Towards Higher-Level Supercompilation

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

July 2 / Meta 2010
Outline

Introduction
The concept of metasystem transition
Supercompiler HOSC

Higher-level supercompilation
Naive example
Proving lemmas by supercompilation
Algorithm

Correctness
Operational theory of improvement
Detecting improvement lemmas by supercompilation

Implementation
Higher-level HOSC

Examples
Non-linear expression
Accumulating parameter revisited
Improving the asymptotics: from $O(n^2)$ to $O(n)$
Outline

Introduction
The concept of metasystem transition
Supercompiler HOSC

Higher-level supercompilation
Naive example
Proving lemmas by supercompilation
Algorithm

Correctness
Operational theory of improvement
Detecting improvement lemmas by supercompilation

Implementation
Higher-level HOSC

Examples
Non-linear expression
Accumulating parameter revisited
Improving the asymptotics: from $O(n^2)$ to $O(n)$
The concept of metasystem transition

Consider a system $S$ of any kind. Suppose that there is a way to make some number of copies of it, possibly with variations. Suppose that these systems are united into a new system $S'$ which has the systems of the $S$ type as its subsystems, and includes also an additional mechanism which somehow examines, controls, modifies and reproduces the $S$-subsystems. Then we call $S'$ a metasystem with respect to $S$, and the creation of $S'$ a metasystem transition. As a result of consecutive metasystem transitions a multilevel hierarchy of control arises, which exhibits complicated forms of behavior.

Supercompiler HOSC

Features

- “Optimized” for program analysis. The input language is a call-by-name one.
  - Aggressively eliminates let-expressions, which increases the depth of program analysis.
  - Applicable for call-by-need. (Programs equivalent in the call-by-name setting are also equivalent in the call-by-need setting)
- Open sourced: http://hosc.googlecode.com/
- Web-interface: http://hosc.appspot.com/
Supercompiler HOSC

There is a complete and formal description

Outline

Introduction
The concept of metasystem transition
Supercompiler HOSC

Higher-level supercompilation
Naive example
Proving lemmas by supercompilation
Algorithm

Correctness
Operational theory of improvement
Detecting improvement lemmas by supercompilation

Implementation
Higher-level HOSC

Examples
Non-linear expression
Accumulating parameter revisited
Improving the asymptotics: from $O(n^2)$ to $O(n)$
Accumulating parameter

Input

data Bool = True | False;
data Nat = Z | S Nat;

even (double x Z) where

even = \lambda x \rightarrow \text{case } x \text{ of } \{ Z \rightarrow \text{True}; S \ x1 \rightarrow \text{odd } x1; \};
odd = \lambda x \rightarrow \text{case } x \text{ of } \{ Z \rightarrow \text{False}; S \ x1 \rightarrow \text{even } x1; \};
double = \lambda x \ y \rightarrow \text{case } x \text{ of } \{ Z \rightarrow y; S \ x1 \rightarrow \text{double } x1 \ (S \ (S \ y)); \};
Accumulating parameter

Driving and whistle

even (double x Z)

case (double x Z) of \{Z → True; S y → odd y;\}

case case x of \{Z → Z; S z → double z (S (S Z));\} of \{Z → True; S y → odd y;\}

x = Z

x = S n

True

case (double n (S (S Z))) of \{Z → True; S m → odd m;\}
Accumulating parameter

The result of “classic” supercompilation

\[
\text{letrec}
\]
\[
f = \lambda w2 \: \lambda p2 \rightarrow
\]
\[
\text{case } w2 \: \text{of } \{
\]
\[
Z \rightarrow
\]
\[
\text{letrec } g = \lambda r2 \rightarrow
\]
\[
\text{case } r2 \: \text{of } \{
\]
\[
S \: r \rightarrow \text{case } r \: \text{of } \{Z \rightarrow \text{False}; S \: z2 \rightarrow g \: z2;\};
\]
\[
Z \rightarrow \text{True};
\]
\[
\}
\]
\[
in \: g \: p2;
\]
\[
S \: z \rightarrow f \: z \: (S \: (S \: p2));
\]
\[
\}
\]
\[
in \: f \: x \: Z
\]

A loss in precision: actually, False can never be returned.
Accumulating parameter

Applying an “enigmatic” lemma

Lemma

case double n (S (S Z)) of {Z → True; S m → odd m;} ≅ even (double n Z)

Avoiding whistle
Accumulating parameter

Applying an “enigmatic” lemma

The result of applying the lemma

letrec $f = \lambda t \to$
\[
\begin{array}{l}
\text{case } t \text{ of } \{ Z \to \text{True}; S s \to f s; \}\n\end{array}
\]
\[\text{in } f x\]

(There are no False in the residual program!)
Proving lemmas by supercompilation
Proving lemmas by supercompilation
Proving lemmas by supercompilation

Multi-result supercompilation
Proving lemmas by supercompilation

Original idea: first-order, call-by-value


Further research: higher-order, call-by-name

“Classic” supercompiler

```
def scp0(expr) =

  ...
  if whistle(e1, history)
    abstract(e1, history)
  ...
```

2-level supercompiler

```
def scp1(expr) =

  ...
  if whistle(e1, history)
    e2 = findEquiv(e1, history)
    if e2 != null
      replace(e1, e2)
    else
      abstract(e1, history)
  ...
```

```
def findEquiv(e1, history)
  for c ← candidates(e1)
    if scp0(e1) == scp0(c)
      if whistle(c, history)
        return c
  return null
```
Outline

Introduction
The concept of metasystem transition
Supercompiler HOSC

Higher-level supercompilation
Naive example
Proving lemmas by supercompilation
Algorithm

Correctness
Operational theory of improvement
Detecting improvement lemmas by supercompilation

Implementation
Higher-level HOSC

Examples
Non-linear expression
Accumulating parameter revisited
Improving the asymptotics: from $O(n^2)$ to $O(n)$
Operational theory of improvement

David Sands, 1994

Operational approximation
An expression $e$ operationally approximates $e'$, $e \sqsubseteq e'$, if for all contexts $C$ such that $C[e]$, $C[e']$ are closed, if the evaluation of $C[e]$ terminates then so does the evaluation of $C[e']$.

Operational equivalence
An expression $e$ is operationally equivalent to $e'$, $e \simeq e'$ if $e \sqsubseteq e'$ and $e' \sqsubseteq e$.

Improvement
An expression $e$ is improved by $e'$, $e \triangleright e'$, if for all contexts $C$ such that $C[e]$ and $C[e']$ are closed, if the computation of $C[e]$ terminates using $n$ function calls, then the computation of $C[e']$ also terminates, and uses no more than $n$ function calls.
Operational theory of improvement

David Sands, 1994

Improvement lemma
A pair \((e_1, e_2)\) is an improvement lemma if \(e_1 \equiv e_2\) and \(e_1 \triangleright e_2\).

Correctness of improved fold-unfold transformation
If \((e_1, e_2)\) is an improvement lemma then the replacement of an expression \(e_1\) with an expression \(e_2\) will not violate the correctness of transformation.

How to detect (= find and prove) improvement lemmas?
**How to detect improvement lemmas?**

*Not so easy*

\[
\begin{align*}
or &= \lambda x \ y \ \rightarrow \ \text{case} \ x \ \text{of} \ \{ \ True \ \rightarrow \ True; \ False \ \rightarrow \ y; \}; \\
even &= \lambda x \ \rightarrow \ \text{case} \ x \ \text{of} \ \{ Z \ \rightarrow \ True; \ S \ x1 \ \rightarrow \ odd \ x1; \}; \\
odd &= \lambda x \ \rightarrow \ \text{case} \ x \ \text{of} \ \{ Z \ \rightarrow \ False; \ S \ x1 \ \rightarrow \ even \ x1; \}; \\
\end{align*}
\]

**Lemma**

\[
\begin{align*}
e1 &= \ or \ (even \ n) \ (odd \ n) \\
e2 &= \ text{case} \ (even \ n) \ \text{of} \ \{ True \ \rightarrow \ True; \ False \ \rightarrow \ odd \ (S \ (S \ n)); \} \\
e1 &\Rightarrow e2 \\
\end{align*}
\]

**But not an improvement lemma**

\[
\begin{align*}
n = Z: & \ e1 \ \leftrightarrow^{2} \ True, \ e2 \ \leftrightarrow^{1} \ True \\
n = S \ Z: & \ e1 \ \leftrightarrow^{5} \ True, \ e2 \ \leftrightarrow^{6} \ True \\
\end{align*}
\]
Detecting improvement lemmas by supercompilation

From theory to practice

Idea
Let us propagate the information about evaluation cost of the original expression into the residual expression.

Tick annotation
* e means that one reduction step of this residual expression corresponds to one unfolding of the original expression.
Detecting improvement lemmas by supercompilation

Annotating a partial process tree

let x=m in case even m of {True -> True; False -> odd x;}

case (even m) of {True -> True; False -> odd x;}

case (case m of {Z -> True; S y -> odd y;})
of {True -> True; False -> odd x;}

True  case (odd y) of {True -> True; False -> odd x;}

m = Z  m = S y  case (even z) of {True -> True; False -> odd x;}

y = Z  y = S z

odd x

case (case y of {Z -> False; S z -> even z;})
of {True -> True; False -> odd x;}

odd m

or (even m) (odd m)
Detecting improvement lemmas by supercompilation
Residuating annotations

*(letrec f = *(λv →
    case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          letrec g = *(λw →
            case w of {
            Z → False;
            S t → *(case t of {Z → True; S z → g z;});})
          in g m;
          S x → f x;
      });
    }));
  in f m)
Detecting improvement lemmas by supercompilation

Embedding of annotations

\[ m \geq n \quad \forall i : e_i \sim^* e'_i \]

\[ m\phi(e_1, \ldots, e_k) \sim^* n\phi(e'_1, \ldots, e'_k) \]

From embedding of annotations to improvements

\[ SC[e_1] \equiv SC[e_2] \quad SC[e_1] \sim^* SC[e_2] \]

\[ e_1 \sim e_2 \]

Theorem

Let \( e'_1 = SC[e_1] \) and \( e'_2 = SC[e_1] \). If \( e'_1 \equiv e'_2 \) and \( e'_1 \sim^* e'_2 \), then \( e_1 \sim e_2 \) and it is correct to replace \( e_1 \) by \( e_2 \) during supercompilation.
Outline

Introduction
   The concept of metasystem transition
   Supercompiler HOSC

Higher-level supercompilation
   Naive example
   Proving lemmas by supercompilation
   Algorithm

Correctness
   Operational theory of improvement
   Detecting improvement lemmas by supercompilation

Implementation
   Higher-level HOSC

Examples
   Non-linear expression
   Accumulating parameter revisited
   Improving the asymptotics: from $O(n^2)$ to $O(n)$
Problems to be solved

- Which “classic” supercompiler to use as the basis for implementing higher-level supercompilation?
- How to guarantee the correctness of transformations?
- How to generate useful lemmas?
- How to ensure the termination of higher-level supercompilation?
HLSC – A Higher-Level Supercompiler

- HOSC is used as the “ground-level” supercompiler.
- Correctness is guaranteed by only using improvement lemmas are used.
- The search for a lemma: straightforwardly, by trying all expressions whose size is less than that of the “bad” expression (expressions are ordered). – There is a room for further research.
- Termination: ad hoc, by limiting the number of lemma applications for a given node. – There is a room for further research.
Outline

Introduction
The concept of metasystem transition
Supercompiler HOSC

Higher-level supercompilation
Naive example
Proving lemmas by supercompilation
Algorithm

Correctness
Operational theory of improvement
Detecting improvement lemmas by supercompilation

Implementation
Higher-level HOSC

Examples
Non-linear expression
Accumulating parameter revisited
Improving the asymptotics: from $O(n^2)$ to $O(n)$
Non-linear expression

Input

data Bool = True | False;
data Nat = Z | S Nat;

or (even m) (odd m) where

even = λx → case x of { Z → True; S x1 → odd x1;};
odd = λx → case x of { Z → False; S x1 → even x1;};

or = λx y → case x of { True → True; False → y;};
Non-linear expression

The whistle blows

or (even m) (odd m)

case (even m) of {True → True; False → (odd m);}

case (case m of {Z -> True; S x -> odd x;})
of {True -> True; False -> odd m;}

m = Z
m = S x

True

case (odd x) of {True → True; False → odd (S x);}

case (case x of {Z -> False; S n -> even n;})
of {True -> True; False -> odd (S x);}

x = Z
x = S n

True

case (even n) of
{True -> True; False -> odd (S (S n));}
Non-linear expression

Generalization

let x=m in case even m of {True -> True; False -> odd x;}

case (even m) of {True -> True; False -> odd x;}

case (case m of {Z -> True; S y -> odd y;}) of {True -> True; False -> odd x;}

True

case (odd y) of {True -> True; False -> odd x;}

y = Z

odd x

case (even z) of {True -> True; False -> odd x;}

y = S z

m = Z

m = S y
Non-linear expression

The result of “classic” supercompilation

*(letrec f = *(λv →
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          letrec g = *(λw →
            case w of {
              Z → False;
              S t → *(case t of {Z → True; S z → g z;});})
            in g m;
          S x → f x;
        });
    });
  in f m)
Non-linear expression

Higher-level supercompilation: the search for a lemma

A lemma of size 5

```haskell
case (even n) of {True → True; False → (odd (S (S n))));}
  \equiv or (even n) (odd n)
```

Not an improvement lemma

Going further: lemmas of size 6

```haskell
\begin{align*}
\textbf{case} & \ (\text{even } n) \ \textbf{of} \ \{ \text{True } \to \text{True}; \ \text{False } \to \text{odd } (S \ (S \ n)) \}; \\
\textbf{\triangleright} \ & \ \textbf{case} \ (\text{even } n) \ \textbf{of} \ \{ \text{True } \to \text{True}; \ \text{False } \to \text{odd } n \}; \\
\textbf{case} & \ (\text{even } n) \ \textbf{of} \ \{ \text{True } \to \text{True}; \ \text{False } \to \text{odd } (S \ (S \ n)) \}; \\
\textbf{\triangleright} \ & \ \textbf{case} \ (\text{odd } n) \ \textbf{of} \ \{ \text{True } \to \text{odd } n; \ \text{False } \to \text{True}; \}
\end{align*}
```

These are improvement lemmas. HLSC applies the first one.
Non-linear expression

Higher-level supercompilation: applying the lemma

or (even m) (odd m)

\[
\text{case (even m) of } \{ \text{True } \rightarrow \text{True}; \text{False } \rightarrow \text{(odd m)}; \}\]

\[
\text{case (case m of } \{ \text{Z } \rightarrow \text{True}; \text{S x } \rightarrow \text{odd x}; \}\) \\
of \{ \text{True } \rightarrow \text{True}; \text{False } \rightarrow \text{odd m}; \}\]

\[
m = \text{Z} \quad m = \text{S x}
\]

True

\[
\text{case (odd x) of } \{ \text{True } \rightarrow \text{True}; \text{False } \rightarrow \text{odd (S x)}; \}\]

\[
\text{case (case x of } \{ \text{Z } \rightarrow \text{False}; \text{S n } \rightarrow \text{even n}; \}\) \\
of \{ \text{True } \rightarrow \text{True}; \text{False } \rightarrow \text{odd (S x)}; \}\]

\[
x = \text{Z} \quad x = \text{S n}
\]

True

\[
\text{case (even n) of } \{ \text{True } \rightarrow \text{True}; \text{False } \rightarrow \text{odd n}; \}\]
Non-linear expression

The result of higher-level supercompilation

\[
\text{letrec } f = \lambda w \rightarrow \\
\text{ case } w \text{ of } \{ \\
\text{ Z } \rightarrow \text{ True}; \\
\text{ S } x \rightarrow \text{ case } x \text{ of } \{ \text{ Z } \rightarrow \text{ True}; \text{ S } z \rightarrow f z; \}; \\
\}
\text{ in } f \text{ m }
\]
Accumulating parameter revisited

Input

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

even (double x Z) where

   even = \x -> case x of {Z -> True; S x1 -> odd x1;};
   odd = \x -> case x of {Z -> False; S x1 -> even x1;};

   double = \x y -> case x of { Z -> y; S x1 -> double x1 (S (S y));};
```
Accumulating parameter revisited

The Whistle blows

even (double x Z)

case (double x Z) of \{Z \rightarrow True; \ S \ y \rightarrow odd \ y;\}\n
case case x of \{Z \rightarrow Z; \ S \ z \rightarrow double \ z \ (S \ (S \ Z));\} \ of \{Z \rightarrow True; \ S \ y \rightarrow odd \ y;\}\n
x = Z \quad x = S \ n

True

case (double n \ (S \ (S \ Z))) \ of \{Z \rightarrow True; \ S \ m \rightarrow odd \ m;\}
Accumulating parameter revisited

Improvement Lemma 1

The Whistle

case double x Z of {Z → True; S y → odd y};
   △.case double n (S (S Z)) of {Z → True; S m → odd m;}

The search for improvement lemmas

case double n (S (S Z)) of {Z → True; S m → odd m;}
   △.case double n (S Z) of {Z → True; S m → even m;}
case double n (S (S Z)) of {Z → True; S m → odd m;}
   △.case double n (S Z) of {Z → False; S m → even m;}

HLSC applies the first lemma.
Accumulating parameter revisited

Improvement Lemma 2

Further driving. The Whistle blows again:

\[
\text{case double } n \ (S \ Z) \ \text{of} \ \{Z \rightarrow \text{True}; \ S \ m \rightarrow \text{even} \ m;\} \\
\triangleleft_c \ \text{case double } p \ (S \ (S \ (S \ Z))) \ \text{of} \ \{Z \rightarrow \text{True}; \ S \ m \rightarrow \text{even} \ m;\}
\]

The search for improvement lemmas

\[
\text{case double } p \ (S \ (S \ (S \ Z))) \ \text{of} \ \{Z \rightarrow \text{True}; \ S \ m \rightarrow \text{even} \ m;\} \\
\triangleright \ \text{case double } p \ (S \ Z) \ \text{of} \ \{Z \rightarrow \text{True}; \ S \ m \rightarrow \text{even} \ m;\}
\]

\[
\text{case double } p \ (S \ (S \ (S \ Z))) \ \text{of} \ \{Z \rightarrow \text{True}; \ S \ m \rightarrow \text{even} \ m;\} \\
\triangleright \ \text{case double } p \ (S \ Z) \ \text{of} \ \{Z \rightarrow \text{False}; \ S \ m \rightarrow \text{even} \ m;\}
\]

HLSC appies the first lemma. Folding is performed.
Accumulating parameter revisited

The result of higher-level supercompilation

case x of {  
Z → True;  
S y1 →  
  letrec f = λt2 →  
    case t2 of {Z → True; S u2 → f u2;}  
    in f y1;  
}
**Improving the asymptotics: from $O(n^2)$ to $O(n)$**

*A naive parser*

```plaintext
doubleA = '' | 'A' doubleA 'A'
```

The complexity of this parser is $O(n^2)$, where $n$ is the length of an input.
Improving the asymptotics: from $O(n^2)$ to $O(n)$

**Input**

```haskell
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 \rightarrow p (eof return) i;
return = \x \rightarrow Some x;
doublea = or nil (join a (join doublea a));
or = \lambda p1 p2 next w \rightarrow case p1 next w of {
    Some w1 \rightarrow Some w1;
    None \rightarrow p2 next w;
};
nil = \lambda next w \rightarrow next w;
join = \lambda p1 p2 next w \rightarrow p1 (p2 next) w;
a = \lambda next w \rightarrow case w of {
    Cons s w1 \rightarrow case s of { A \rightarrow next w1; B \rightarrow None;};
    Nil \rightarrow None;};
b = \lambda next w \rightarrow ...
eof = \lambda next w \rightarrow case w of { Cons s w1 \rightarrow None; Nil \rightarrow next Nil;};
```
Improving the asymptotics: from $O(n^2)$ to $O(n)$

Applying a lemma

Driving. The Whistle blows:

case (case word of {Cons v32 v33 → None; Nil → return Nil;}) of {
  Some v34 → Some v34;
  None → join a (join doublea a) (eof return) word;
} ≤ₜₕₜₛₜₛₜₚₜₜ

case (case v97 of {Cons v149 v150 → None; Nil → return Nil;}) of {
  Some v151 → Some v151;
  None → (join a) (join doublea a) (a (eof return)) (Cons A v97);
}

The search for improvement lemmas

case (case v97 of {Cons v149 v150 → None; Nil → return Nil;}) of {
  Some v151 → Some v151;
  None → (join a) (join doublea a) (a (eof return)) (Cons A v97);
} ≥

case (case v97 of {Cons v149 v150 → None; Nil → return Nil;}) of {
  Some v151 → Some v151;
  None → (join a) (join doublea a) (eof return) v97;
Improving the asymptotics: from $O(n^2)$ to $O(n)$

The result of higher-level supercompilation

```plaintext
letrec
  f = λs14 →
  case s14 of { 
    Cons w1 w4 →
    case w1 of { 
      A → case w4 of { 
        Cons y7 y2 → case y7 of { A → f y2; B → None; }; 
        Nil → None; 
      }; 
      B → None; 
    }; 
    Nil → Some Nil; 
  }

in
  f word
```

The complexity of this parser is $O(n)$, where $n$ is the length of an input.
Improving the asymptotics: from $O(n^2)$ to $O(n)$

Transformation of BNF-grammars

**Input**

```plaintext
doubleA = '' | 'A' doubleA 'A'
```

**Output**

```plaintext
doubleA = '' | 'A' 'A' doubleA
```

It also means that we proved the equivalence of two BNF-grammars for free!
Related work

• R. M. Burstall and J. Darlington. A transformation system for developing recursive programs.

  The “eureka” step.

• Y. Futamura. Generalized Partial Computation.

  The use of theorem prover.

• P. Wadler. Concatenation vanishes.

  A single (improvement!) lemma: \( xs ++ [] = xs \).

• G. W. Hamilton. Distillation.

  Comparing and generalization of computation graphs.
Discussion

- The main idea of higher-level supercompilation is based on the principle of meta-system transition.
- Conceptual simplicity and modularity.
- Easy to adapt to other supercompilers.
- Many frameworks relies on existing lemmas. Higher-level supercompilation automatically discovers lemmas on demand.

Further work

- Make it fast and efficient (by using some ideas from theorem proving, like rippling and difference matching).
- Smarter application of lemmas.
- “Conditional” supercompilation: lemmas may be supplied as pre-conditions.