Multi-result supercompilation as a tool for program analysis

Ilya Klyuchnikov and Sergei A. Romanenko
(Keldysh Institute of Applied Mathematics, Moscow)

Luleå University of Technology, Sweden
April 28, 2011
Program analysis by supercompilation

Let SC be a semantics-preserving supercompiler:

\[ e' = SC[e] \Rightarrow e' \equiv e \]

The idea: instead of analyzing \( e \), we may analyze \( SC[e] \).

- \( SC[e] \) may be easier to understand than \( e \).
- Some hidden properties of \( e \) may become apparent in \( SC[e] \).

This is an instance of *transformational approach* to program analysis. (In principle, any semantics-preserving program transformer can be used.)
An example: proving that all values returned by $e$ satisfy $p$

Suppose that

- $e$ is an expression.
- $p$ is a predicate.
- $e$ and $p$ are written in the same language.
- $SC$ is a semantics-preserving supercompiler.

How to prove that anything returned by $e$ satisfies $p$? Consider the program $p \ e$, and supercompile it!

If we are lucky, $SC[p \ e]$ is just (the constant) $\text{True}$. Thus, $p \ e \equiv SC[p \ e] = \text{True}$. 
SC[ρ e] in action: the verification of protocols


SCP 4 : Verification of Protocols
http://refal.botik.ru/protocols/
Goals of supercompilation

- "Efficiency" can be measured (in bytes and seconds).
- "Understandability" is more difficult to measure and to formalize.

A problem: how to explain the goal to an analyzing SC? What is "good" and what is "bad"?
First-order vs. higher-order SC


- Deals with Refal, a functional language.
- First-order.
- Call-by-value.
- Does not preserve termination properties of programs.

HOSC (Klyuchnikov, [http://code.google.com/p/hosc/](http://code.google.com/p/hosc/))

- Deals with HLL, a subset of Haskell.
- Higher-order
- Call-by-name.
- Preserves the semantics of programs.
Why higher-order & call-by-name?

- A program is considered as a specification/formalization/model of something.
- A program is to be analyzed, rather than executed.

Higher-order:

- Specifications/models can be written in DSLs, implemented by combinators.
- Higher-order logics (quantifiers over functions/predicates).

Call-by-name:

- Termination properties are easier to preserve.
- Infinite data structures are useful for writing specifications of infinite processes.
Higher-order SC: Church numbers (1)

Notation: $f^0 x = x$, $f^k x = f(f^{k-1} x)$.

data Nat = Z | S Nat;

Peano numbers: $S^k Z$.
Church numbers: $\lambda s \ z \rightarrow s^k z$.

unchurch = \n -> n S Z;

foldn = \s \ z \ x -> case x of {
    Z -> z;
    S x1 -> s (foldn s z x1); };

church = \n -> foldn (\m \ f \ x -> f (m f x)) (\f \ x -> x) n;
Higher-order SC: Church numbers (2)

\[\text{add} = \lambda x \ y \to \text{foldn} \ S \ y \ x;\]
\[\text{mult} = \lambda x \ y \to \text{foldn} \ (\text{add} \ y) \ Z \ x;\]
\[\text{churchAdd} = \lambda m \to n \to \lambda s \to \lambda z \to m \ s \ (n \ s \ z);\]
\[\text{churchMult} = \lambda m \ n \ f \to m \ (n \ f);\]

A problem: are the following expressions equivalent?

\[(\text{mult} \ x \ y)\]
\[\text{unchurch}((\text{churchMult} \ (\text{church} \ x) \ (\text{church} \ y)))\]
Higher-order SC: Church numbers (3)

By supercompiling the expressions with HOSC

\[(\text{mult } x \ y)\]
\[(\text{unchurch} (\text{churchMult} (\text{church } x) (\text{church } y)))\]

we get the same residual program (modulo a renaming)!

\[
\text{letrec}
\text{f=} (\\text{\ s6\ ->}
\quad (\\text{\ t6\ ->}
\quad \quad \text{case } \text{s6} \text{ of } \{ 
\quad \quad \quad \text{Z } \rightarrow \text{Z};
\quad \quad \quad \text{S } u6 \rightarrow \text{letrec } g= (\\text{x7\-> \ case } \text{x7} \text{ of } \{ 
\quad \quad \quad \quad \text{S } v \rightarrow (\text{S } (g \ v));
\quad \quad \quad \quad \text{Z } \rightarrow \text{f u6 t6; } \}
\quad \quad \quad \text{in } (g \ t6);)
\quad \quad \}
\quad \text{in } (g \ t6);)
\quad \text{in } \text{f x y}
\]
Proving the equivalence of expressions by means of supercompilation (1)

Let $\forall e, e \cong SC[e]$, i.e. SC is semantics-preserving.

Proof technique:

$$SC[e_1] \equiv SC[e_2] \Rightarrow e_1 \cong e_2$$

Justification (close to a tautology):

$$e_1 \cong SC[e_1] \equiv SC[e_2] \equiv e_2$$

A problem: SC may fail to guess, which versions of residual programs to produce, in order for them to be identical (modulo a renaming).
Proving the equivalence of expressions by means of supercompilation (2)


Restrictions:

- All functions must be total (since SCP4 does not preserve termination properties).
- First-order logic (no quantifiers over functions/predicates, since Refal is a first-order language).
- No infinite data structures (Refal is a call-by-value language).
Proving the equivalence of expressions by means of supercompilation (3)


The technique was shown to work even if

- Functions may non-terminate.
- Free variables may be of functional types.
- Data may be infinite.

How? By *constructing* HOSC, a supercompiler that is really capable of "catching the mice".
Another example: Proving the equivalence of small-step & big-step abstract machines


An example of *transformational approach* to program analysis.

A **manual** proof by Danvy & Millikin: a non-trivial sequence of program transformations from the first program to the second one.

A proof **by supercompilation**: just supercompile two programs by HOSC to get the same residual program! (See [live](#).)
Tuning SC for program analysis (1)


Gives a refined definition of homeomorphic embedding taking into account the difference between free and bound variables.

HOSC 1.0 does not terminate for some input programs (an example)!


HOSC 1.1 always terminates.
Extended homeomorphic embedding

Classic embedding

Variables

\[
\begin{align*}
\exists i : e &\leq e' \\
v_1 &\leq v_2 \\
v_1 &\leq \phi(e'_1, \ldots, e'_k) \\
\phi(e_1, \ldots, e_k) &\leq \phi(e'_1, \ldots, e'_k)
\end{align*}
\]

Diving

\[
\begin{align*}
e' &\leq e'' |_\rho \\
\text{if } e' &\leq_v e'' |_\rho \text{ or } e' &\leq_d e'' |_\rho \text{ or } e' &\leq_c e'' |_\rho
\end{align*}
\]

Coupling

Extended embedding

Variables

\[
\begin{align*}
f &\leq_v f |_\rho \\
v_1 &\leq v_2 |_\rho \\
v_1 &\leq_v v_2 |_\rho \\
v_1 &\leq_v v_2 |_\rho \\
v_1 &\not\in \text{domain}(\rho) \text{ and } v_2 &\not\in \text{range}(\rho)
\end{align*}
\]

Diving

\[
\begin{align*}
\forall v &\in f v(e) : v \notin \text{domain}(\rho) \\
e &\leq_d c e_i |_\rho \\
e &\leq_d \lambda v_0 \rightarrow e_0 |_\rho \\
e &\leq_d e_i |_\rho \\
e &\leq_d \text{case } c_0 \text{ of } \{c_i \ v_{ik} \rightarrow e_i;\} |_\rho \\
e &\leq_d \text{case } c_0 \text{ of } \{c_i \ v_{ik} \rightarrow e_i;\} |_\rho \\
e &\leq_d \text{case } c_0 \text{ of } \{c_i \ v_{ik} \rightarrow e_i;\} |_\rho
\end{align*}
\]

Coupling

\[
\begin{align*}
\overline{c e_i} &\leq_c c \overline{e_i} |_\rho \\
\lambda v_1 &\rightarrow e_1 \leq_c \lambda v_2 \rightarrow e_2 |_\rho \\
e' e'_i &\leq_c e'' e'' |_\rho \\
\text{case } e' \text{ of } \{c_i \ v_{ik} \rightarrow e_i;\} &\leq_c \text{case } e'' \text{ of } \{c_i \ v_{ik} \rightarrow e_i;\}_| \rho
\end{align*}
\]

\[
\begin{align*}
&\text{if } \forall i : e'_i \leq e'' |_\rho \\
&\text{if } e_1 \leq e_1 |_{\rho \cup \{(v_1, v_2)\}} \\
&\text{if } e' \leq e'' |_\rho \text{ and } \forall i : e'_i \leq e'' |_\rho \\
&\text{if } e' \leq e'' |_\rho \text{ and } \forall i : e'_i \leq e'' |_{\rho \cup \{(v_{ik}, v_{ik})\}}
\end{align*}
\]
Extended embedding is well-quasi-order

Theorem (Kruskal, Higman). For any infinite sequence of expressions $e_1, e_2, \ldots e_n, \ldots$ there are $i<j$, such that $e_i \vartriangleleft e_j$

Extended whistle doesn't blow for ANY sequence!

Theorem (Klyuchnikov). For any infinite sequence of expressions $e_1, e_2, \ldots e_n, \ldots$, appearing on a branch of partial process tree $t$, there are $i<j$, such that $e_i \vartriangleleft^* e_j$
The proof of the pudding is in the eating

HOSC 0, using the "classic" homeomorphic embedding, proved only 6 of 25 equivalences (from the first chapter).

HOSC 1, using "extended" homeomorphic embedding, is able to prove 25 of 25 equivalences.
Tuning SC for program analysis (2)


HOSC 1.5 is a revised (and simplified) version of HOSC 1.1.

The first published algorithm for finding a most specific generalization for expressions *with bound variables*.

The definition of homeomorphic embedding should take into account the possibility of the following *generalization*!

Unlike $x$ and $Sx$, there is **no embedding** for $\lambda x.x$ and $\lambda x.Sx$!
"Higher-level" supercompilation?

An idea:

- The power of "ordinary" ("basic"?, "ground"?) is limited.
- Let us consider supercompilation as a "primitive operation" and construct a "metasystem" (in V.F. Turchin's terms).

The "ground" supercompilers are controlled by the metalevel (which, eventually, may be a supercompiler as well).
Examples of higher-level SC

- Futamura projections: $SC_3[SC_2[SC_1]]$.
  - $SC_2$ simulates the execution of $SC_1$, and this is controlled by $SC_3$.
- Proving the equivalence of expressions.
  - $SC[e_1] \equiv SC[e_2]$.
- Proving improvement lemmas ($e_1$, $e_2$).
  - $SC[e_1] \equiv SC[e_2]$ and tick annotations in $SC[e_1]$ are embedded in tick annotations in $SC[e_2]$.
- Two-level supercompilation.
  - The "upper" supercompiler applies improvement lemmas checked by means of the "ground" supercompiler.
- Distillation (Hamilton, ...).
The idea:

- Mark in the residual program with "ticks" (the points where there has been an unfolding step during driving).
- Check two residual expressions for "homeomorphic embedding" with respect to "ticks".
Checking improvement lemmas (2)

Annotating a partial process tree with "ticks"

```
letrec f=
  let x=n in
  case even n of {True → True; False → odd x;}
  case (even n) of {True → True; False → odd x;}
  case (case n of {Z → True; S y → odd y;})
  of {True → True; False → odd x;}
  n = Z n = S y
  True case (odd y) of {True → True; False → odd x;}
  * case (case y of {Z → False; S z → even z;})
  of {True → True; False → odd x;}
  y = Z y = S z
  odd x case (even z) of {True → True; False → odd x;}
  S x → f x;};} in f n
```
What is new:

- An explicit algorithm for generating the residual program annotated with ticks from a partial process tree.
- An improved algorithm for comparing tick annotations based on normalization of ticks.
Two-level supercompilation (1)

The goal is to avoid generalization!

- The whistle blows for $\alpha$ and $\beta$.
- $\alpha$ (or $\beta$?) has to be generalized.
- A generalization is an evil, as it causes some loss of information.
Two-level supercompilation (2)

The goal is to avoid generalization!

- Let \( \gamma \) be an expression such that
  - \((\beta, \gamma)\) is an improvement lemma;
  - the whistle is silent for \( \alpha \) and \( \gamma \).
- By replacing \( \beta \) with \( \gamma \), we can avoid generalization!
Two-level supercompilation (3)

def scp0(e) = {
    ...
    if whistle(e1, e2)
        abstract(e1, e2)
    ...
}

def scp1(e) = {
    ...
    if whistle(e1, e2)
        e3 = findEquiv(e1)
        if e3 != null
            replace(e1, e3)
        else
            abstract(e1, e2)
    ...
}

def findEquiv(e1) = {
    for c <- candidates(e1)
        if scp0(e1) == scp0(c)
            return c
    return null
}
Two-level supercompilation (4)

How to speed up the search for lemmas and make the lemmas "friendlier"?


Some tricks related to

- finding improvement lemmas by inspecting and manipulating the expressions that have already appeared in the partial process tree;
- extracting "human-friendly" (and more abstract) lemmas from the lemmas produced automatically (which are often cumbersome and too specific).
Two-level supercompilation (5)

**Theorem** (Sørensen, 1994). Classical positive supercompiler for a call-by-name language cannot improve the asymptotic of a program.

However, as shown in


an $O(n^2)$ parser corresponding to the grammar

\[ p = a \ p \ a \ | \ empty. \]

can be transformed by a 2-level supercompiler to an $O(n)$ parser corresponding to the grammar:

\[ p' = a \ a \ p' \ | \ empty. \]
$O(n^2)$ $\Rightarrow$ $O(n)$: source parser

**Complexity: $O(n^2)$**

data Symbol = A | B;
data List a = Nil | Cons a (List a);
data Option a = Some a | None;

match doublea word \textbf{where}

\begin{align*}
\text{match} &= \lambda p \ i \to p \ (\text{eof} \ \text{return}) \ i; \\
\text{return} &= \lambda x \to \text{Some} \ x; \\
\text{doublea} &= \text{or} \ \text{nil} \ (\text{join} \ a \ (\text{join} \ \text{doublea} \ a)); \\
\text{or} &= \lambda p1 \ p2 \ \text{next} \ w \to \text{case} \ p1 \ \text{next} \ w \ \text{of} \ \{ \ \text{Some} \ w1 \to \text{Some} \ w1; \\
& \quad \text{None} \to p2 \ \text{next} \ w; \}; \\
\text{nil} &= \\lambda \text{next} \ w \to \text{next} \ w; \\
\text{join} &= \lambda p1 \ p2 \ \text{next} \ w \to p1 \ (p2 \ \text{next}) \ w; \\
\text{a} &= \\lambda \text{next} \ w \to \text{case} \ w \ \text{of} \ \{ \ \text{Nil} \to \text{None}; \\
& \quad \text{Cons} \ s \ w1 \to \text{case} \ s \ \text{of} \ \{ \ A \to \text{next} \ w1; \ B \to \text{None};\};\}; \\
\text{b} &= \\lambda \text{next} \ w \to \text{case} \ w \ \text{of} \ \{ \ \text{Nil} \to \text{None}; \\
& \quad \text{Cons} \ s \ w1 \to \text{case} \ s \ \text{of} \ \{ \ A \to \text{None}; \ B \to \text{next} \ w1;\};\}; \\
\text{eof} &= \\lambda \text{next} \ w \to \text{case} \ w \ \text{of} \ \{ \ \text{Cons} \ s \ w1 \to \text{None}; \ \text{Nil} \to \text{next} \ \text{Nil};\};
\end{align*}
$O(n^2) \Rightarrow O(n):$ residual parser

Ordinary SC, complexity $O(n^2)$

case word of {
  Cons y9 t5 ->
  case word of { Cons w13 w9 ->
    case w13 of {
      A -> (letrec f=(\r21-> (\s21-> case r21 of { Cons r3 y5 ->
        case r3 of { A -> case (s21 y5) of { Some z7 -> (Some z7);
                      None -> ((f y5) (\s8->
                        case s8 of {
                          Cons z5 s18 -> case z5 of { A -> (s21 s18); B -> None; };
                          Nil -> None;
                        }));
                      }; B -> None;
                      }; Nil -> None;}})
        in
        ((f w9) (\v16-> case v16 of { Cons t6 w2 -> None; Nil -> (Some Nil); })))
    }; Nil -> None;
  }; Nil -> (Some Nil);}
\( O(n^2) \Rightarrow O(n): \text{residual parser} \)

2-level SC, complexity \( O(n) \)

\[
\text{letrec}
\]
\[
f = (\lambda s14 ->
    \text{case } s14 \text{ of }
    \text{Cons } z12 y8 \rightarrow
    \text{case } z12 \text{ of }
    \text{A } \rightarrow \text{case } y8 \text{ of }
    \text{Cons } s3 s2 \rightarrow \text{case } s3 \text{ of }
    \text{A } \rightarrow (f s2); \text{B } \rightarrow \text{None};
    \text{Nil } \rightarrow \text{None};
    \text{B } \rightarrow \text{None};
    \text{Nil } \rightarrow (\text{Some Nil});
    \}
\text{in}
\]
\[
f \text{ word}
\]
Supercompilation relation

\[ e' = SC[e] \Rightarrow e \ SC_{rel} e' \]


The purpose: theoretical (proofs of correctness).
Klyuchnikov: \( HOSC_0 \supseteq HOSC_{1/2} \supseteq HOSC \).
Deterministic vs. nondeterministic SC

A taxonomy of supercompilation

- **Deterministic SC:**
  \[ e' = \text{SC}[e] \]
  an operation (a single \(e')\).

- **Nondeterministic SC:**
  \[ e \text{ SC } e' \]
  a relation (one or more \(e')\).

- **Multi-result SC:**
  \[ \text{MSC}[e] \subseteq \{ e' \mid e \text{ SC } e' \} \land \text{MSC}[e] \neq \emptyset \]
  an operation (a non-empty set of residual programs).

For practical purposes, it is desirable for \(\text{MSC}[e]\) to be finite.
From determinism to non-determinism

Non-determinism is popular in the field of *model-checking*.

A model is produced by throwing away "irrelevant" details.

A typical situation:

```
if p then e1 else e2
```

By abstracting away the condition $p$, we get

```
choice { e1; e2; }
```

A *tree* of possible states at run-time (instead of a *sequence*).  

Nondeterministic SC

The supercompilation relation SC can be formulated as a nondeterministic program:

\[ t = e_0 \]
\[ \text{while } \text{incomplete}(t) \text{ do} \]
\[ \text{beta} = \text{unprocessedLeaf}(t) \]
\[ t = \text{choice} \{
\]
\[ \text{drive}(t, \text{beta}); \]
\[ \text{generalize}(t, \text{beta}); \]
\[ \text{fold}(t, \text{beta}); \]
\[ \text{fail}; \} \]
\[ \text{end} \]

In this way we abstract away the whistle and the strategies. But they come back in a multi-result MSC!
Problems with MSC

- How to make MSC\([\text{e}]\) finite?
  - An answer: by killing partial process trees that make a whistle blow.
- How to (automatically) reduce the size of MSC\([\text{e}]\)?
  - An answer: by "normalizing" residual programs and mergin ones with "insignificant" differences.
  - An answer: by analyzing residual programs and throwing away ones that are "dull" and/or "uninteresting".

MSC opens a new area of research (rather than gives a "final solution" to a problem).

A simple implementation of MSC (the branch "multi"): https://github.com/ilya-klyuchnikov/spsc-lite-scala
Proving equivalences by means of MSC

Let $e' \in MSC[e] \Rightarrow e' \equiv e$, i.e. a multi-result supercompiler $MSC$ is semantics-preserving.

**Proof technique:**

$$\exists e' \in MSC[e_1] \cap MSC[e_2] \Rightarrow e_1 \equiv e_2$$

**Justification** (close to a tautology):

$$e' \in MSC[e_1] \cap MSC[e_2] \Rightarrow$$

$$e' \in MSC[e_1] \& e' \in MSC[e_2] \Rightarrow$$

$$e' \equiv e_1 \& e' \equiv e_2 \Rightarrow e_1 \equiv e_2$$
Proving equivalences by transitivity

The principle: $e_1 \equiv e_2 \& e_2 \equiv e_3 \Rightarrow e_1 \equiv e_3$

Proof technique:

- Suppose we have failed to prove $e_1 \equiv e_3$.
- Let us pick up an expression $e_2$.
- Suppose we are able to prove both $e_1 \equiv e_2$ and $e_2 \equiv e_3$.
- Then we conclude that $e_1 \equiv e_3$.

Let us check $\equiv$ by means of MSC!
How to use MSC for proving equivalences by transitivity?

An implementation:

- Suppose $MSC[e_1] \cap MSC[e_3] = \emptyset$.
- Let us pick up an expression $e_2$.
- If there are
  
  $e' \in MSC[e_1] \cap MSC[e_2]$  
  $e'' \in MSC[e_2] \cap MSC[e_3]$,
- then $e_1 \equiv e_3$. 

![Venn diagram showing the intersection of MSC sets](image)
A deterministic SC is unable to prove \( e_1 \cong e_3 \), if \( SC[e_1] \not\equiv SC[e_3] \)!

\[
\begin{align*}
SC[e_1] &\equiv SC[e_2] \not\equiv SC[e_3] \\
SC[e_1] &\not\equiv SC[e_2] \equiv SC[e_3]
\end{align*}
\]

Just a speculation. No interesting examples yet. :-(
But they are bound to be found by our new postgraduates. :-(
MSC and 2-level supercompilation

Some interesting possibilities:

- MSC instead of SC at the lower level.
  - More lemmas can be found.
- MSC instead of SC at the upper level.
  - Several different lemmas can be tried at the same node.

**An idea.** Residual programs can be ranked according to their "non-triviality".

- The more improvement lemmas have been applied during 2-level supercompilation, the less trivial is the residual program!
MRSC: a framework for creating multi-result supercompilers


Parameterized over

- the object language;
- the language of configurations;
- driving, whistle, generalization.

Provides a number of combinators to produce multi-result and two-level supercompilers from ordinary ones.
Conclusions

- Supercompilation can be treated as a "primitive operation" in order to build more complex system. This is an instance of "metasystem transition" (in terms of V.F. Turchin).
- By abstracting away the whistle and the strategies, we get nondeterministic supercompilation (a supercompilation relation).
- By "rehabilitating" the whistle and the strategies, we remove some nondeterminism to come to multi-result supercompilation (MSC).
- MSC is more powerful in solving certain problems than deterministic one.
- MSC is a new area of research. Not much is done yet...
Thank you!