On the Termination of Higher-Order Positive Supercompilation

Geoff Hamilton
hamilton@computing.dcu.ie

Lero@DCU
School of Computing
Dublin City University
Dublin, Ireland

VPT 2013

Outline

Background

- This work is inspired by the supercompilation transformation algorithm developed by Turchin.
- Although originally developed in the early 1970s, supercompilation did not become more widely known outside Russia until much later:
  - Published only in less accessible journals.
  - Defined on the unconventional language Refal.
- Supercompilation became more widely known through the positive supercompilation algorithm (Sørensen, Glück and Jones):
  - Simplified algorithm.
  - Defined on a more familiar functional language.

Motivation

- The verification of unfold/fold program transformation systems requires that we prove their termination.
- This can be done using the theory of local improvement (Sands) provided that each folding step follows from a corresponding unfolding step.
- This is the case for first-order positive supercompilation since folding is attempted prior to each unfolding step.
- This is not the case for higher-order positive supercompilation since folding must be attempted at other steps also e.g., $(\lambda x \rightarrow x\ x)\ (\lambda x \rightarrow x\ x)$.
- Checking all previous expressions encountered during transformation is very expensive.
- We would like to reduce the number of expressions which need to be checked while still ensuring termination.

Language

We use the following higher-order functional language:

$prog$	$::= e_0$ **where** $f_1 = e_1 \ldots f_n = e_n$	Program
e	$::= x$	Variable
	$\mid c\ e_1 \ldots e_n$	Constructor
	$\mid f$	Function Call
	$\mid \lambda x \to e$	λ-Abstraction
	$\mid e_0\ e_1$	Application
	$\mid$ **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n$	Case Expression
	$\mid$ **let** $x = e_0$ **in** e_1	Let Expression
p	$::= c\ x_1 \ldots x_n$	Pattern

Positive Supercompilation: Overview

- At the heart of the positive supercompilation algorithm are a number of driving rules which reduce a term (possibly containing free variables) using normal-order reduction.
- Function unfolding is performed as a part of this reduction process, and folding is performed upon encountering a renaming of a memoised expression.
- To ensure the termination of the transformation, generalization is performed when an expression is encountered which is a homeomorphic embedding of a memoised expression.

Divergence

- In the absence of generalization, the sequence of terms encountered can diverge and non-termination can occur.
- Four different possible causes of non-termination of higher-order positive supercompilation:
  1. Obstructing function calls
  2. Accumulating parameters
  3. Accumulating narrowing
  4. Accumulating spine

Obstructing Function Calls

Consider the following program:

$$
\begin{array}{l}
\mathit{nrev}\ \mathit{xs} \\
\textbf{where} \\
\mathit{nrev} \quad = \quad \lambda \mathit{xs} \rightarrow \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nil} \qquad\quad \Rightarrow \mathit{Nil} \\
\qquad\qquad\qquad\quad \mid \mathit{Cons}\ x'\ \mathit{xs}' \Rightarrow \mathit{app}\ (\mathit{nrev}\ \mathit{xs}')\ (\mathit{Cons}\ x'\ \mathit{Nil}) \\
\mathit{app} \quad = \quad \lambda \mathit{xs} \rightarrow \lambda \mathit{ys} \rightarrow \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nil} \qquad\quad \Rightarrow \mathit{ys} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{Cons}\ x'\ \mathit{xs}' \Rightarrow \mathit{Cons}\ x'\ (\mathit{app}\ \mathit{xs}'\ \mathit{ys})
\end{array}
$$

During transformation, we encounter the progressively larger terms:
nrev xs
case (*nrev xs*) **of** $\cdots$
case (**case** (*nrev xs*) **of** $\cdots$) **of** $\cdots$
etc.

Accumulating Parameters

Consider the following program:

$$
\begin{array}{l}
\mathit{arev}\ \mathit{xs} \\
\textbf{where} \\
\mathit{arev} \quad = \quad \lambda \mathit{xs} \rightarrow \mathit{arev}'\ \mathit{xs}\ \mathit{Nil} \\
\mathit{arev}' \quad = \quad \lambda \mathit{xs} \rightarrow \lambda \mathit{ys} \rightarrow \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nil} \qquad\quad \Rightarrow \mathit{ys} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{Cons}\ x'\ \mathit{xs}' \Rightarrow \mathit{arev}'\ \mathit{xs}'\ (\mathit{Cons}\ x'\ \mathit{ys})
\end{array}
$$

During transformation, we encounter the progressively larger terms:
$\mathit{arev}'\ \mathit{xs}\ \mathit{Nil}$
$\mathit{arev}'\ \mathit{xs}'\ (\mathit{Cons}\ x'\ \mathit{Nil})$
$\mathit{arev}'\ \mathit{xs}''\ (\mathit{Cons}\ x''\ (\mathit{Cons}\ x'\ \mathit{Nil}))$
etc.

Accumulating Narrowing

Consider the following program:

$$
\begin{array}{l}
\mathit{app}\ \mathit{xs}\ \mathit{xs} \\
\textbf{where} \\
\mathit{app} \quad = \quad \lambda \mathit{xs} \rightarrow \lambda \mathit{ys} \rightarrow \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nil} \qquad\quad \Rightarrow \mathit{ys} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Cons}\ \mathit{x}\ \mathit{xs} \Rightarrow \mathit{Cons}\ \mathit{x}\ (\mathit{app}\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

During transformation, we encounter the progressively larger terms:
$\mathit{app}\ \mathit{xs}\ \mathit{xs}$
$\mathit{app}\ \mathit{xs}'\ (\mathit{Cons}\ \mathit{x}'\ \mathit{xs}')$
$\mathit{app}\ \mathit{xs}''\ (\mathit{Cons}\ \mathit{x}'\ (\mathit{Cons}\ \mathit{x}''\ \mathit{xs}''))$
etc.

Accumulating Spine

Consider the following program:

$$
\begin{array}{l}
f\ x \\
\textbf{where} \\
f \quad = \quad \lambda x \rightarrow f\ x\ x
\end{array}
$$

During transformation, we encounter the progressively larger terms:
$f\ x$
$f\ x\ x$
$f\ x\ x\ x$
etc.

Homeomorphic Embedding

- The homeomorphic embedding relation $\lesssim$ is a well-quasi order which provides a so-called whistle to stop driving due to potential divergence, and to indicate that generalization should be performed.
- An expression is embedded within another by this relation if either diving (denoted by $\trianglelefteq_d$) or coupling (denoted by $\trianglelefteq_c$) can be performed.
- Diving occurs when an expression is embedded in a sub-expression of another expression.
- Coupling occurs when two expressions have the same top-level construct and all the corresponding sub-expressions of the two constructs are embedded.

Homeomorphic Embedding

$$\frac{e_1 \trianglelefteq_c e_2}{e_1 \trianglelefteq e_2} \qquad \frac{e_1 \trianglelefteq_d e_2}{e_1 \trianglelefteq e_2}$$

$$x \trianglelefteq_c x \qquad f \trianglelefteq_c f$$

$$\frac{\forall i \in \{1 \ldots n\}.e_i \trianglelefteq e'_i}{(c\ e_1 \ldots e_n) \trianglelefteq_c (c\ e'_1 \ldots e'_n)} \qquad \frac{\exists i \in \{1 \ldots n\}.e \trianglelefteq e_i}{e \trianglelefteq_d (c\ e_1 \ldots e_n)}$$

$$\frac{e \trianglelefteq (e'\{x' \mapsto x\})}{\lambda x.e \trianglelefteq_c \lambda x'.e'} \qquad \frac{e \trianglelefteq e'}{e \trianglelefteq_d \lambda x.e'}$$

$$\frac{e_0 \trianglelefteq e'_0 \quad e_1 \trianglelefteq e'_1}{(e_0\ e_1) \trianglelefteq_c (e'_0\ e'_1)} \qquad \frac{\exists i \in \{0,1\}.e \trianglelefteq e_i}{e \trianglelefteq_d (e_0\ e_1)}$$

$$\frac{e_0 \trianglelefteq e'_0 \quad \forall i \in \{1 \ldots n\}.\exists \sigma.p_i \equiv (p'_i\ \sigma) \wedge e_i \trianglelefteq (e'_i\ \sigma)}{(\textbf{case}\ e_0\ \textbf{of}\ p_1 \to e_1 | \ldots | p_n \to e_n) \trianglelefteq_c (\textbf{case}\ e'_0\ \textbf{of}\ p'_1 \to e'_1 | \ldots | p'_n \to e'_n)}$$

$$\frac{\exists i \in \{0 \ldots n\}.e \trianglelefteq e_i}{e \trianglelefteq_d (\textbf{case}\ e_0\ \textbf{of}\ p_1 \to e_1 | \ldots | p_n \to e_n)}$$

Homeomorphic Embedding

- The homeomorphic embedding relation $\lesssim$ is defined as: $e_1 \lesssim e_2$ iff $\exists \sigma . e_1 \sigma \trianglelefteq_c e_2$
- σ is a renaming, so there is no longer a requirement that all of the free variables in the two expressions match.
- Some examples of homeomorphic embedding are as follows:

1.	$f\ (g\ x) \lesssim f\ (g\ y)$	6.	$f\ (g\ x) \not\lesssim g\ (f\ y)$
2.	$f\ (h\ x) \lesssim f\ (g\ (h\ y))$	7.	$g\ (h\ x) \not\lesssim f\ (g\ (h\ y))$
3.	$f\ x\ y \lesssim f\ z\ z$	8.	$f\ z\ z \not\lesssim f\ x\ y$
4.	$f\ x\ x \lesssim f\ (g\ y)\ (h\ y)$	9.	$f\ (g\ y)\ (h\ y) \not\lesssim f\ x\ x$
5.	$\lambda x.x \lesssim \lambda y.y$	10.	$\lambda x.x \not\lesssim \lambda y.x$

Generalization

- The generalization of two expressions e_1 and e_2 is a triple $(e_g, \theta_1, \theta_2)$ where 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 generalization we define for expressions e_1 and e_2 is the most specific generalization, denoted by $e_1 \sqcap e_2$.
- The most specific generalization of expressions e_1 and e_2 is a generalization $(e_g, \theta_1, \theta_2)$ such that for every other generalization $(e'_g, \theta'_1, \theta'_2)$ of e_1 and e_2, e_g is an instance of e'_g.

Generalization

$$x \sqcap x = x \qquad f \sqcap f = f$$

$$(c\ e_1 \ldots e_n) \sqcap (c\ e'_1 \ldots e'_n) = (c\ e^g_1 \ldots e^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i)$$
where $\forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i$

$$(\lambda x.e_0) \sqcap (\lambda x'.e'_0) = (\lambda x.e^g_0, \theta_0, \theta'_0)$$
where $(e^g_0, \theta_0, \theta'_0) = e_0 \sqcap (e'_0\{x' \mapsto x\})$

$$(e_0\ e_1) \sqcap (e'_0\ e'_1) = (e^g_0\ e^g_1, \theta_0 \cup \theta_1, \theta'_0 \cup \theta'_1)$$
where
$(e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0$
$(e^g_1, \theta_1, \theta'_1) = e_1 \sqcap e'_1$

$$(\textbf{case}\ e_0\ \textbf{of}\ p_1 \to e_1 | \ldots | p_n \to e_n) \sqcap (\textbf{case}\ e'_0\ \textbf{of}\ p'_1 \to e'_1 | \ldots | p'_n \to e'_n)$$
$$= (\textbf{case}\ e^g_0\ \textbf{of}\ p_1 \to e^g_1 | \ldots | p_n \to e^g_n, \bigcup_{i=0}^n \theta_i, \bigcup_{i=0}^n \theta'_i)$$
where
$(e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0$
$\forall i \in \{1 \ldots n\}.\exists \sigma.p_i \equiv (p'_i\ \sigma) \wedge (e^g_i, \theta_i, \theta'_i) = e_i \sqcap (e'_i\ \sigma)$

$e \sqcap e' = (x, \{x \mapsto e\}, \{x \mapsto e'\})$ in all other cases (x is fresh)

Generalization

Some examples of applying this most specific generalization:

e_1	e_2	$e_1 \sqcap e_2$
$f\ (g\ x)$	$f\ (g\ y)$	$(f\ g\ v, \{v \mapsto x\}, \{v \mapsto y\})$
$f\ (h\ x)$	$f\ (g\ (h\ y))$	$(f\ v, \{v \mapsto h\ x\}, \{v \mapsto g\ (h\ y)\})$
$f\ x\ y$	$f\ z\ z$	$(f\ v_1\ v_2, \{v_1 \mapsto x, v_2 \mapsto y\}, \{v_1 \mapsto z, v_2 \mapsto z\})$
$f\ x\ x$	$f\ (g\ y)\ (h\ y)$	$(f\ v_1\ v_2, \{v_1 \mapsto x, v_2 \mapsto x\}, \{v_1 \mapsto g\ y, v_2 \mapsto h\ y\})$
$\lambda x.x$	$\lambda y.y$	$(\lambda x.x, \{\}, \{\})$

When to Memoise

- For a first-order language, it is sufficient to memoise only expressions immmediately prior to an unfolding step.
  - This is sufficient to ensure termination (Sørensen).
  - Showing local improvement (Sands) can be used to prove termination and no loss of efficiency.
- For a higher-order language, this is not sufficient to prove termination as other expressions need to be memoised.
  - Local improvement cannot be used to prove termination.
  - Also cannot be used to prove no loss of efficiency.
  - For example: $(\lambda x \to x\ x)\ (\lambda x \to x\ x)$
  - Another example:

$$\textbf{data}\ \mathrm{D} = \mathrm{F}\ (\mathrm{D} \to \mathrm{D})$$

$$(\lambda f \to f\ (F\ (\lambda x \to f\ x\ x))\ (F\ (\lambda x \to f\ x\ x)))\ (\lambda y \to \textbf{case}\ y\ \textbf{of}\ F\ g \to g)$$

Reducing the Need for Memoisation

To ensure that any potentially infinite sequence of transformation steps always includes function unfolding, we require that programs are in λ–prefix form:

$prog$	$::= pf_0$ **where** $f_1 = pf_1 \ldots f_n = pf_n$	Program
pf	$::= \lambda x \rightarrow pf$	λ-Abstraction
	$\mid pf'$	λ-Free Expression
pf'	$::= x$	Variable
	$\mid c\ pf'_1 \ldots pf'_n$	Constructor
	$\mid f$	Function Call
	$\mid pf'_0\ pf'_1$	Application
	$\mid$ **let** $x = pf'_0$ **in** pf'_1	Let Expression
	$\mid$ **case** pf'_0 **of** $p_1 \Rightarrow pf'_1 \mid \cdots \mid p_n \Rightarrow pf'_n$	Case Expression
p	$::= c\ x_1 \ldots x_n$	Pattern

Reducing the Need for Memoisation

It is quite straightforward to convert any program into this form:

- Replace λ-abstractions which are not in the prefix of the program expression or the prefix of a function body with a freshly named function.
- λ-lifting is also performed to abstract over any of the free variables in the λ-abstraction.
- If the λ-abstraction matches one which has already been replaced by a function call, then it is replaced with a call to the same function as previously.

Example

The following program:

$$(\lambda x \to x\ x)\ (\lambda x \to x\ x)$$

is transformed to the following:

$$\begin{array}{l} f\ f \\ \textbf{where} \\ f \quad = \quad \lambda x \to x\ x \end{array}$$

Thus, any potentially infinite sequence of transformation steps would have to include the unfolding of f.

Example

$$\textbf{data}\ \mathrm{D} = \mathrm{F}\ (\mathrm{D} \to \mathrm{D})$$

$$(\lambda f \to f\ (F\ (\lambda x \to f\ x\ x))\ (F\ (\lambda x \to f\ x\ x)))\ (\lambda y \to \textbf{case}\ y\ \textbf{of}\ F\ g \to g)$$

is transformed to the following:

$$\begin{array}{l} f_1\ f_2 \\ \textbf{where} \\ f_1 = \lambda f \to f\ (F\ (f_3\ f))\ (F\ (f_3\ f)) \\ f_2 = \lambda y \to \textbf{case}\ y\ \textbf{of}\ F\ g \to g \\ f_3 = \lambda f \to \lambda x \to f\ x\ x \end{array}$$

Thus, any potentially infinite sequence of transformation steps would have to include the unfolding of f_2 and f_3.

Proving Termination

To prove termination, we firstly show that if the original input to our positive supercompilation algorithm is in λ-prefix form, then all of the terms subsequently encountered must be in the following form:

$$\begin{aligned} sf ::= & \ \textbf{case}\ sf\ \textbf{of}\ p_1 \Rightarrow pf'_1 \mid \cdots \mid p_n \Rightarrow pf'_n \\ & \mid sf\ pf' \\ & \mid pf \end{aligned}$$

- where pf and pf' are as defined earlier (pf can contain λ-abstractions; pf' cannot).
- The only λ-abstractions are in the prefix of the redex.

We then prove that if all the terms encountered are in the above form, then every infinite sequence of transformation steps must include function unfolding.

Proving Termination

Finally, we prove that if the input to our positive supercompilation algorithm is in λ-prefix form, then it is guaranteed to terminate.

- If our positive supercompilation algorithm did not terminate then the set of memoised expressions must be infinite, since every infinite sequence of transformation steps must include function unfolding.
- Every new expression which is memoised cannot have any of the previously memoised expressions embedded within it by the homeomorphic embedding relation $\lesssim$, since folding or generalization would have been performed instead.
- This contradicts the fact that $\lesssim$ is a well-quasi-order.

Related Work

- In the higher-order formulations of positive supercompilation given by Mitchell and Bolingbroke, all expressions are memoised.
- In the formulations given by Jonsson and Mendel-Gleason, only expressions encountered prior to an unfolding step are memoised, but all types must be positive.
- In the higher-order supercompiler HOSC by Klyuchnikov, only those expressions which are considered to be non-trivial are memoised.
  - The definition of what constitutes a non-trivial expression has had to be changed.
  - The definition in HOSC 1.0 did not memoise enough and will not terminate on some of our example programs.
  - The definition in HOSC 1.1 memoised too much and produced poor residual programs.

Conclusions

- We have shown a simple approach which can be applied to programs prior to transformation by higher-order positive supercompilation so that only expressions encountered prior to an unfolding step need to be memoised.
- Using our approach, less new functions will be created and generalization will be performed less often, resulting in more improved residual programs.
- We have implemented the techniques described in this paper and preliminary experiments show that they make the resulting supercompiler more efficient.
- We are not able to prove that the programs resulting from transformation are an improvement over the original.
- We are also using this approach in the implementation of our distillation algorithm.