get rid of bound variables introduced by case-expressions and λ-abstractions.

4.1 Replacing case-expressions with constructors

Let us get rid of bound variables in case-expressions. It can be done by representing a case-expression as a composition of a special *case-constructor* and a λ-abstraction in each of its branches. The name of the case-constructor is determined by the type of the case-expression's argument. The first argument of the case-constructor holds the case-expression's argument, the next arguments hold the case-expression's branches (represented by λ-abstractions). The branches are sorted according to the order in which constructors are enumerated in the declaration of the data type.

For example, the expression

`case x of {Z` $\rightarrow$ `Z; S y` $\rightarrow$ `S y;}`

is represented as

`CaseNat(x, Z,` λ`y` $\rightarrow$ `S y)`

Now let *prog* be an input program. Since *prog* is finite, the number of data type declarations in *prog* is finite. Therefore, the number of possible case-constructors is also finite. Driving is unable to produce new case-constructors, since case-expressions in $\mathcal{M}(prog)$ are either eliminated by reduction or introduced by unfolding of function definitions. Thus the set of case-constructors appearing in $\mathcal{M}(prog)$ is finite.

4.2 Replacing variable names with de Bruijn indices

Unfortunately, driving is able to produce a (potentially) infinite number of bound variables in λ-abstractions. Fortunately, we can eliminite the names of bound variables by replacing them with *de Bruijn indices* [2], which are

natural numbers, such that an index k denotes the number of binders that are in scope between that occurrence and its corresponding binder.

When performing the check for homeomorphic embedding, de Bruijn indices can be treated as just special nullary constructors. It will be shown that, unlike variable names, the set of de Bruijn indices in $\mathcal{M}(prog)$ is finite.

Upon replacing variable names with de Bruijn indices, the above expression takes the form:

```
CaseNat(x, Z, λS 1)
```

The result of encoding an expression e using case-constructors and de Bruijn indices will be denoted by $\mathcal{E}_1[\![e]\!]$, the result of encoding a program $prog$ by $\mathcal{E}_1[\![prog]\!]$.

Lemma 15 ($\mathcal{E}_1$, $\trianglelefteq$, $\trianglelefteq^1$)**.** $e_1 \trianglelefteq^1 e_2 \Leftrightarrow \mathcal{E}_1[\![e_1]\!] \trianglelefteq \mathcal{E}_1[\![e_2]\!]$

Proof. By structural induction over pairs of original expressions and encoded expressions. □

Let $\mathcal{M}_{\mathcal{E}_1}(prog)$ denote $\{\mathcal{E}_1[\![e]\!] \mid e \in \mathcal{M}(prog)\}$.

Lemma 16. *De Bruijn indices appearing in $\mathcal{M}_{\mathcal{E}_1}(prog)$ are bounded.*

Proof. De Bruijn indices appearing in $\mathcal{E}_1[\![prog]\!]$ (and, hence, in the encoded target expression) are bounded, just because $\mathcal{E}_1[\![prog]\!]$ is finite. An index in $\mathcal{M}_{\mathcal{E}_1}(prog)$ cannot exceed the least upper bound for the indices in $\mathcal{E}_1[\![prog]\!]$, because, if an expression e' is produced by a driving step from an expression e, indices in $\mathcal{E}_1[\![e']\!]$ cannot be greater than indexes in $\mathcal{E}_1[\![e]\!]$ and $\mathcal{E}_1[\![prog]\!]$. Indeed, consider the driving rules (listed in Appendix C).

- D_1, D_2, D_8. A driving step does not change de Bruijn indices.
- D_3. The de Bruijn index corresponding to the reduced λ-abstraction disappears (becomes a free variable), all other indices do not change.
- D_4. The indices inside the context do not change. The indices in the unfolded function definition are bounded, since the input program is finite.
- D_5. Since driving in HOSC performs *normal order* β-reduction, the de Bruijn index corresponding to v_0 disappears after the reduction step. All other indices remain unchanged.
- D_6, D_7. This case is analogous to D_5.

□

Lemma 17. $(\mathcal{M}(prog), \trianglelefteq^1)$ *is a well-quasi-order.*

Proof. A number of constructors in $\mathcal{M}_{\mathcal{E}_1}(prog)$ is finite due to Lemma 16. Thus $(\mathcal{M}_{\mathcal{E}_1}(prog), \trianglelefteq)$ is a well-quasi-order. So, due to Lemma 15, $(\mathcal{M}(prog), \trianglelefteq^1)$ is also a well-quasi-order. □

Note that de Bruijn indices remain bound, when applying rules D_5, D_6 and D_7, due to HOSC performing β-reduction in *normal order*! Reducing an expression in non-normal-order may cause de Bruijn indices to grow:

`(λx →` <u>`(λy z → y) (λv → x)`</u>`) w ⇒ (λx z v → x) w`

Or, in terms of de Bruijn indices:

`( λ→` <u>`( λλ→ 2) ( λ→ 2)`</u>`) w ⇒ ( λλλ→ 3) w`

4.3 Extended de Bruijn indices

Let us refine the encoding $\mathcal{E}_1$ in order to take into account the additional restriction imposed by $\trianglelefteq^2$: an expression whose free variables have been already registered in the table ρ is not allowed to dive into another expression (Section 4.3 in [6]).

This requirement can be accounted for by adding to each de Bruijn index a subindex k, such that k equals the number of binders and enclosing constructors that are in scope between that occurrence and its corresponding binder.

While a de Bruijn index only depends on the structure of enclosing λ-abstractions, its subindex takes into account the structure of enclosing λ-abstractions and constructors.

Let $\mathcal{E}_2$ denote the encoding $\mathcal{E}_1$ refined with subindices. Here are 2 examples:

e	$\mathcal{E}_2[\![e]\!]$
`λx → x`	$\lambda\!\rightarrow 1_1$
`λx → S x`	$\lambda\!\rightarrow$ `S` 1_2

The equality of the two extended indices is defined as the equality of the corresponding indices and subindices. When checking two expressions for homeomorphic embedding, we treat different extended indices as different constructors.

Lemma 18 $(\mathcal{E}_2, \trianglelefteq, \trianglelefteq^2)$**.** $e_1 \trianglelefteq^2 e_2 \Leftrightarrow \mathcal{E}_2[\![e_1]\!] \trianglelefteq \mathcal{E}_2[\![e_2]\!]$

Proof. By structural induction over pairs of original expressions and encoded expressions (as in the proof of Lemma 15). □

Let $\mathcal{M}_{\mathcal{E}_2}(prog)$ denote $\{\mathcal{E}_2[\![e]\!] \mid e \in \mathcal{M}(prog)\}$.

Lemma 19. *De Bruijn subindices appearing in* $\mathcal{M}_{\mathcal{E}_2}(prog)$ *are bounded.*

Proof. Virtually the same as for Lemma 16. □

Lemma 20. $(\mathcal{M}(prog), \trianglelefteq^2)$ *is a well-quasi-order.*

Proof. The number of constructors in $\mathcal{M}_{\mathcal{E}_2}(prog)$ is finite due to Lemma 19. Thus $(\mathcal{M}_{\mathcal{E}_2}(prog), \trianglelefteq)$ is a well-quasi-order. So, due to Lemma 18, $(\mathcal{M}(prog), \trianglelefteq^2$) is also a well-quasi-order. □

4.4 The problem of arity

In the encoding $\mathcal{E}_2$, applications are represented by binary constructors (encoding $\mathcal{A}_2$ in Section 3.3). Let us introduce a new encoding $\mathcal{E}_3$, in which, unlike $\mathcal{E}_2$, applications are represented by a set of constructors with different arities (encoding $\mathcal{A}_*$ in Section 3.3).

The Hindley-Milner typing discipline infers *principal* types for expressions in a program. A principal type may contain type variables, which, in turn, may be instantiated with different *concrete* types in different typing contexts. Let us consider a simple function *id*:

```
id :: a → a;
id = λx → x;
```

The type of *id* is $id :: \forall a.a \to a$. Since the type variable a is universally quantified and is instantiated depending on the context, we cannot reason about the arity of the symbol *id* in a simple way. Indeed, the symbol *id* may be of an arbitrary arity in a certain context. For instance, *id* may be applied to an arbitrary number of arguments, each one also being *id*!

```
id
id id
id id id
...
id id id id ...
```

This is possible owing to the function *id* being a polymorphic one.

Thus, if the type of an expression e_0 (or, in particular, a variable) is polymorphic, then there can be constructed a *well-typed* expression $e = e_0\ e_1 \ldots e_n$ of any arity n.

However, if a type t_0 of an expression e_0 does not contain type variables (i.e. is *monomorphic*), then it can be easily shown that in any *well-typed* expression $e = e_0\ e_1 \ldots e_n$ the arity n is bounded.

If we consider idA, a concretization of *id* for a concrete type A, then the arity of idA in any context cannot exceed 1, since an expression $idA\ e$ has the non-functional type $idA\ e :: A$.

The following examples shed some light on why the Hindley-Milner typing discipline prevents the generation of expressions of unbounded arity:

$$
\begin{array}{rcll}
t & ::= & \tau & \text{(type variable)} \\
 & | & t \to t & \text{(arrow)} \\
 & | & TyCon\ \overline{t_i} & \text{(type constructor)}
\end{array}
$$

Figure 4: Syntax of HLL types

```
f = λx → f f x;
h = λx → h x h;
f1 = λx y → f1 (x y) y;
f2 = λx y → f2 x (y x);
```

Although the above function definitions might produce expressions with unbounded arity, they are just not well-typed.

Definition 21 (The arity of an application)**.** *The arity $a[\![e]\!]$ of an application e is defined inductively as follows:*

$$
\begin{array}{lcl}
a[\![v]\!] & = & 0 \\
a[\![f]\!] & = & 0 \\
a[\![\lambda v_0 \to e_0]\!] & = & 0 \\
a[\![c\ \overline{e_i}]\!] & = & 0 \\
a[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] & = & 0 \\
a[\![e_0\ e_1\ \ldots\ e_n]\!] & = & n
\end{array}
$$

Lemma 22 ($\mathcal{E}_3$, $\trianglelefteq$, $\trianglelefteq^3$)**.** $e_1 \trianglelefteq^3 e_2 \Leftrightarrow \mathcal{E}_3[\![e_1]\!] \trianglelefteq \mathcal{E}_2[\![e_3]\!]$

Proof. By structural induction over pairs of original and encoded expressions (analogously to Lemmas 15 and 18). □

The language HLL is typed according to Hindley-Milner type discipline [1]. The syntax of types is shown in Fig. 4.

Definition 23 (Monomorphic type)**.** *A type is monomorphic if it does not contain type variables.*

Definition 24 (The arity of a type)**.** *The arity $A(t)$ of a type t is defined inductively as follows:*

$$
\begin{array}{lcl}
A[\![\alpha]\!] & = & 0 \\
A[\![t_1 \to t_2]\!] & = & 1 + A(t_2) \\
A[\![TyCon\ \overline{t_i}]\!] & = & 0
\end{array}
$$

Let $\mathcal{M}_{\mathcal{E}_3}(prog)$ denote $\{\mathcal{E}_3[\![e]\!] \mid e \in \mathcal{M}(prog)\}$.

We need to show that Kruskal's theorem is valid for the set $\mathcal{M}_{\mathcal{E}_3}(prog)$. If we succeed in proving that the arity of expression types in $\mathcal{M}_{\mathcal{E}_3}(prog)$ is bounded, this will imply that the arity of expressions in $\mathcal{M}_{\mathcal{E}_3}(prog)$ is also bounded. Hence, it will be possible to encode all applications in $\mathcal{M}_{\mathcal{E}_3}(prog)$ by a finite number of constructors. Hence, Kruskal's theorem will be shown to be applicable.

Unfortunately, the arity of polymorphic types is difficult to predict, as type variables in a polymorphic type can be instantiated with functional types, thereby increasing the arity of the type. On the other hand, the arity of monomorphic type is well-defined, since they do not contain type variables.

We will show that, for a monomorphically typed program, the arity of expressions produced by driving is bound, because the arity of their types is bound.

The subtle point, however, is that although the supercompilation algorithm implemented in HOSC takes into account the fact that the input program is typable, it does not explicitly deal with concrete types.

Thus, we can use the following trick. Given a polymorphically typed input program *prog*, we can replace it with a monomorphically typed $prog_m$, such that HOSC produces the same process tree for *prog* and $prog_m$. Hence, if the arities of applications in $\mathcal{M}_{\mathcal{E}_3}(prog_m)$ are bound, then the same is true of $\mathcal{M}_{\mathcal{E}_3}(prog)$, just because $\mathcal{M}_{\mathcal{E}_3}(prog_m) = \mathcal{M}_{\mathcal{E}_3}(prog)$.

In the following section we describe the *monomorphization* procedure for a source program *prog*.

4.4.1 Monomorphization

In order to infer types according to Hindley-Milner type system, a graph of function calls is constructed [12], to produce a directed graph consisting from *strongly connected components* sorted in a reversed topological order. Functions within a certain component are monomorphic in a sense that all occurrences of a function name defined in this component have the same type (possibly with type variables). Functions defined in a component SCC_j do not depend on functions defined in components SCC_i for $i < j$. We also assume that there is a special component SCC_0 corresponding to the target expression.

Monomorphization transforms a program *prog* into a new program $prog_m$ *operationally equivalent to the original one*. At the beginning the program $prog_m$ is the same as the original program *prog*. We assume that a type is *explicitly* assigned to every subexpression and function name in $prog_m$.

Let A be an arbitrary, but fixed, *base* type, say defined as

```
data A = A;
```

Every step of monomorphization reduces the dependency graph of strongly connected components in the following way.

1. If some types assigned to the target expression and its subexpressions contain type variables, we replace these type variables with A, thereby making the target expression monomorphic. Thus, the component SCC_0 becomes monomorphic.

2. Choose a component SCC_i with a minimal $i > 0$. If there is no such component then, monomorphization is completed.

3. If the component SCC_0 does not depend on SCC_i, delete SCC_i from the dependency graph and go to Step 2.

   (a) Embed an *instance* of the component SCC_i into the component SCC_0 in the following way. Take *one* occurrence of a function name f defined in SCC_i. This occurrence has a monomorhic type since SCC_0 is already monomorphic. Replace this occurrence with f_i, where f_i is a new function name. Copy the function definitions $f \ldots g$ from SCC_i to SCC_0 renaming the functions to $f_i \ldots g_i$. If the component SCC_i has function names with types containing type variables that do not depend on the context, then instantiate these type variables with the base type A. At this step the component SCC_0 remains monomorphic: the occurrence f within SCC_0 was monomorphic, so (in the context of this occurrence) we unambiguously assigned monomorphic types to all (sub)expressions within the copied definitions.

   If the component SCC_0 still refers to SCC_0, repeat 3(a). Otherwise, go to Step 2.

Lemma 25. *Monomorphization of a program terminates in a finite number of steps.*

Proof. The number of vertices (strongly connected components) and the number of edges (call dependencies) in the original graph are finite. Step 1 is executed once and doesn't change these numbers. Step 2 doesn't change the number of vertices and the number of edges. Performing Step 3 deletes a vertex so the number of edges is reduced. For a certain SCC_i, Step 3(a) is performed while there are occurrences of function names defined in SCC_i and used in SCC_0. But the number of such occurrences if finite, and performing Step 3(a) eliminates one of them. □

In other words, monomorphization of a program terminates, because the graph of inter-component dependencies is acyclic, and each occurrence of a

function name defined in a component has the same type within this component.

Lemma 26. *The process tree produced by driving a monomorphized version of a program is the same as the tree produced by driving the original program (modulo the indices in function names).*

Proof. The monomorphized versions of function definitions $f_1, \ldots, f_i$ are the same as the original definition f (modulo indices in function names). On the other hand, the driving algorithm implemented in HOSC does not depend on concrete function names. Thus, by erasing indices in a process tree produced by driving a monomorphized program $prog_m$, we obtain the tree that is produced by driving the original program $prog$. □

4.4.2 Boundedness of arity

Lemma 27. *The arities of applications in expressions produced by driving a program prog is bounded by the maximal arity of types assigned to subexpressions in the monomorphized program* $prog_m$.

Proof. Consider the process tree produced by driving $prog_m$. All expressions labeling the nodes are monomorphic. So it is sufficient to show that the arities of types assigned to all subexpressions produced by driving are bounded. This is true of the expression in the root node, which is the target expression of $prog_m$. Let us show that the boundedness of the arities of applications is preserved by a driving step.

- D_1, D_2, D_3, D_8. Replacing an expression with one of its subexpressions cannot increase the maximum arity of types related to this expression.
- D_4. The maximal arity after this step is not greater than that at the previous step and is not greater than the maximal arity of type of subexpressions in the definition of f_0.
- D_5, D_6, D_7. The maximal arity of type cannot increase since $type(v_0) = type(e_0)$, $type(v_{ik}) = type(e'_k)$, $type(v\ \overline{e'_j}) = type(p_i)$, correspondingly.

Thus, the arity of applications produced by driving $prog_m$ is bounded. So, by Lemma 26, the arity of applications produced by driving $prog$ is bounded as well. □

Corollary 28. *The set of constructors* $App_2, \ldots, App_n$ *appearing in* $\mathcal{M}_{\mathcal{E}_3}(prog)$ *is finite.*

Theorem 29. $(\mathcal{M}(prog), \trianglelefteq^3)$ *is a well-quasi order.*

e	$\mathcal{E}_3[\![e]\!]$
`map f`	`App2(map, f)`
`map (compose f g)`	`App2(map, App3(compose, f, g))`
`λx → Cons x Nil`	`λ→ Cons(`1_2`, Nil)`
`case x of {Z →x; S b →f (g b);}`	`CaseNat(x, x, λ→ App2(f, App2(g,` 1_3`)))`

Figure 5: Examples of encoding

Proof. Consider $\mathcal{M}_{\mathcal{E}_3}(prog)$. The set of constructors in $\mathcal{M}_{\mathcal{E}_3}(prog)$ is finite due to Lemmas 16, 19 and 27. Thus, $(\mathcal{M}_{\mathcal{E}_3}(prog), \trianglelefteq)$ is a well-quasi order. So it follows from Lemma 22 that $(\mathcal{M}(prog), \trianglelefteq^3)$ is a well-quasi order. □

Corollary 30. *Since $\trianglelefteq^3$ coincides with $\trianglelefteq|_\rho$, we have shown that $\trianglelefteq|_\rho$ is a well-quasi order on $\mathcal{M}(prog)$.*

Corollary 31. *The relation $\trianglelefteq_c|_\rho$ is a well-quasi order on $\mathcal{M}(prog)$.*

Proof. Follows from the theorem 29 and Higman's lemma [3]. □

4.5 Encoding $\mathcal{E}_3$

There are examples of embedding and non-embedding via $\trianglelefteq|_\rho$ in section 4.5 in [6]. Encodings $\mathcal{E}_3$ of some expressions from mentioned examples are shown in Fig 5.

An encoding similar to $\mathcal{E}_1$ was used in [10] to solve name-capture problem for deforestation.

Encoding expressions via $\mathcal{E}_3$ eliminates the need for mapping ρ for the extended embedding $\trianglelefteq|_\rho$.

5 Termination of the supercompiler HOSC 1.1

The theorems and proofs of this section are modified versions of theorems and proofs in [14].

The idea is to show that the conditions of the theorem 12 are satisfied.

We will not consider operation $fold$ explicitly, since in [6] this operation was only used to make the algorithm of residual program construction less cumbersome. From now we assume that a leaf is a processed one if conditions for performing a $fold$ step hold.

Lemma 32. *The supercompiler HOSC is a Cauchy transformer.*

Proof. Define the relation $\succeq$ on $\mathcal{L}$ as follows:

$$let\ x'_1 = e'_1, \ldots, x'_m = e'_m\ in\ e' \succeq let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \Leftrightarrow m = 0\ \&\ n \geq 0$$

It is a routine to show that $\succeq$ a well-founded quasi-order. Let us show that at every step i of partial process tree construction

$$t_{i+1} = t_i\{\gamma := t'\}$$

for some leaf γ, and one of the following conditions is satisfied:

- $\gamma \in leaves(t_i)$ and $\gamma.expr = t'.root.expr$ (driving),
- $\gamma.expr \succ t'.root.expr$ (generalization).

It suffices to consider each of the operations that can be performed by HOSC:

1. $t_{i+1} = drive(t_i, \beta) = t_i\{\beta := t'\}$, where $\beta \in leaves(t_i)$ and $t' = \beta.expr \to e_1, \ldots, e_n$. Then:

$$\beta.expr = t'.root.expr$$

2. $t_{i+1} = abstract(t_i, \beta, \alpha) = t_i\{\beta := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \to\}$, where $\alpha = ancestor(t,\ \beta,\ \lessdot)$, $\alpha.expr \not\approx \beta.expr$, β is a non-trivial node, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \{\}, \theta) = \alpha.expr \sqcap \beta.expr$, $\beta.expr = e\theta$. Then $\alpha.expr \equiv e$ and $\beta.expr \equiv \alpha.expr\theta$, but $\alpha.expr \not\approx \beta.expr$, that is $n > 0$. Thus:

$$\beta.expr \succ let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e = t'.root.expr$$

3. $t_{i+1} = abstract(t_i, \alpha, \beta) = t_i\{\alpha := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \to\}$, where $\alpha = ancestor(t,\ \beta,\ \trianglelefteq_c)$, $\alpha.expr\ /\!\lessdot \beta.expr$, β is a non-trivial node, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $\alpha.expr = e\theta_1$. Then $\alpha.expr \not\equiv e$, but $\alpha.expr = e\theta_1$, that is $n > 0$. Thus:

$$\alpha.expr \succ let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e = t'.root.expr$$

□

Definition 33 (Set of expressions at a driving step)**.** *Set of expressions at a driving step $\mathcal{M}^i_{\mathcal{SC}}(prog)$ is defined as a set of expressions labeling a tree t_i at i-th step of partial process tree construction.*

Definition 34 (Set of expressions of supercompilation)**.** *Set of expressions of supercompilation $\mathcal{M}_{\mathcal{SC}}(prog)$ is defined as a union of sets:*

$$\mathcal{M}_{\mathcal{SC}}(prog) = \bigcup \mathcal{M}^i_{\mathcal{SC}}(prog)$$

Lemma 35. *$(\mathcal{E} \cap \mathcal{M}_{\mathcal{SC}}(prog), \trianglelefteq^3)$ is a well-quasi-order.*

Proof. Consider $\mathcal{M}_{\mathcal{E}_3}(prog)$. It is a routine to show that generalization $e = let\ \overline{v_i = e_i};\ in\ e_g$ does not increase de Bruijn indices, de Bruijn subindices and maximal arity of application. It follows from Lemma 29 that $(\mathcal{E} \cap \mathcal{M}_{\mathcal{SC}}(prog), \trianglelefteq^3)$ is a well-quasi-order. □

Corollary 36. *The relation $\trianglelefteq|_\rho$ is a well-quasi-order on $\mathcal{E} \cap \mathcal{M}_{\mathcal{SC}}(prog)$.*

Proof. Analogously to the corollary 30. □

Corollary 37. *The relation $\trianglelefteq_c|_\rho$ is a well-quasi-order on $\mathcal{E} \cap \mathcal{M}_{\mathcal{SC}}(prog)$.*

Proof. Analogously to the corollary 31. □

Lemma 38. *HOSC maintains a finitary, continuous predicate when constructing a partial process tree.*

Proof. Define $l : \mathcal{L} \to \mathcal{E}$ by:

$$l(let\ \overline{v_i = e_i};\ in\ e_g) = e_g\{\overline{v_i := e_i}\}$$

Define $\sqsupseteq$ on $\mathcal{L}$ by:

$$e_1 \sqsupseteq e_2 \Leftrightarrow (\mathcal{S}[\![l(e_1)]\!] > \mathcal{S}[\![l(e_2)]\!]) \vee (\mathcal{S}[\![l(e_1)]\!] = \mathcal{S}[\![l(e_2)]\!] \wedge l(e_1) \gtrdot l(e_2))$$

Since $\lessdot$ is a well-founded quasi-order, it is a routine check that $\sqsubseteq$ is also a well-founded quasi-order. Consider the predicate $q : T(l) \to \mathbb{B}$ defined as:

$$q(t) = p(t^0)$$

where t^0 – is an interior of t and the predicate $p : T(l) \to \mathbb{B}$:

$$p(t) = \begin{cases} 0 & \text{if } \exists \alpha, \beta : \alpha = ancestor(t,\ \beta,\ \trianglelefteq_c) \text{ and } \alpha, \beta \text{ are non-trivial} \\ 0 & \text{if } \exists \alpha, \beta : \alpha \to \beta,\ \alpha, \beta \text{ are trivial are } \alpha.expr \not\trianglerighteq \beta.expr \\ 1 & \text{otherwise} \end{cases}$$

The sets of non-trivial and trivial expressions constitute a partition of $\mathcal{M}_{\mathcal{SC}}(prog)$. $\trianglelefteq_c|_\rho$ – is a well-quasi-order on the set of non-trivial expressions of $\mathcal{M}_{\mathcal{SC}}(prog)$ and $\sqsubseteq$ is a well-founded quasi-order on the set of trivial expressions of $\mathcal{M}_{\mathcal{SC}}(prog)$. Thus (see Proposition 11), p and q are finitary and continuous predicates. Let us show that HOSC maintains q.

Given a tree t and a node β, we say that β is *good* in t if the following conditions both hold:

(i) β is non-trivial, β is not a leaf of $t \Rightarrow$
$ancestor(t, \beta, \trianglelefteq_c) = \bullet$

(ii) $\exists \alpha : \alpha \to \beta$ and α is trivial $\Rightarrow \alpha.expr \sqsupset \beta.expr$

A tree t is *good* if all its leaves are good.

It is easy to see that $q(t) = 1$ if t is good. Thus it suffices to show that for any i t_i is good. We proceed by induction on i.

For $i = 0$, (i)-(ii) are both vacuously satisfied.

For $i > 0$, suppose that t_i is good. Consider different operations for producing t_{i+1}:

1. $t_{i+1} = drive(t_i, \beta) = t_i\{\beta := t'\}$, where $\beta \in leaves(t_i)$ and $t' = \beta.expr \to e_1, \ldots, e_n$ and $e_1, \ldots, e_n = \mathcal{D}[\![\beta.expr]\!]$. We need to verify that β and $children(t_{i+1}, \beta)$ are good in t_{i+1}.

   Consider β: (1) if β is non-trivial, the algorithm guarantees that condition (i) is satisfied for β, condition (ii) follows from the induction hypothesis, (2) if β is trivial, then condition (i) is satisfied in a trivial way for β, condition (ii) follows from the induction hypothesis.

   Consider $children(t_{i+1}, \beta)$: (1) if β is non-trivial, conditions (i) and (ii) are vacuously satisfied for $children(t_{i+1}, \beta)$ (2) if β is trivial then condition (i) is satisfied in a trivial way, and condition (ii) also holds since $\forall i : \beta.expr \sqsupset e_i$, – it is important, that driving of a trivial node results in nodes with expressions of a smaller size (see 6).

2. $t_{i+1} = abstract(t_i, \beta, \alpha) = t_i\{\beta := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \to\}$, where $\alpha = ancestor(t,\ \beta,\ \lessdot)$, $\alpha.expr \not\simeq \beta.expr$, β is non-trivial, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \{\}, \theta) = \alpha.expr \sqcap \beta.expr$, $\beta.expr = e\theta$. We need to check that β is good in t_{i+1}.

   Condition (i) is satisfied in a trivial way. From the induction hypothesis and the fact that $l(\beta_i.expr) = l(let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e) = l(\beta_{i+1}.expr)$, it follow that condition (ii) holds.

3. $t_{i+1} = abstract(t_i, \alpha, \beta) = t_i\{\alpha := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \to\}$, where $\alpha = ancestor(t,\ \beta,\ \trianglelefteq_c)$, $\alpha.expr\ /\!\lessdot \beta.expr$, β is non-trivial, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $\alpha.expr = e\theta_1$. Analogously to the prevous case: we need to check that α is good t_{i+1}.

   Condition (i) is satisfied in a trivial way. From the induction hypothesis and the fact that $l(\alpha_i.expr) = l(let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e) = l(\alpha_{i+1}.expr)$, it follow that condition (ii) holds.

□

Corollary 39. *The construction of a partial process tree by the supercompiler HOSC terminates.*

Proof. From Lemmas 32 and 38 it follows that the conditions of the theorem 12 hold. □

```
data D = F (D → D);

λf → apply (F (λx → f (apply x x)))(F (λx → f (apply x x)))
where

apply = λx → case x of { F f → f; };
```

Figure 6: Fixed point operator using a global definition

```
data D = F (D → D);

λf → (λy → case y of { F g → g; })
    (F (λx → f ((λy → case y of { F g → g; }) x x)))
       (F (λx → f ((λy → case y of { F g → g; }) x x)))
```

Figure 7: Fixed point operator without global definitions

6 Possible non-termination of HOSC 1.0

The supercompiler HOSC 1.0 may non-terminate. Consider the sample program in Fig. 6, in which the fixed point operator is defined without using explicit recursion. On this program by HOSC 1.0 terminates, because non-trivial nodes are encountered (and checked by whistle) while constructing the partial process tree.

```
apply (F (λx → f (apply x x)))(F (λx → f (apply x x)))
```

However, if we unfold the calls to `apply` by inlining its definition, HOSC 1.0 does not terminate any more! The reason is that driving the modified program (Fig. 7) produces an infinite sequence of trivial nodes, which are not examined by the whistle.

This subtle problem is fixed in HOSC 1.1 (see Appendix B).

7 Related work

The first *complete* and *formal* description of a supercompiler with a proof of its termination can be found in Sørensen's Master's Thesis [13]. Sørensen's supercompiler deals with a first-order call-by-name functional language. Later Sørensen developed a framework for proving termination of program transformers (working with trees) [14].

Mitchell and Runciman developed a supercompiler Supero for a subset of Haskell programming language (call-by-name) [11]. A proof of termination of

Supero was not published.

Jonsson and Nordlander developed a supercompiler for a higher-order functional language with the call-by-value semantics [5] and proved its termination [4].

This work differs from [11, 5] in the following.

- When checking for homeomorphic embedding, HOSC makes the distinction between free and bound variables: bound variables may only be embedded into *corresponding* bound variables. So the whistle used in HOSC blows less frequently than the whistles in [11, 5], which results in fewer over-generalizations.
- HOSC's whistle blows only for expressions that are related by coupling at the top level. It ensures the existence of a non-trivial most specific generalization. So HOSC always performs a generalization in an unambiguous way. In [11, 5] the whistle blows even if the expressions are related by diving, which leads to certain ambiguity in the process of generalization.
- In [11, 5] all global definition are assumed to be of fixed arity, and all applications of global functions are required to be saturated. In order to eliminate partial applications of global functions, additional λ-abstractions are inserted in the program. HOSC does not have such limitation and allows any (well-typed) partial applications.

The main contribution of the present work is the proof that the extended homeomorphic embedding (making distinction between free and bound variables) is still a well-quasi-order on the set of expressions *produced by supercompiling a program* (rather than on an arbitrary set of expressions). This is sufficient, however, for ensuring termination of supercompilation. In addition, the use of a weaker homeomorphic relation enables the supercompiler to avoid over-generalization in some subtle cases.

Acknowledgements

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

References

[1] 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.

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

[3] Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1-2):69–116, 1987.

[4] 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.

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

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

[7] 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. Springer, 2010.

[8] M. Leuschel. On the power of homeomorphic embedding for online termination. In *Static Analysis*, volume 1503 of *LNCS*, pages 230–245. Springer, 1998.

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

[10] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 154–165. Springer, 1993.

[11] 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.

[12] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

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

[14] 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.

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

[16] V. F. Turchin. The algorithm of generalization in the supercompiler. In *Partial Evaluation and Mixed Computation*, 1988.

A HOSC 1.0: minor bugs and errors

In [6] there are a few minor bugs and misprints.

Section 5.2 should begin as follows:

Common functor rule has the form:

$$(v, \{v := e_1\} \cup \theta_1, \{v := e_2\} \cup \theta_2) \Rightarrow \ldots$$

and can only be applied on condition that

$$e_1 \trianglelefteq_c e_2 \text{ or } e_1 \trianglelefteq_v e_2$$

The 7th rule of driving (Section 6, Fig. 5) should be the following:

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

B HOSC 1.1: what is different from HOSC 1.0

In HOSC 1.1 the division of nodes into trivial and non-trivial ones is somewhat different from that in HOSC 1.0, taking into account the *sizes of expressions*.

The size of an expression e is denoted by $\mathcal{S}[\![e]\!]$ and is inductively defined as follows:

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

A node β is said to be *non-trivial* if one of the following conditions is satisfied:

1. $\beta.expr = con\langle f\rangle$,
2. $\beta.expr = con\langle case\ v\ \overline{e_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$,
3. $\beta.expr = con\langle(\lambda v \to e_0)\ e_1\rangle$ and $\mathcal{S}[\![(\lambda v \to e_0)\ e_1]\!] \leq \mathcal{S}[\![e_0\{v\ :=\ e_1\}]\!]$,
4. $\beta.expr = con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\}\rangle$ and $\mathcal{S}[\![case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\}]\!] \leq \mathcal{S}[\![e_j\{\overline{v_{jk}\ :=\ e'_k}\}]\!]$

Otherwise, β is said to be *trivial*.

Unlike HOSC 1.0, a non-trivial node satisfying condition 3 or 4 can be considered as a base or repeat node in a partial process tree. Thus the algorithm of constructing residual programs has been modified: the rules corresponding to non-trivial nodes satisfying condition 3 or 4 are completely analogously to the rules C_7^1, C_7^2 and C_7^3 in [6].

C Driving rules

Since driving rules presented in [6] have to be individually referenced in the text, we reproduce them here, providing rule names.

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

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

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

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

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

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

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

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