Proving the Equivalence of Higher-Order Terms by Means of Supercompilation

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Novosibirsk, June 17 2008

Outline

Introduction

A Brief Survey on Supercompilers
HOSC - an Experimental Supercompiler

Proving properties of programs

HOSC DEMO: Parameterized testing

Proving equality and equivalence

HOSC DEMO: Church numbers
HOSC DEMO: Map composition
The Idea of Proving term equivalence

Applications

Library of Lemmas
Towards a Higher-Level Supercompiler

Summary

A Brief Survey on Supercompilers

SPEC

SCP[1,2,3] - Turchin et al.
SCP4 - A. Nemytykh
Supero - N. Mitchell
SC for Timber - P. Jonnson
JScp - A. Klimov
Poitin - G. Hamilton

A Brief Survey on Supercompilers

SPEC	*Primary goal*
SCP[1,2,3] - Turchin et al.	OPT
SCP4 - A. Nemytykh	SELF-APP
Supero - N. Mitchell	OPT
SC for Timber - P. Jonnson	OPT
JScp - A. Klimov	OPT
Poitin - G. Hamilton	OPT

A Brief Survey on Supercompilers

SPEC	*Primary goal*	*Preserves semantics*
SCP[1,2,3] - Turchin et al.	OPT	NO
SCP4 - A. Nemytykh	SELF-APP	NO
Supero - N. Mitchell	OPT	YES
SC for Timber - P. Jonnson	OPT	YES
JScp - A. Klimov	OPT	YES
Poitin - G. Hamilton	OPT	YES

A Brief Survey on Supercompilers

SPEC	*Primary goal*	*Preserves semantics*	*Easy to try*
SCP[1,2,3] - Turchin et al.	OPT	NO	-
SCP4 - A. Nemytykh	SELF-APP	NO	If you know Refal
Supero - N. Mitchell	OPT	YES	If you use YHC
SC for Timber - P. Jonnson	OPT	YES	-
JScp - A. Klimov	OPT	YES	If you are Klimov
Poitin - G. Hamilton	OPT	YES	-

A Brief Survey on Supercompilers

SPEC	*Primary goal*	*Preserves semantics*	*Easy to try*
SCP[1,2,3] - Turchin et al.	OPT	NO	-
SCP4 - A. Nemytykh	SELF-APP	NO	If you know Refal
Supero - N. Mitchell	OPT	YES	If you use YHC
SC for Timber - P. Jonnson	OPT	YES	-
JScp - A. Klimov	OPT	YES	If you are Klimov
Poitin - G. Hamilton	OPT	YES	-
HOSC		YES	If you have a browser

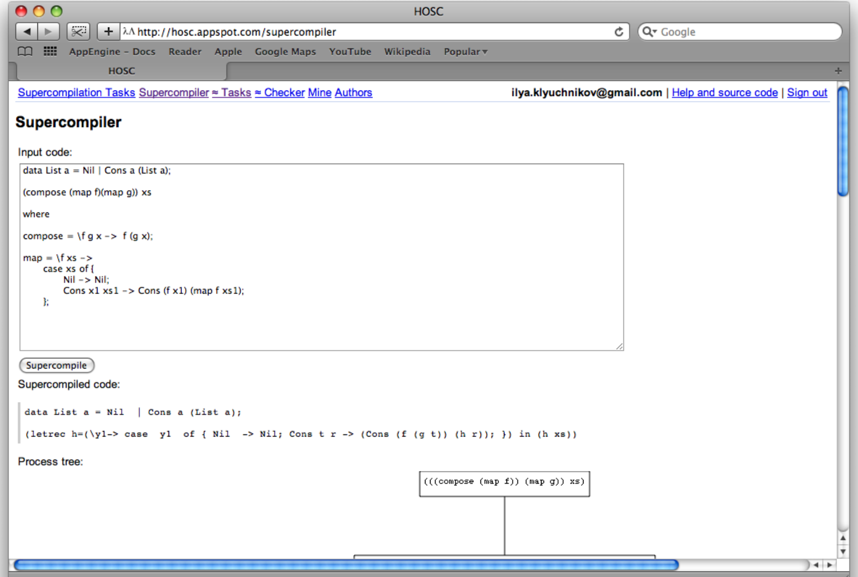
A Brief Survey on Supercompilers

SPEC	*Primary goal*	*Preserves semantics*	*Easy to try*
SCP[1,2,3] - Turchin et al.	OPT	NO	-
SCP4 - A. Nemytykh	SELF-APP	NO	If you know Refal
Supero - N. Mitchell	OPT	YES	If you use YHC
SC for Timber - P. Jonnson	OPT	YES	-
JScp - A. Klimov	OPT	YES	If you are Klimov
Poitin - G. Hamilton	OPT	YES	-
HOSC	ANALYSIS	YES	If you have a browser

HOSC - an Experimental Supercompiler

- Deals with a simple higher-order functional language with lazy semantisc (a subset of Haskell)
- Preserves semantics
- Open Source
- Runs in a browser. Try it at `http://hosc.appspot.com`

HOSC DEMO



Supercompiler

Input code:

```
data List a = Nil | Cons a (List a);

(compose (map f)(map g)) xs

where

compose = \f g x -> f (g x);

map = \f xs ->
    case xs of {
        Nil -> Nil;
        Cons x1 xs1 -> Cons (f x1) (map f xs1);
    };
```

Supercompiled code:

```
data List a = Nil  | Cons a (List a);

(letrec h=(\y1-> case  y1  of { Nil  -> Nil; Cons t r -> (Cons (f (g t)) (h r)); }) in (h xs))
```

Process tree:

Parameterized testing: a source program

```
data List a = Nil | Cons a (List a);
data Enum = A | B;
data Boolean = True | False;
contains x (app xs (app (Cons x Nil) zs)) where
app = \xs ys →
    case xs of {
      Nil → ys;
      Cons z zs → Cons z (app zs ys);};
contains = \x xs →
  case xs of {
    Nil → False;
    Cons x1 xs1 → or (eq x1 x) (contains x xs1);};
eq = \x y → case x of {
  A → case y of {A → True; B → False;};
  B → case y of {A → False; B → True;};};
or = \x y → case x of {True → True;False → y;};
```

Parameterized testing: the residual program

```
data List a = Nil  | Cons a (List a);
data Enum   = A  | B ;
data Boolean  = True  | False ;
letrec f=\w2 p2→
  case  p2  of {
    Nil  → case  w2  of { A  → True; B  → True; };
    Cons w p →
      case  w  of {
        A → case  w2  of { A → True; B  → f B p; };
        B → case  w2  of { A → f A p; B → True; };
      };
  }
in
  f x xs
```

Church numbers

`0 = \f x → x`

`1 = \f x → f x`

`2 = \f x → f (f x)`

`3 = \f x → f (f (f x))`

`...`

`n = \f x →` $\mathtt{f}^{n}\mathtt{x}$

`...`

$\mathtt{f}^{m+n}\mathtt{x} = \mathtt{f}^{m}(\mathtt{f}^{n}\mathtt{x})$

`churchAdd = \m n → (\f x → m f (n f x));`

Church numbers

```
data Nat = Z | S Nat;

unchurch(churchAdd (church x) (church y)) = add  x y
where
church = \n → case n of {
  Z    → \f x → x;
  S n1 → \f x → f (church n1 f x);
};
unchurch = \n → n (\x → S x) Z;
churchAdd = \m n → (\f x → m f (n f x));
add = \x y → case x of {
  Z → y;
  S x1 → S (add x1 y);
};
```

Church numbers: a source program

```
data Nat = Z | S Nat;
data Boolean = False | True;
eq (add x y) (unchurch(churchAdd (church x) (church y)))
eq = \x y → case x of {
  Z    → case y of {Z → True;   S y1 → False;    } ;
  S x1 → case y of {Z → False; S y1 → eq x1 y1;} ;
};
church = \n → case n of {
  Z    → \f x → x;
  S n1 → \f x → f (church n1 f x);
};
unchurch = \n → n (\x → S x) Z;
churchAdd = \m n → (\f x → m f (n f x));
add = \x y → case x of {
  Z → y;
  S x1 → S (add x1 y);
};
```

Church numbers: the residual program

```
data Nat   = Z   | S Nat;
data Boolean  = False  | True;
case x of {
  Z  → case y  of {Z  → True; S w4 →
    letrec f=\a → case a of {Z → True; S x4 → f x4;}
    in f w4;};
  S r6 → letrec g=\u11→
    case  u11  of {
      Z → case y of {
        Z → True;
        S x9 → letrec h=\v11→
        case v11 of { Z → True; S b → h b;} in h x9;
      };
      S y7 → (g y7);}
    in g r6;}
```

Map composition

```
data List a = Nil | Cons a (List a);

map (compose f g) xs = (compose (map f )(map g)) xs
where

map = \f1 ys →
    case ys of {
        Nil → Nil;
        Cons y1 ys1 → Cons (f1 y1) (map f1 ys1);
    };

compose = \f1 f2 x → f1 (f2 x);
```

Task

Conjecture

```
map (compose f) xs = (compose (map f g)(map g)) xs
```

Restrictions

- No equality out of the box.
- List xs may be infinite (or bottom).
- Functions f and g may be non-terminating.

map (compose f g) xs: a source program

```
data List a = Nil | Cons a (List a);

map (compose f g) xs
where

map = \f1 ys →
    case ys of {
        Nil → Nil;
        Cons y1 ys1 → Cons (f1 y1) (map f1 ys1);
    };

compose = \f1 f2 x → f1 (f2 x);
```

map (compose f g) xs: the residual program

```
data List a = Nil | Cons a (List a)
letrec
  h = \ys.
    case ys of
      Nil → Nil
      Cons y1 ys1 → Cons (f (g y1)) (h ys1)
in
  h xs
```

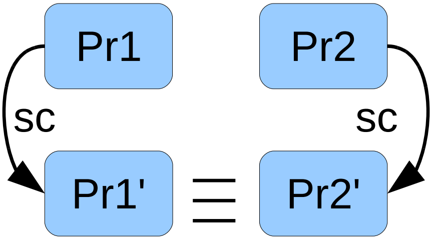
(compose (map f)(map g)) xs: a source program

```
data List a = Nil | Cons a (List a)

(compose (map f)(map g)) xs
where

map = \f1 ys →
     case ys of {
          Nil → Nil;
          Cons y1 ys1 → Cons (f1 y1) (map f1 ys1);
     };

compose = \f1 f2 x → f1 (f2 x);
```

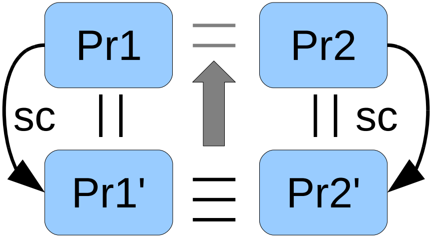
map f (map g xs) xs: the residual program

```
data List a = Nil | Cons a (List a)
letrec
  h = \ys.
    case ys of
      Nil → Nil
      Cons y1 ys1 → Cons (f (g y1)) (h ys1)
in
  h xs
```

The Idea



The Idea



Normalization by supercompilation

More formally

$$\frac{sc(A) = A' \qquad sc(B) = B' \qquad A' \equiv B'}{A = B}$$

$=$ means equivalent, $\equiv$ means syntactically isomorphic

Power of strict equivalence

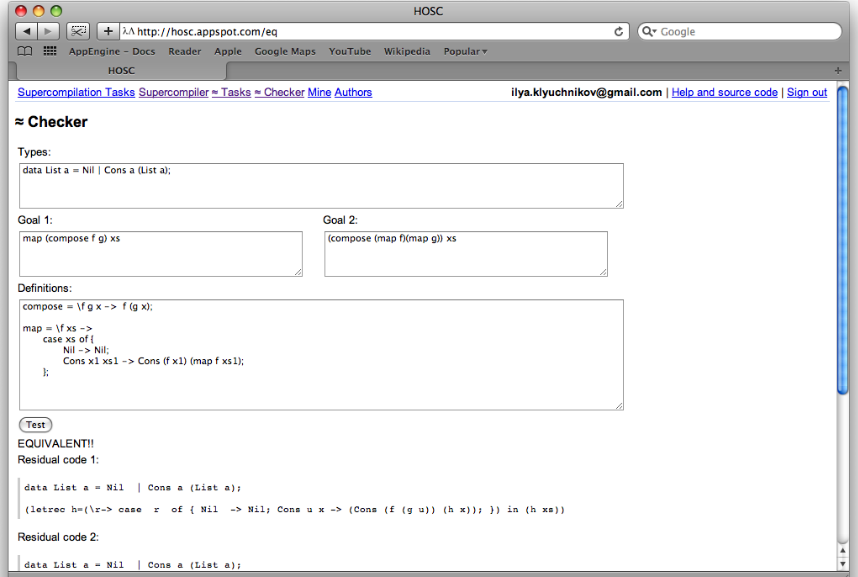
We can use transitivity when reasoning:

$$\frac{A = C \qquad B = C}{A = B}$$

Non-strict equivalence:

$$\frac{A \rightsquigarrow C \qquad B \rightsquigarrow C}{A \mathbin{?} B}$$

Automatic Checker



≈ Checker

Types:

```
data List a = Nil | Cons a (List a);
```

Goal 1:

```
map (compose f g) xs
```

Goal 2:

```
(compose (map f)(map g)) xs
```

Definitions:

```
compose = \f g x -> f (g x);

map = \f xs ->
    case xs of {
        Nil -> Nil;
        Cons x1 xs1 -> Cons (f x1) (map f xs1);
    };
```

EQUIVALENT!!

Residual code 1:

```
data List a = Nil  | Cons a (List a);

(letrec h=(\r-> case  r  of { Nil  -> Nil; Cons u x -> (Cons (f (g u)) (h x)); }) in (h xs))
```

Residual code 2:

```
data List a = Nil  | Cons a (List a);
```

Normalization-based approach to proving term equivalence

- Works for polymorphic data types
- Works for non-terminating functions
- Works for infinite data structures

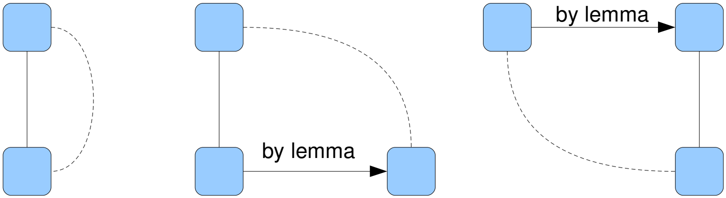
Library of Lemmas

```
compose (map f) unit = compose unit  f
compose (map  f) join = compose join (map (map  f))
append (map  f  xs) (map  f  ys) = map  f (append  xs  ys)
append (append  xs ys)  zs = append  xs (append  ys  zs)
filter  p (map  f xs) = map  f (filter (compose  p f)  xs)
iterate  f ( f x) = map  f (iterate  f x)
map (compose  f g)  xs = (compose (map  f)(map  g))  xs
rep (append  xs ys)  zs = (compose ( rep xs) (rep  ys))  zs
(compose abs rep)  xs = idList  xs
map (fp (P  f g)) (zip (P  x y)) = zip (fp (P (map  f) (map  g)) (P  x y))
append  r (Cons  p ps) =
  case (append  r (Cons  p Nil)) of
    Nil  → ps
    Cons v vs → Cons v (append vs  ps)
```

Library of Lemmas

```
data List a = Nil | Cons (List a);
data Boolean = True | False;
data Pair a b = P a b;

compose = \f g x → f (g x);
unit = \x → Cons x Nil;
rep = \xs → append xs;
abs = \f → f Nil;
iterate = \f x → Cons x (iterate f (f x));
fp = \p1 p2 → case p1 of {P a1 a2 →
    case p2 of {P b1 b2 → P (a1 b1) (a2 b2);};};
map = \f xs → case xs of {Nil → Nil;
    Cons x1 xs1 → Cons (f x1) (map f xs1);}
join = \xs → case xs of  {Nil → Nil;
      Cons x1 xs1 → append x1 (join xs1);};
append = \xs ys → case xs of {Nil → ys;
    Cons x1 xs1 → Cons x1 (append xs1 ys);};
idList = \xs → case xs of {Nil → Nil;
      Cons x1 xs1 → Cons x1 (idList xs1);};
filter = \p xs → case xs of {Nil → Nil;
      Cons x xs1 → case (p x) of {
          True → Cons x (filter p xs1);
          False → filter p xs1;};};
zip = \p → case p of {P xs ys → case xs of {
      Nil → Nil;
      Cons x1 xs1 → case ys of{
            Nil → Nil;
            Cons y1 ys1 → Cons (P x1 y1) (zip (P xs1 ys1));};};}
```

Improved configuration analysis



- Summary
    - The experimental opern-sourced supercompiler HOSC: easy to run.
    - The simple idea for proving term equivalence by means of supercompilation was described.
    - The fully automatic equivalence checker was implemented.
- Future Work
    - Automatic generation of lemma library for a given program.
    - Encorporate lemmas into HOSC to make it more powerful.
- Announcement
    - "SPSC: a Simple Supercompiler in Scala" - at PU'09