Towards Higher-Level Supercompilation

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

July 2 / Meta 2010

Outline

Outline

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.

V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In Partial Evaluation, volume 1110 of LNCS, pages 481–509. Springer, 1996.

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

- Ilya Klyuchnikov. Supercompiler HOSC 1.0: under the hood. *KIAM Preprint 63*. 2009.
- Ilya Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. *KIAM Preprint 21*. 2010.
- Ilya Klyuchnikov. Supercompiler HOSC: proof of correctness. *KIAM Preprint 31*. 2010.

Outline

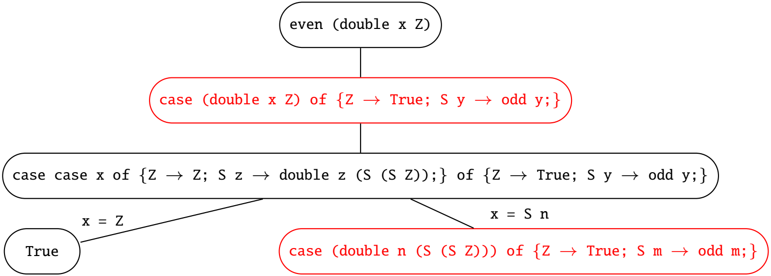
Accumulating parameter

Input

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

even (double x Z) where

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

double = λx y → case x of { Z → y; S x1 → double x1 (S (S y));};
```

Accumulating parameter

Driving and whistle



Accumulating parameter

The result of "classic" supercompilation

```
letrec
  f = λw2 λp2→
    case w2 of {
      Z →
        letrec g = λr2→
          case r2 of {
            S r → case r of {Z → False; S z2 → g z2;};
            Z → True;
          }
        in g p2;
      S z → f z (S (S p2));
    }
in f x Z
```

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

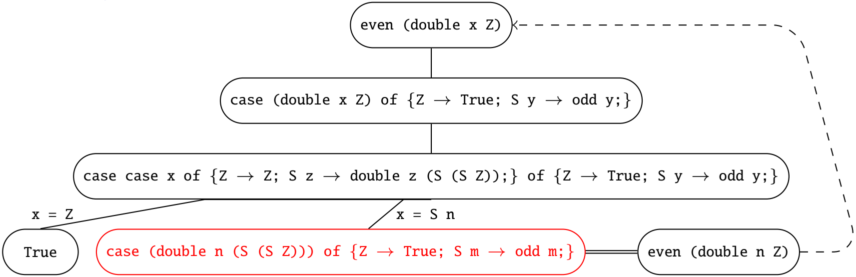
Accumulating parameter

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

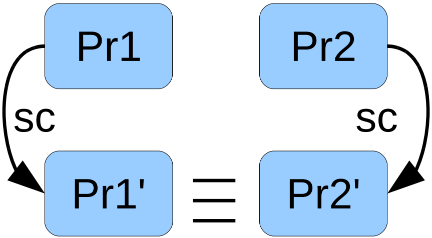
Applying an "enigmatic" lemma

The result of applying the lemma

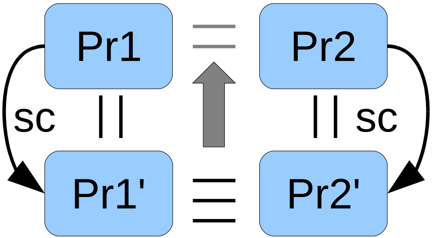
```
letrec f = λt →
  case t of { Z → True; S s → f s; }
in f x
```

(There are no `False` in the residual program!)

Proving lemmas by supercompilation

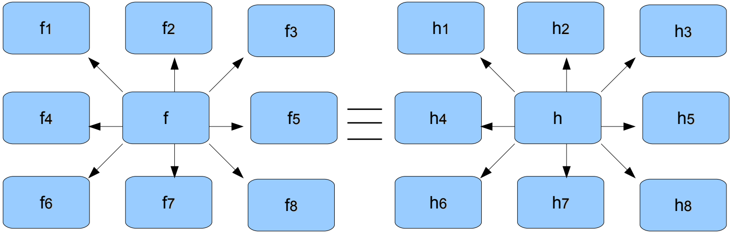


Proving lemmas by supercompilation



Proving lemmas by supercompilation

Multi-result supercompilation



Proving lemmas by supercompilation

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

- A.P. Lisitsa and M. Webster. Supercompilation for equivalence testing in metamorphic computer viruses detection. Meta 2008.

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

- I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. PSI 2009.
- I. Klyuchnikov. Inferring and proving properties of functional programs by means of supercompilation. PhD Thesis. 2010.

Higher-level supercompilation algorithm (sketch)

"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)
      and (not whistle(c, history))
        return c
  return null
```

Outline

Operational theory of improvement

David Sands, 1994

Operational approximation

An expression e operationally approximates e', $e \sqsubseteq_{\sim} e'$, 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', $e \cong e'$ if $e \sqsubseteq_{\sim} e'$ and $e' \sqsubseteq_{\sim} e$.

Improvement

An expression e is improved by e', $e \trianglerighteq_{\sim} e'$, 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 $e_1 \cong e_2$ and $e_1 \trianglerighteq\!\!\sim e_2$.

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

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

Lemma

```
e1 = or (even n) (odd n)
e2 = case (even n) of {True → True; False → odd (S (S n));}
e1 ≅ e2
```

But not an improvement lemma

n = Z: e1 $\mapsto^2$ True, e2 $\mapsto^1$ True

n = S Z: e1 $\mapsto^5$ True, e2 $\mapsto^6$ True

Detecting improvement lemmas by supercompilation

From theory to practice

Idea

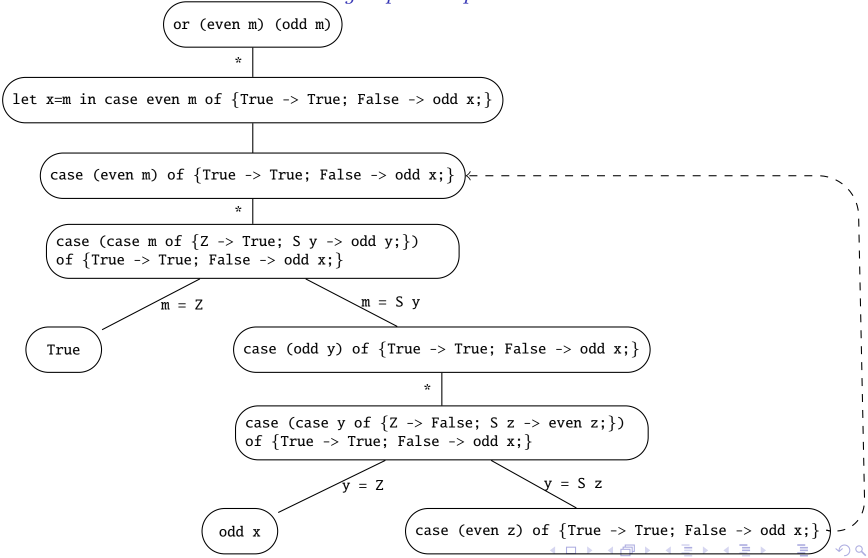
Let us propagate the information about evaluation cost of the original expression into the residual expression.

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



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

$$\frac{m \geq n \qquad \forall i : e_i \underset{\sim}{\triangleright}^* e_i'}{{}^m\phi(e_1, \ldots, e_k) \underset{\sim}{\triangleright}^* {}^n\phi(e_1', \ldots, e_k')}$$

From embedding of annotations to improvements

$$\frac{\mathcal{SC}[\![e_1]\!] \equiv \mathcal{SC}[\![e_2]\!] \qquad \mathcal{SC}[\![e_1]\!] \underset{\sim}{\triangleright}^* \mathcal{SC}[\![e_2]\!]}{e_1 \underset{\sim}{\triangleright} e_2}$$

Theorem

Let $e_1' = \mathcal{SC}[\![e_1]\!]$ and $e_2' = \mathcal{SC}[\![e_1]\!]$. If $e_1' \equiv e_2'$ and $e_1' \underset{\sim}{\triangleright}^* e_2'$, then $e_1 \underset{\sim}{\triangleright} e_2$ and it is correct to replace e_1 by e_2 during supercompilation.

Outline

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

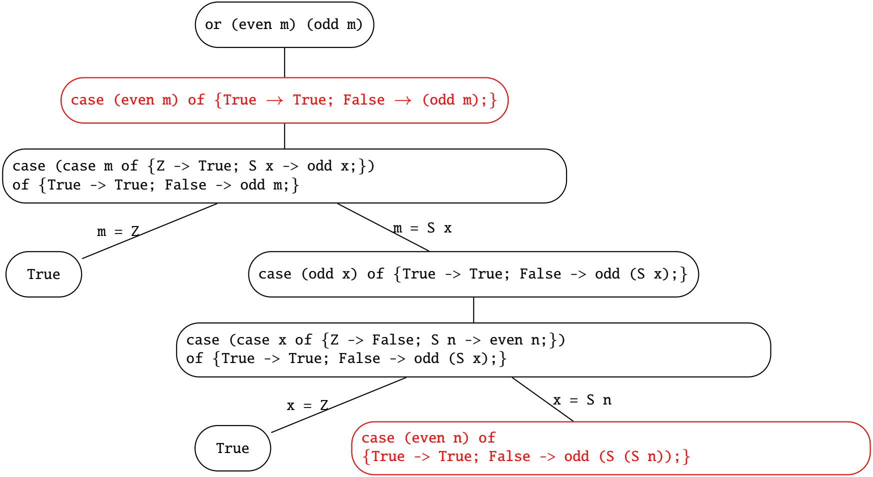
Non-linear expression

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;};
```

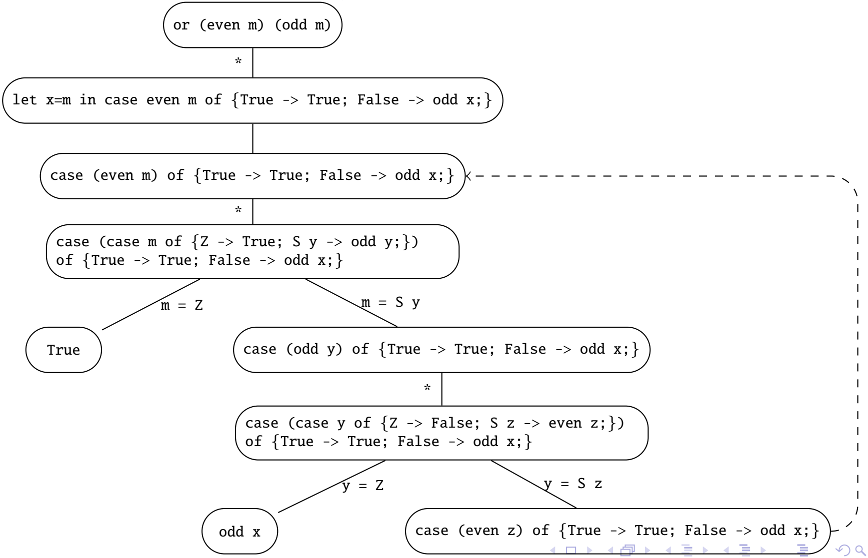
Non-linear expression

The whistle blows



Non-linear expression

Generalization



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

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

Not an improvement lemma

Going further: lemmas of size 6

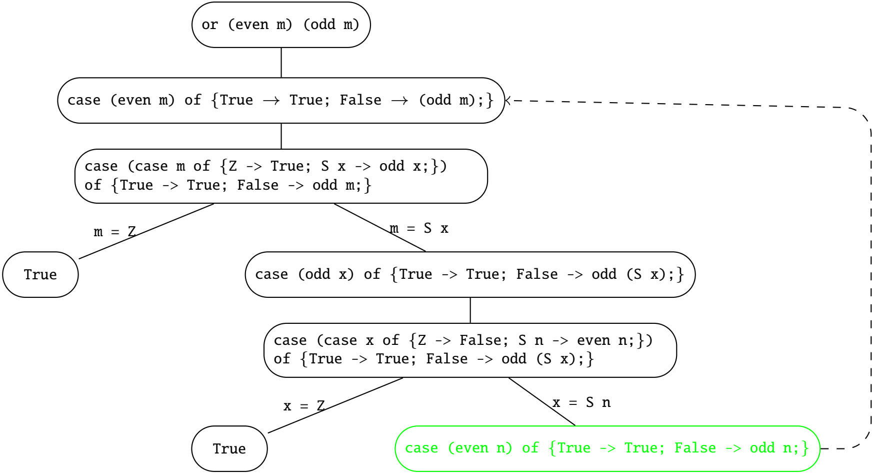
```
case (even n) of {True → True; False → odd (S (S n));}
   ⊵~ case (even n) of {True → True; False → odd n;}
```

```
case (even n) of {True → True; False → odd (S (S n));}
   ⊵~ case (odd n) of {True → odd n; False → True;}
```

These are improvement lemmas. HLSC applies the first one.

Non-linear expression

Higher-level supercompilation: applying the lemma



Non-linear expression

The result of higher-level supercompilation

```
letrec f = λw →
  case w of {
    Z → True;
    S x → case x of { Z → True; S z → f z;};
  }
in f m
```

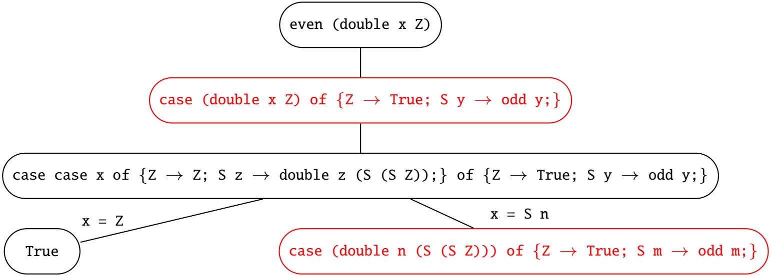
Accumulating parameter revisited

Input

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

even (double x Z) where

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

double = λx y → case x of { Z → y; S x1 → double x1 (S (S y));};
```

Accumulating parameter revisited

The Whistle blows



Accumulating parameter revisited

Improvement Lemma 1

The Whistle

```
case double x Z of {Z → True; S y → odd y);}
   ⊴c 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 appies the first lemma.

Accumulating parameter revisited

Improvement Lemma 2

Further driving. The Whistle blows again:

$\texttt{\textbf{case} double n (S Z) \textbf{of} \{Z} \rightarrow \texttt{True; S m} \rightarrow \texttt{even m;\}}$
$\quad \trianglelefteq_c \texttt{\textbf{case} double p (S (S (S Z))) \textbf{of} \{Z} \rightarrow \texttt{True; S m} \rightarrow \texttt{even m;\}}$

The search for improvement lemmas

$\texttt{\textbf{case} double p (S (S (S Z))) \textbf{of} \{Z} \rightarrow \texttt{True; S m} \rightarrow \texttt{even m;\}}$
$\qquad \underset{\sim}{\triangleright} \texttt{\textbf{case} double p (S Z) \textbf{of} \{Z} \rightarrow \texttt{True; S m} \rightarrow \texttt{even m;\}}$
$\texttt{\textbf{case} double p (S (S (S Z))) \textbf{of} \{Z} \rightarrow \texttt{True; S m} \rightarrow \texttt{even m;\}}$
$\qquad \underset{\sim}{\triangleright} \texttt{\textbf{case} double p (S Z) \textbf{of} \{Z} \rightarrow \texttt{False; S m} \rightarrow \texttt{even m;\}}$

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

```
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

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

match doublea word where

match = λp i → p (eof return) i;
return = λx → Some x;
doublea = or nil (join a (join doublea a));
or = λp1 p2 next w → case p1 next w of {
  Some w1 → Some w1;
  None → p2 next w;
};
nil = λnext w → next w;
join = λp1 p2 next w → p1 (p2 next) w;
a = λnext w → case w of {
  Cons s w1 → case s of { A → next w1; B → None;};
  Nil → None;};
b = λnext w → ...
eof = λnext w → case w of { Cons s w1 → None; Nil → 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;
} ⊴c
  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

```
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

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

Output

```
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.