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:

$$e' = SC[e] \Rightarrow e' \cong e$$

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!

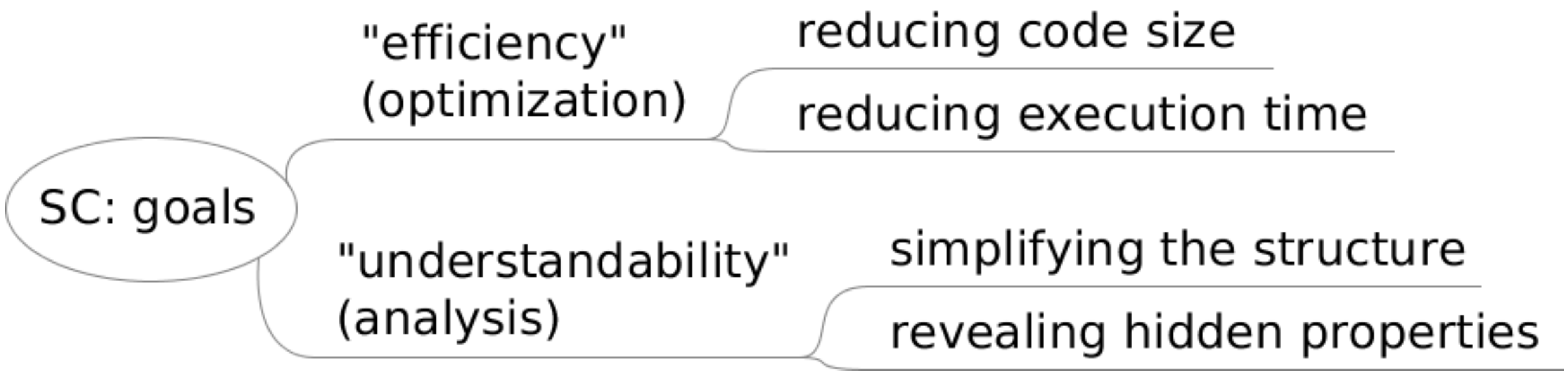
If we are lucky, *SC*[*p e*] is just (the constant) True.
Thus, p e ≅ SC[p e] = True.

SC[*p e*] in action: the verification of protocols

- Alexei Lisitsa and Andrei P. Nemytykh. **Reachability Analysis in Verification via Supercompilation.** International Journal of Foundations of Computer Science, Vol. 19, No. 4 (2008) 953-969
- Alexei Lisitsa and Andrei P. Nemytykh. **Verification as Specialization of Interpreters with Respect to Data.** In *Proceedings of First International Workshop on Metacomputation in Russia, META'2008*. Pereslavl-Zalessky, Russia, 2-5 July 2008, ISBN 978-5-901795-12-5, pp 94--112

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

SCP4 (Nemytykh, http://www.botik.ru/~scp/doc/docs.html)

- 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/)

- 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: $\mathtt{S}^k\ \mathtt{Z}$.
Church numbers: $\backslash\mathtt{s}\ \mathtt{z} \to \mathtt{s}^k\ \mathtt{z}$.

```
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)

```
add = \x y -> foldn S y x;
mult = \x y -> foldn (add y) Z x;

churchAdd = \m -> n -> \s -> \z -> m s (n s z);
churchMult = \m n f -> m (n f);
```

A problem: are the following expressions equivalent?

```
(mult x y)
unchurch(churchMult (church x) (church y))
```

Higher-order SC: Church numbers (3)

By supercompiling the expressions with HOSC

```
(mult x y)
(unchurch(churchMult (church x) (church y))
```

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

```
letrec
 f=(\s6->
   (\t6->
     case  s6  of {
       Z  -> Z;
       S u6 -> letrec g= \x7-> case  x7  of {
                 S v -> (S (g v));
                 Z  -> f u6 t6; }
                in (g t6);
     }))
in 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):

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

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)

- Alexei Lisitsa and Matt Webster. **Supercompilation for Equivalence Testing in Metamorphic Computer Viruses Detection.** In *Proceedings of First International Workshop on Metacomputation in Russia, META'2008*. Pereslavl-Zalessky, Russia, 2-5 July 2008, ISBN 978-5-901795-12-5, pp 113-118.

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)

- Ilya Klyuchnikov and Sergei Romanenko. **Proving the Equivalence of Higher-Order Terms by Means of Supercompilation.** In: *Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics*. LNCS 5947, 2009.

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

- Olivier Danvy and Kevin Millikin. 2008. **On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion.** *Inf. Process. Lett.* 106, 3 (April 2008), 100-109. [PDF](PDF) http://dx.doi.org/10.1016/j.ipl.2007.10.010

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)

- Ilya Klyuchnikov. **Supercompiler HOSC 1.0: under the hood.** Preprint 63. Keldysh Institute of Applied Mathematics, Moscow, 2009. http://pat.keldysh.ru/~ilya/

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)!

- Ilya Klyuchnikov. **Supercompiler HOSC 1.1: proof of termination.** Preprint 21. Keldysh Institute of Applied Mathematics, Moscow, 2010.

HOSC 1.1 ***always terminates***.

Extended homeomorphic embedding

Classic embedding

Variables | Diving | Coupling

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

Extended embedding

$$e' \trianglelefteq e'' \mid_\rho \quad \text{if } e' \trianglelefteq_v e'' \mid_\rho \text{ or } e' \trianglelefteq_d e'' \mid_\rho \text{ or } e' \trianglelefteq_c e'' \mid_\rho$$

Variables

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

Diving

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

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

Coupling

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

Extended embedding is well-quasi-order

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

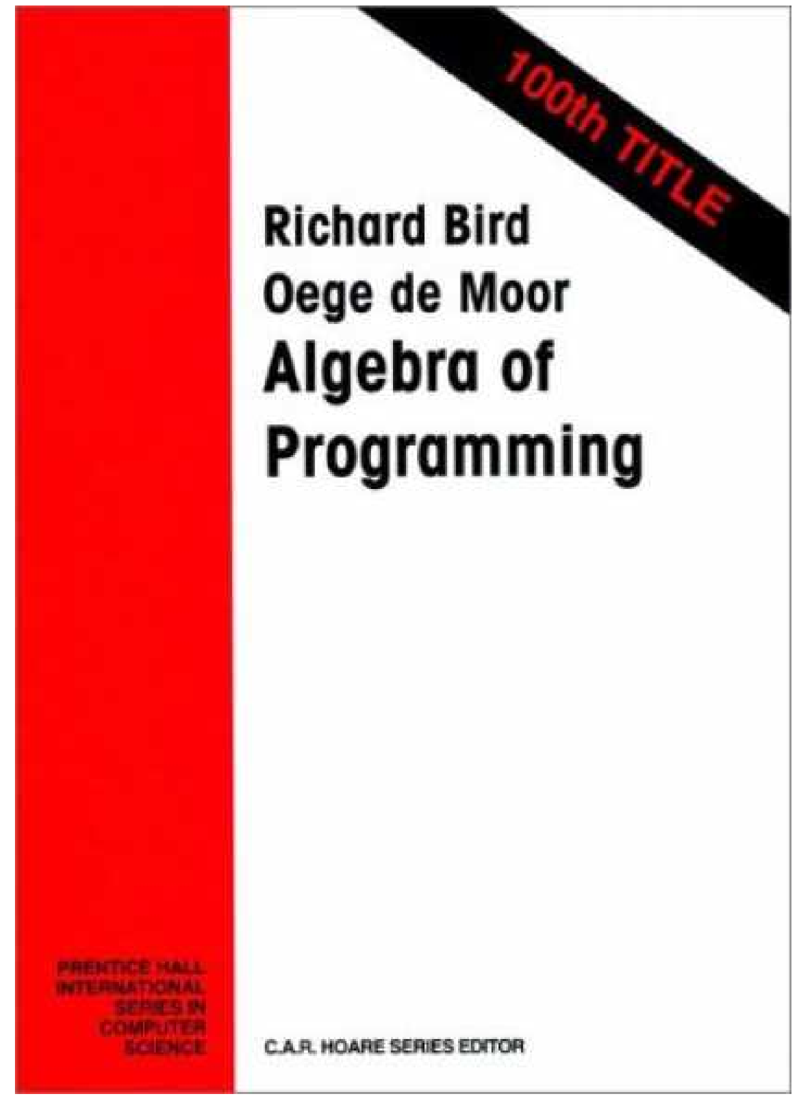
Extended whistle doesn't blow for ANY sequence!

Theorem (Klyuchnikov). For any infinite sequence of expressions e_1, e_2, ... e_n, .., appearing on ***a branch of partial process tree*** t, there are i<j, such that
$e_i \lhd^* e_j$

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)

- Ilya Klyuchnikov. **Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting.** Preprint 62. Keldysh Institute of Applied Mathematics, Moscow, 2010. http://pat.keldysh.ru/~ilya/

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***!

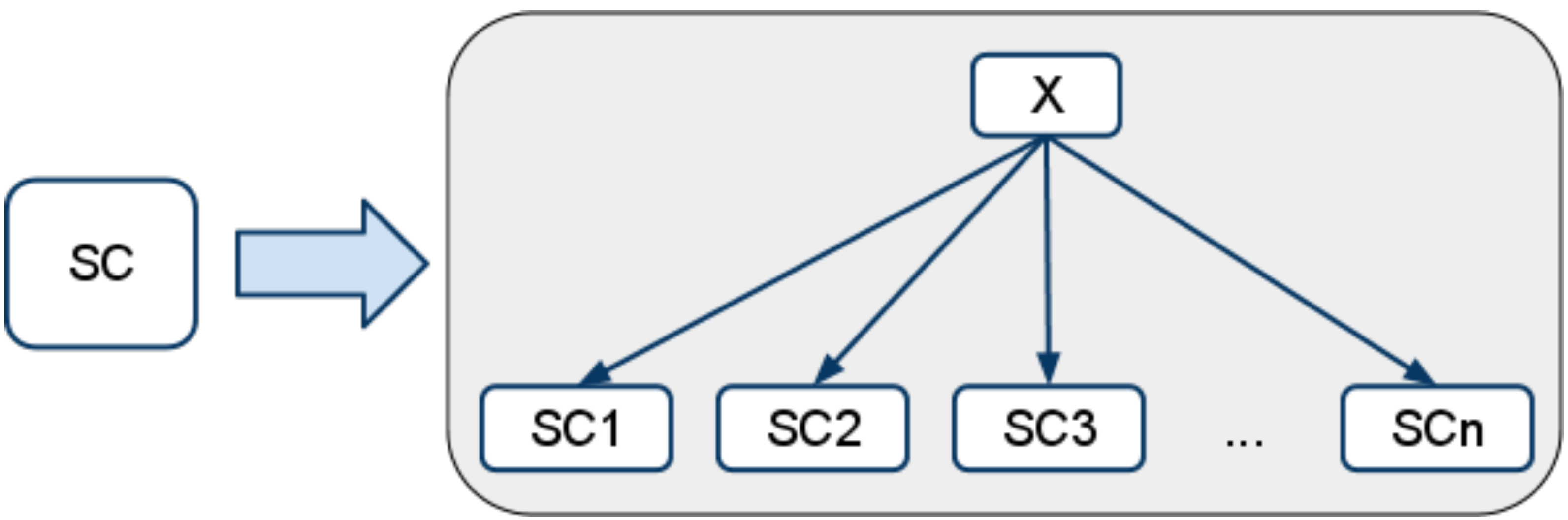
Unlike $\mathbf{x}$ and $\mathbf{S\ x}$,
there is ***no embedding*** for $\boldsymbol{\lambda}\mathbf{x.x}$ and $\boldsymbol{\lambda}\mathbf{x.S\ x}$!

"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, ...).

Checking improvement lemmas (1)

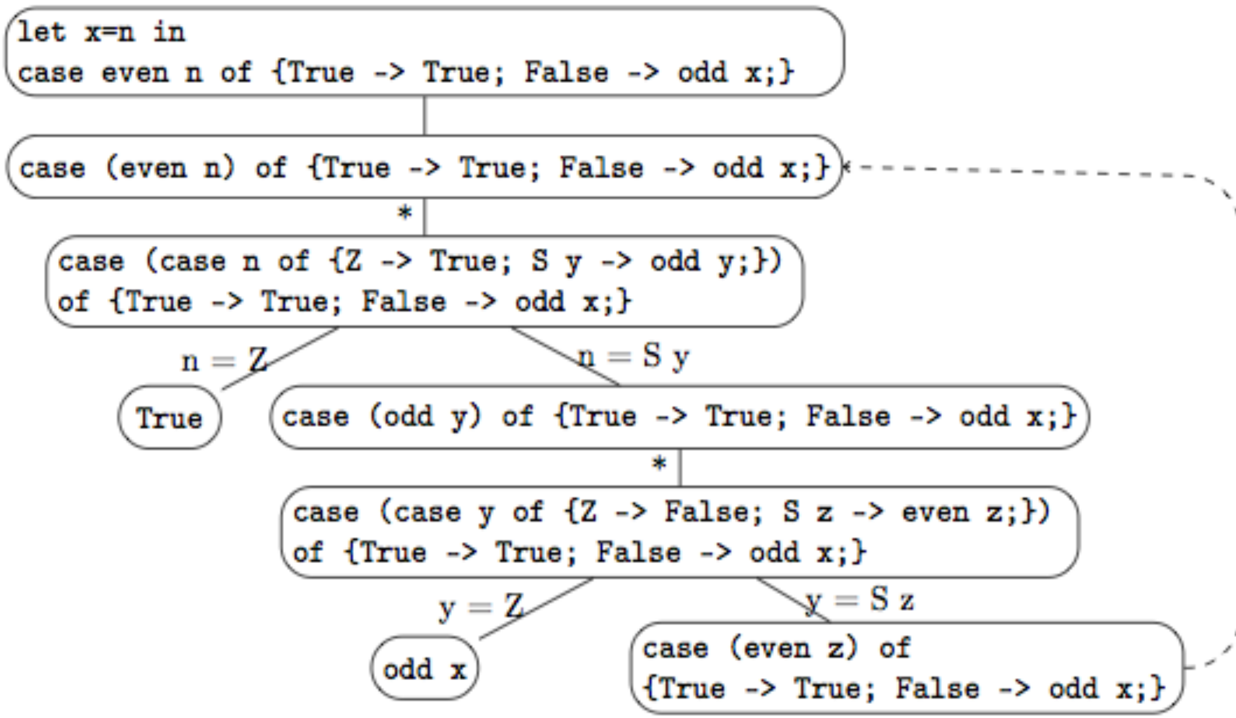
- Ilya Klyuchnikov and Sergei Romanenko. **Towards Higher-Level Supercompilation.** In Second International Workshop on Metacomputation in Russia (Proceedings of the second International Workshop on Metacomputation in Russia. Pereslavl-Zalessky, Russia, July 1-5, 2010). A. P. Nemytykh, Ed. - Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2010, 186 p. ISBN 978-5-901795-21-7, pages 82-101. http://pat.keldysh.ru/~ilya/

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"



Propagating "ticks" into the residual program

letrec f=*\v → case v of { Z → True;
  S p → *case p of { Z→ (letrec g = *\w → case w of {
      Z → False;
      S t → * case t of {Z → True; S z → g z;};} in g n;
    S x → f x;};} in f n

Checking improvement lemmas (3)

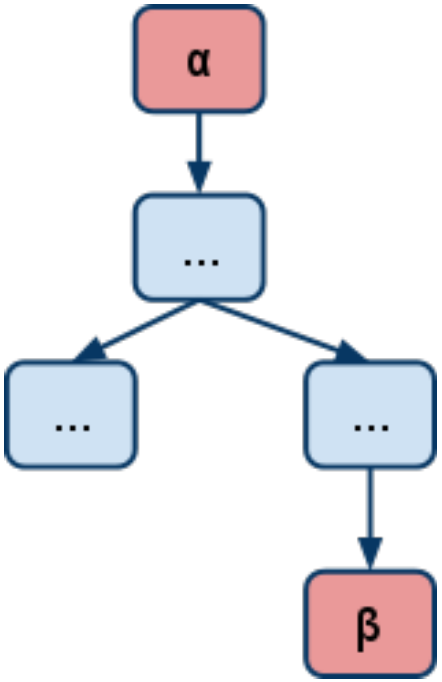
- Ilya Klyuchnikov. **Towards effective two-level supercompilation.** Preprint 81. Keldysh Institute of Applied Mathematics, Moscow. 2010. http://pat.keldysh.ru/~ilya/

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)

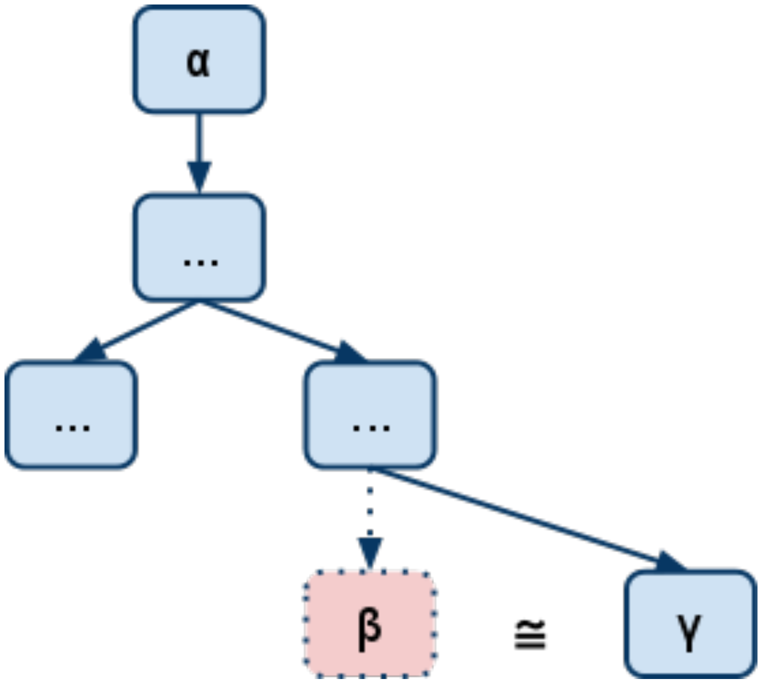
The goal is to avoid generalization!



- The whistle blows for α and β.
- α (or β?) 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 γ be an expression such that
  - (β, γ) is an improvement lemma;
  - the whistle is silent for α and γ.
- By replacing β with γ, 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
}
```

scp1 is simplified (no check for improvement).

Two-level supercompilation (4)

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

- Ilya Klyuchnikov. **Towards effective two-level supercompilation.** *Preprint 81. Keldysh Institute of Applied Mathematics, Moscow.* 2010. http://pat.keldysh.ru/~ilya/

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

- Ilya Klyuchnikov. **Towards effective two-level supercompilation.** *Preprint 81. Keldysh Institute of Applied Mathematics, Moscow.* 2010. http://pat.keldysh.ru/~ilya/

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 grammmar:

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 where

match = \p i -> p (eof return) i;
return = \x -> Some x;
doublea = or nil (join a (join doublea a));
or = \p1 p2 next w -> case p1 next w of { Some w1 -> Some w1;
   None -> p2 next w;};
nil = \next w -> next w;
join = \p1 p2 next w -> p1 (p2 next) w;
a = \next w -> case w of { Nil -> None;
   Cons s w1 -> case s of { A -> next w1; B -> None;};};
b = \next w -> case w of { Nil -> None;
   Cons s w1 -> case s of { A -> None; B -> next w1;};};
eof = \next w -> case w of { Cons s w1 -> None; Nil -> next Nil;};
```

$O(n^2)$ ⇨ $O(n)$: 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); })));
          B  -> None;
        }; Nil  -> None;
    };Nil  -> (Some Nil);
}
```

$O(n^2)$ ⇨ $O(n)$: residual parser

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

```
letrec
  f=(\s14->
    case  s14  of {
      Cons z12 y8 ->
        case  z12  of {
          A  -> case  y8  of {
            Cons s3 s2 -> case  s3  of { A  -> (f s2); B  -> None; };
            Nil  -> None; };
          B  -> None; };
      Nil  -> (Some Nil);
    }
in
  f word
```

Supercompilation relation

$$e' = SC[e] \quad ⇨ \quad e \; SC_{rel} \; e'$$

- Andrei V. Klimov. **A Program Specialization Relation Based on Supercompilation and its Properties.** In *First International Workshop on Metacomputation in Russia (Proceedings of the first International Workshop on Metacomputation in Russia. Pereslavl-Zalessky, Russia, July 2-5, 2008)*. A. P. Nemytykh, Ed. - Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008, 108 p. ISBN 978-5-901795-12-5, pages 54-77. http://pat.keldysh.ru/~anklimov/
- Ilya Klyuchnikov. **Supercompiler HOSC: proof of correctness.** Preprint 31. Keldysh Institute of Applied Mathematics, Moscow, 2010. http://pat.keldysh.ru/~ilya/

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**:
  $e' = SC[e]$
  an *operation* (a single e').
- **Nondeterministic SC**:
  $e \; SC \; e'$
  a *relation* (one or more e').
- **Multi-result SC:**
  $MSC[e] \subseteq \{ e' \mid e \; SC \; e' \}$ & $MSC[e] \neq \varnothing$
  an *operation* (a non-empty set of residual programs).

For practical purposes, it is desirable for $MSC[e]$ 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 = e0
while incomplete(t) do
  beta = unprocessedLeaf(t)
  t = choice{
              drive(t, beta);
              generalize(t, beta);
              fold(t, beta);
              fail; }
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*[*e*] finite?
  - An answer: by killing partial process trees that make a whistle blow.
- How to (automatically) reduce the size of MSC[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 $e' \in MSC[e] \Rightarrow e' \cong e$, i.e. a multi-result supercompiler *MSC* is semantics-preserving.

Proof technique:

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

Justification (close to a tautology):

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

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

$e' \cong e_1\ \&\ e' \cong e_2 \Rightarrow e_1 \cong e_2$

Proving equivalences by transitivity

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

Proof technique:

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

Le us check $\cong$ by means of *MSC*!

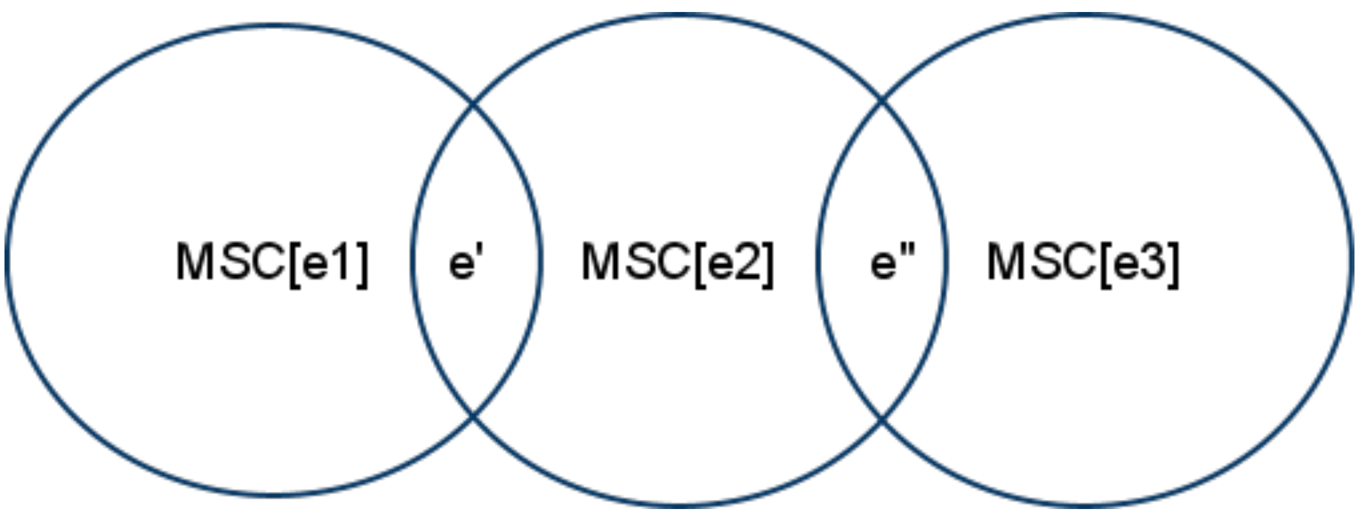
How to use *MSC* for proving equivalences by transitivity?

An implementation:

- Suppose $MSC[e_1] \cap MSC[e_3] = \varnothing$.
- 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 $e_1 \cong e_3$.



A deterministic SC is unable to prove $e_1 \cong e_3$, if $SC[e_1] \not\equiv SC[e_3]$!

$SC[e_1] \equiv \mathrm{SC}[e_2] \not\equiv SC[e_3]$

$SC[e_1] \not\equiv \mathrm{SC}[e_2] \equiv SC[e_3]$

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

- Ilya Klyuchnikov and Sergei Romanenko. Multi-Result Supercompilation as Branching Growth of the Penultimate Level in Metasystem Transitions. *Accepted for Ershov Informatics Conference 2011*. http://pat.keldysh.ru/~ilya/

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 nondetermistic 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!