Inferring and proving properties of functional programs by means of supercompilation

Ilya Klyuchnikov
Keldysh Institute of Applied Mathematics
http://pat.keldysh.ru/~ilya/

Meta 2010

Outline

1. Program analysis by transformation
    - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
    - Checking correctness of monads
5. Higher-level supercompilation
    - Detecting improvement lemmas
    - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

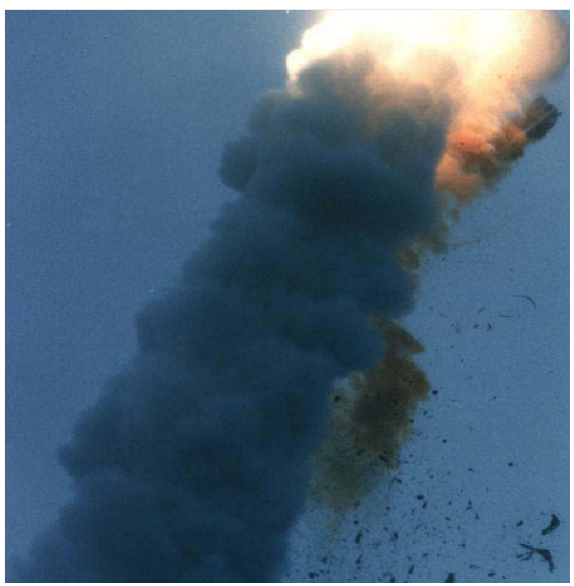
Outline

1. Program analysis by transformation
   - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
   - Checking correctness of monads
5. Higher-level supercompilation
   - Detecting improvement lemmas
   - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

Problem of correctness

Errors are expensive:

1. 1994. The Pentium FDIV bug (a bug in the Intel P5 Pentium floating point unit)
2. 1996. Ariane 5 Flight 501. The rocket was destroyed by its automated self-destruct system
3. 1998. Huygens probe. Channel A data lost.



Checking correctness of programs

- Testing (unit testing): p(f(x1)),...,p(f(xn)).
- Formal verification: ∀ X P(f(X))
  - Model checking
- **Checking by transformation:**
  p(f(X)) ==> p'(X)
  (not well-studied)
  From testing: p is code in the same language as program being tested
  From verification: all cases rather than several.

Many errors are detectable in models rather than in real programs.
Example - Mars Pathfinder 1997 (incorrect synchronization of processes).

Analysis by transformation

`p(X, f(X))` - a statement about program `f`
(p is a *program* on *the same* language!)
The statement is transformed:
`p(X, f(X))=> p'(X)`
The transformed statement is analyzed mechanically.

`p'(X)` has the form `True`
Verified. p' doesn't depend on X, so ∀X: `p'(X)`.

`p'(X)` has the form `... False ...`
Possibly, an error is found: ∃X: `p'(X)`
(Further analysis is required)
An application: Xu, Peyton Jones and Claessen. Static contract checking for Haskell. IN *Proceedings of the 36th annual symposium on Principles of programming languages*. 2009. (Contracts + symbolic execution at compile time)

Verification and supercompilation

Supercompilation is a program transformation technique capable of simplifying composition of function.

Successful application - Nemytykh 2005, 2007:

Verification of models of cache-coherent protocols coded in REFAL (Андрей Немытых).

```
test(loop(actions)) => loop'(actions)
```

`loop'` is analized

Errors were found in two published protocols.

Problem definition

Investigate facilities of program analysis by supercompilation in the higher-order call-by-name setting.

Haskell language (call-by-name) is chosen.

- Functional
  - Algebraic specifications (contacts)
- Lazy evaluation
  - Infinite data structures
  - Process modeling
- Higher-order functions
  - Specs may contain universally quantified functions (second-order logic)
  - Predicates as arguments of other predicates (higher-order logic)
- Statically typed
  - Reliable specifications

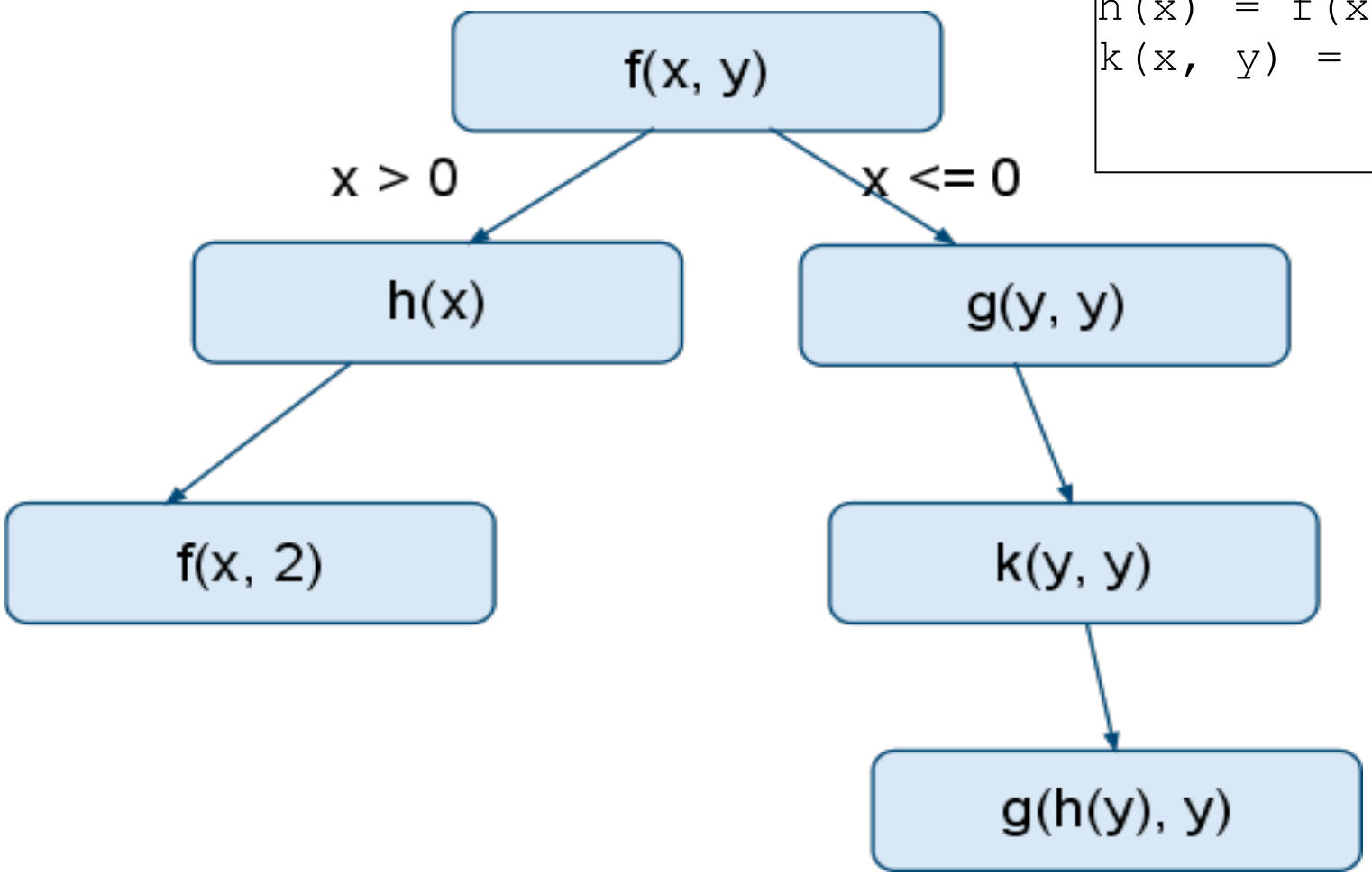
Outline

1. Program analysis by transformation
   - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
   - Checking correctness of monads
5. Higher-level supercompilation
   - Detecting improvement lemmas
   - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

What is supercompilation

Construction of tree of calculations
(tree is *infinite* in general)

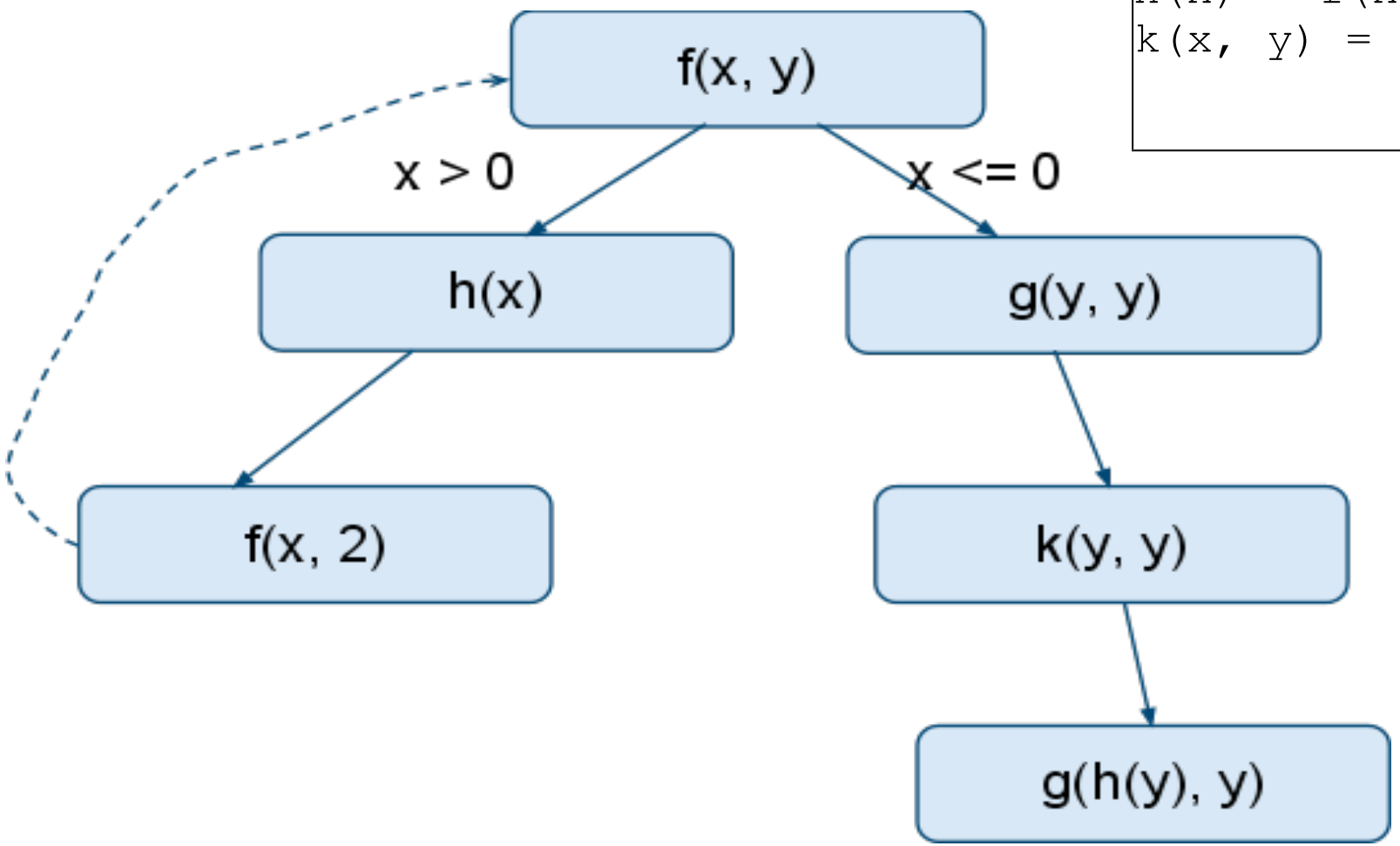
```
f(x, y)= if x > 0
         then h(x)
         else g(y, y)
g(x, y) = k(x, y)
h(x) = f(x, 2)
k(x, y) = g(h(x), y)
```



What is supercompilation

Folding of tree into a *finite* graph
(reduction of configurations to already processed ones)

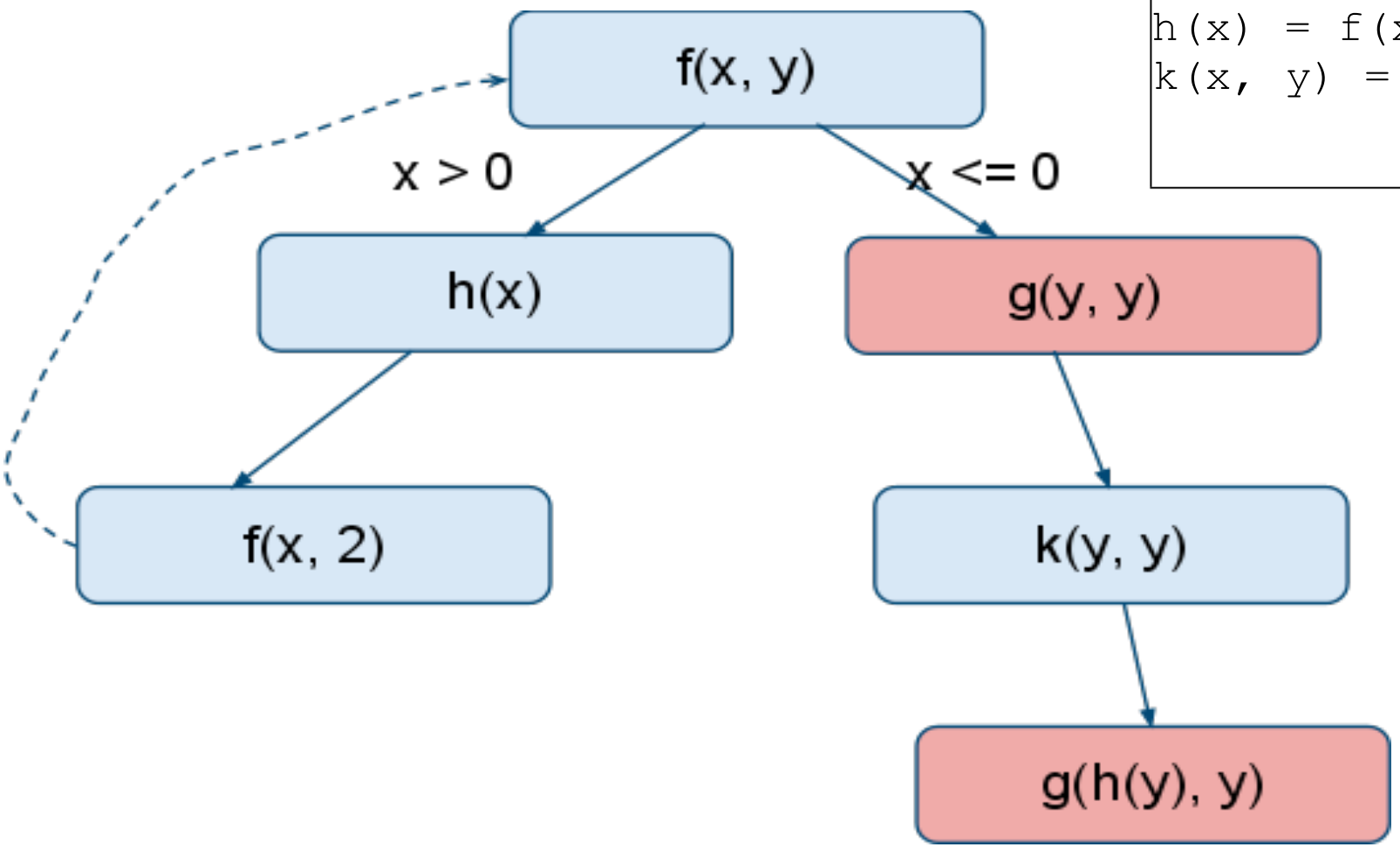
```
f(x, y)= if x > 0
         then h(x)
         else g(y, y)
g(x, y) = k(x, y)
h(x) = f(x, 2)
k(x, y) = g(h(x), y)
```



What is supercompilation

Detection of infinite branches

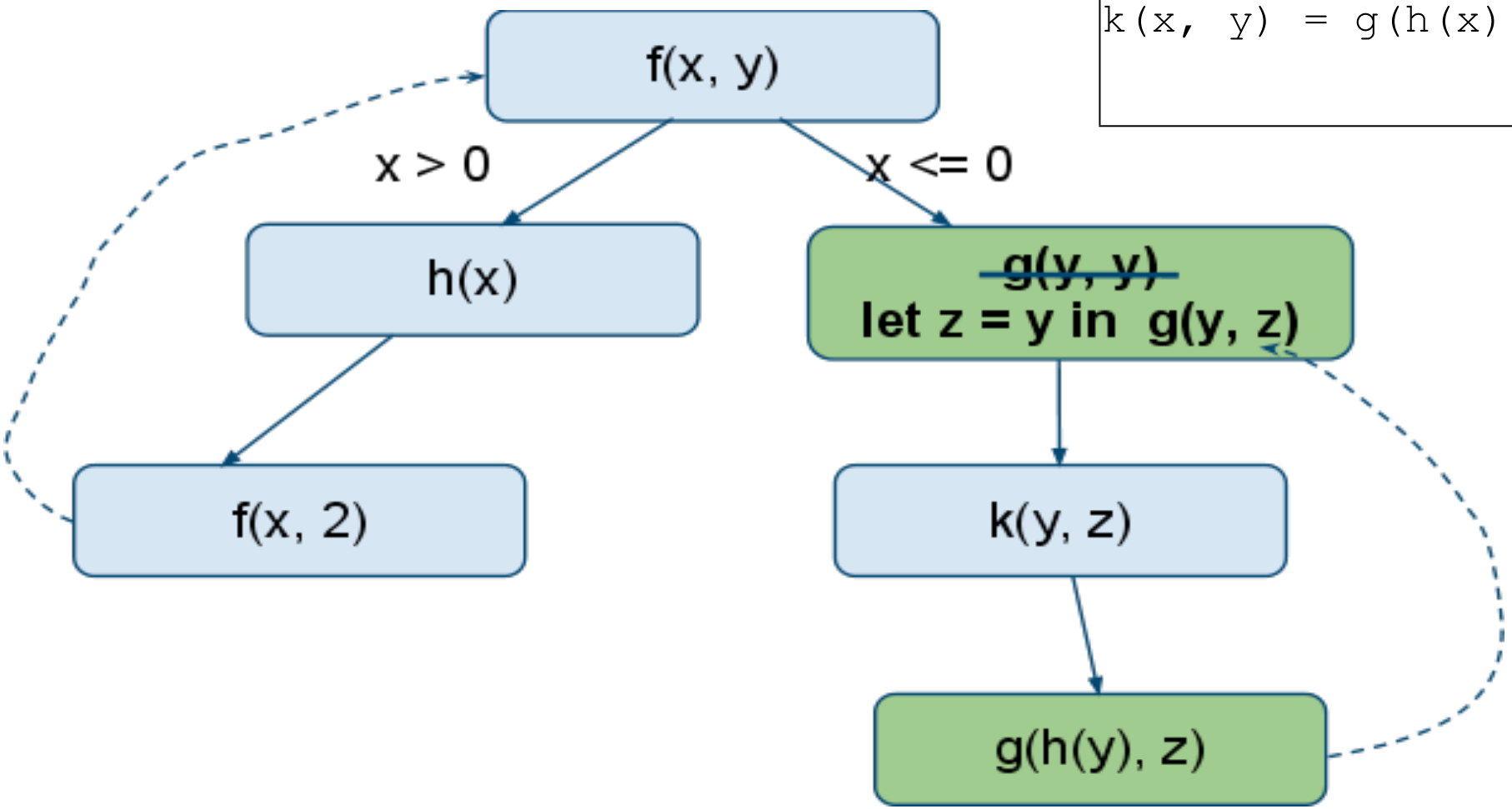
```
f(x, y)= if x > 0
         then h(x)
         else g(y, y)
g(x, y) = k(x, y)
h(x) = f(x, 2)
k(x, y) = g(h(x), y)
```



What is supercompilation

Generalization
(in order to create opportunities for folding)

```
f(x, y)= if x > 0
         then h(x)
         else g(y, y)
g(x, y) = k(x, y)
h(x) = f(x, 2)
k(x, y) = g(h(x), y)
```



What is supercompilation

Generalization criteria (whistle): homeomorphic embedding of configurations

g(y , y) ◁ g(h(y) , y)

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

Theorem (Kruskal, Higman) 1950-s. For **any infinite** sequence of expressions
e1, e2, … en …
there exist **i < j** such, that **ei ◁ ej**

Outline

1. Program analysis by transformation
   - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
   - Checking correctness of monads
5. Higher-level supercompilation
   - Detecting improvement lemmas
   - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

Requirements

Logical properties

- Strict preserving of equivalence
- Higher-order fuctions (higher-order logic)
- Infinite data structures (modeling of infinite processes)

"Quality guarantee"

- Proof of correctness (can we trust, whether equivalence is really preserved?)
- Proof of termination (nice to have)
- Source code

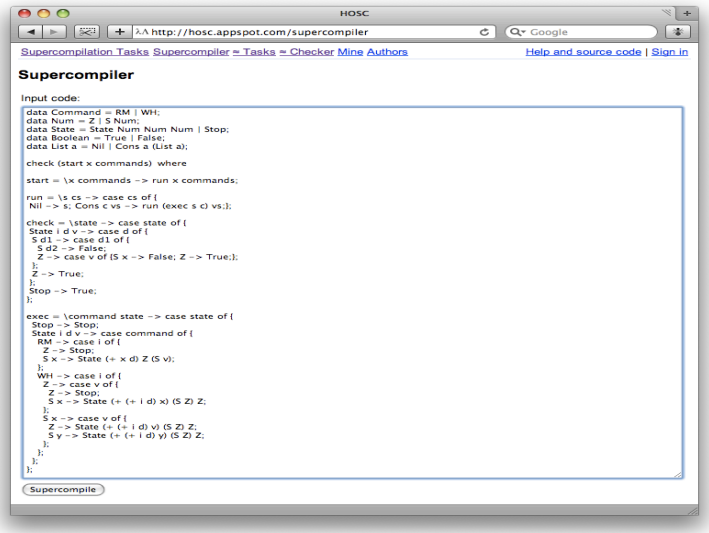
Status of existing supercompilers

	Equivalent transformation	Higher-order functions	Infinite data structures	Proof of correctness	Proof of termination	Source code
SCP 4 (Turchin, Nemytykh) 1980 - 2000	-	-	-	-	-	+
Simple supercompiler (Sørensen) - 1990-s	+	-	+	+	+	-
Supercompiler for TSG (Abramov) - 1990-e	+	-	-	-	-	+
Jscp (Klimov et al.)	+	-	-	-	-	-
Supero (Mitchell) 2007	+	+	+	-	- (!)	+
Supercompiler for Timber (Jonsson) 2008	+	+	-	+	+	-

Supercompiler HOSC

- Correctness is proved
- Open source
- Web-interface

http://code.google.com/p/hosc/
http://hosc.appspot.com



	Equivalent transformations	Higher-order functions	Infinite data structures	Proof of correctness	Proof of termination	Source code
Supercompiler HOSC (Klyuchnikov) 2008	+	+	+	+	+	+

Klyuchnikov I.G. Supercompiler HOSC 1.0: under the hood KIAM Preprint № 63, Moscow, 2009.

Encountered problems

Higher-order functions:

- Bound variables
- Program may loop without explicit recursion
- What is equivalence?

Goal is analysis rather than optimization

- Sometimes it is useful to "degrade" the program (in terms of performance)

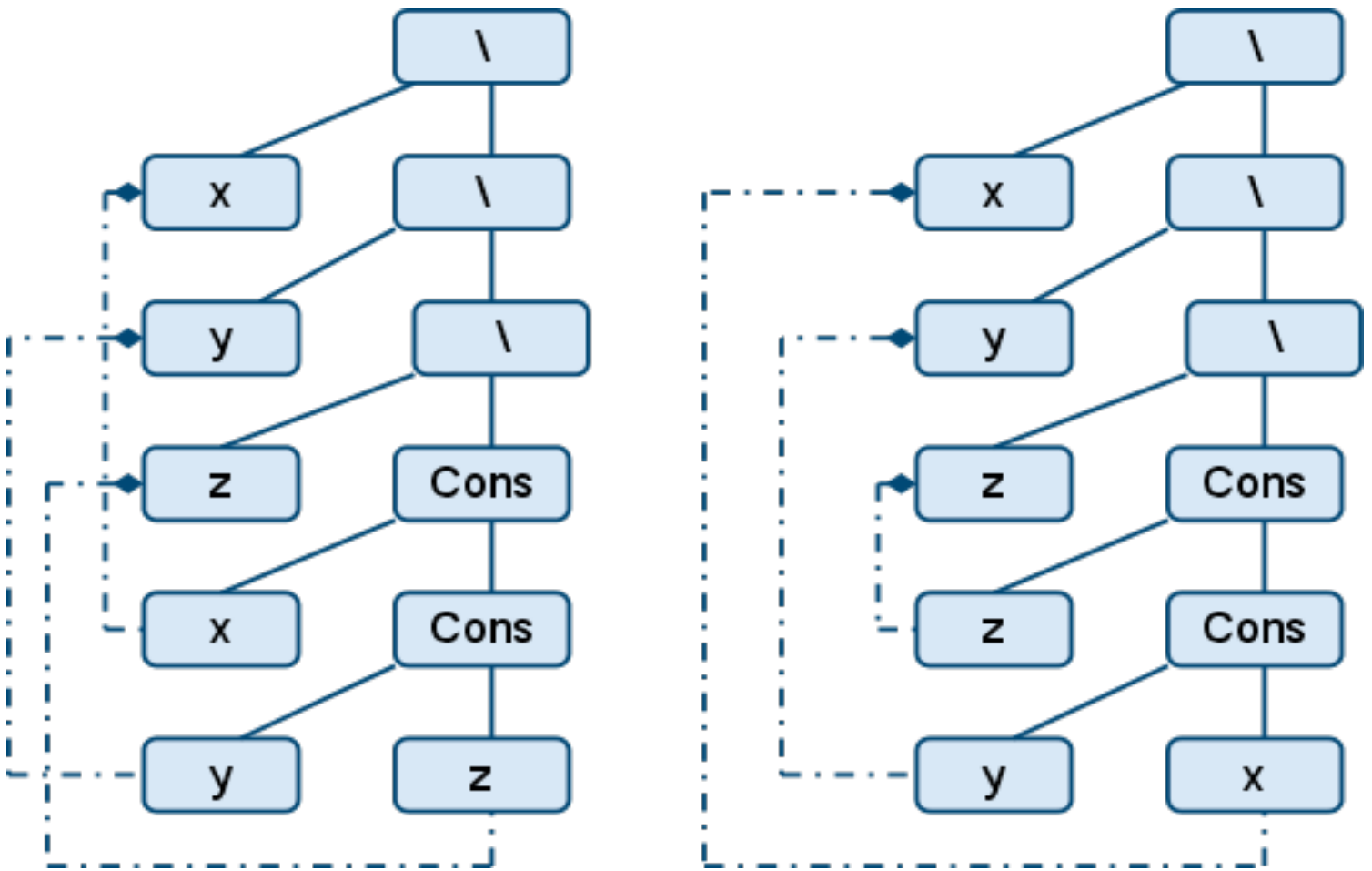
What had to be done?

- Extended homeomorphic embedding taking into account bound variables
- Extended generalization algorithm
- Preventing infinite reduction
- Decreasing arity (lambda-dropping)
- Folding of any configurations is allowed

Extended homeomorphic embedding

Distinguishing bound variables

```
\x y z -> Cons x (Cons y z) \x y z -> Cons z (Cons y x)
```



Extended homeomorphic embedding

Classical embedding

Variables

$$v_1 \trianglelefteq v_2$$

Diving

$$\frac{\exists i : e \trianglelefteq e'_i}{e \trianglelefteq \phi(e'_1, \ldots, e'_k)}$$

Coupling

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

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

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

Diving

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

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

Coupling

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

Extended embedding is well-quasi-order

Theorem (Kruskal, Higman). For any infinite sequence of expressions e1, e2, ... en, ..
there are i<j, such that
ei ◁ ej

Extended whistle doesn't blow for ANY sequence!

Theorem (Klyuchnikov). For any infinite sequence of expressions e1, e2, ... en, .., located at a branch of partial process tree t, there are i<j, such that
ei ◁* ej

Proof of terminations

Encountered problems?

- Infinite set of bound variables
- Unbounded arity of application
- Termination depends on typing!
- We need to fold not only con<f> configurations

Klyuchnikov I.G. Supercompiler HOSC 1.1: proof of termination. KIAM Preprint № 21, Moscow, 2010.

Proof of termination

Counter-example was found:

```
data D = F (D -> D);

fix h where

fix = \f -> apply
     (F (\x -> f (apply x x)))
         (F (\x -> f (apply x x)));

apply = \x -> case x of { F f -> f; };
```

If folding of configurations in the form con<f>
only is allowed then supercompilation will terminate because of calls to **`apply`**

Proof of termination

Let us inline **`apply`** into a definition of **`fix`**

```
data D = F (D -> D);

fix h where

fix = \f -> (\y -> case y of { F g -> g; })
      (F (\x -> f ((\y -> case y of { F g -> g; }) x x)))
      (F (\x -> f ((\y -> case y of { F g -> g; }) x x)));
```

No termination anymore!

Supercompiler HOSC 1.0 doesn't terminate

Supercompilers Supero и Supero 2010 do not terminate as well!

http://community.haskell.org/~ndm/temp/

Discussion in Google Groups

Proof of termination

Let us inline **`apply`** into a definition of **`fix`**

```
data D = F (D -> D);

fix h where

fix = \f -> (\y -> case y of { F g -> g; })
      (F (\x -> f ((\y -> case y of { F g -> g; }) x x)))
      (F (\x -> f ((\y -> case y of { F g -> g; }) x x)));
```

HOSC 1.1 supercompiles it into:

```
letrec g = h g in g
```

Correctness

What is equivalence?

In the first-order setting equivalence of programs reduces to the equality of results (data)
f = g, if
$\forall x$: f(x) = g(x).

In the higher-order setting f(x) may be a function! - How to compare functions?

Correctness

Operational theory of improvement (Sands, 1990)

Context is an expression C with a hole [ ] in the place of a subexpression. C[e] - expression, where hole is replaced by e.

Equivalence - expressions e1 and e2 are equivalent, if for any context C evaluations of both C[e1] and C[e2] either converge or diverge.

An expression e2 is an improvement of an expression e1, if for any context C, if computation of C[e1] terminates using n function calls, then computation of C[e2] terminates using no more than n function calls.

Correctness

Operational theory of improvement (Sands, 1990)

Operational theory of improvement is a "standard" toll for proving correctness of program transformations:

- Higher-Order Deforestation (Сэндс, 1995)
- Call-by-value Higher-Order Supercompilation (Jonsson, 2008)
- Flattening is an Improvement (Riely, Prins, 2000)
- ...

The idea is to show that a transfomed program is an improvement of an original program (and additional conditions hold).

Proof of correctness

Problems encountered:

1. The result of transformation performed by HOSC is not neccessary an improvement (may be good for analysis).
   - In order to prove correctness, we use supercombinators with maximal free expressions abstracted (Peyton Jones)
2. Decreasing arity of functions
   - We show correctness of transformation without decreasing arity and use correctness of lambda-dropping.
3. Typing: inferred type of transformed expression may be more general (No one pointed it out yet)
   - We use explicit type constraints in a residual program.

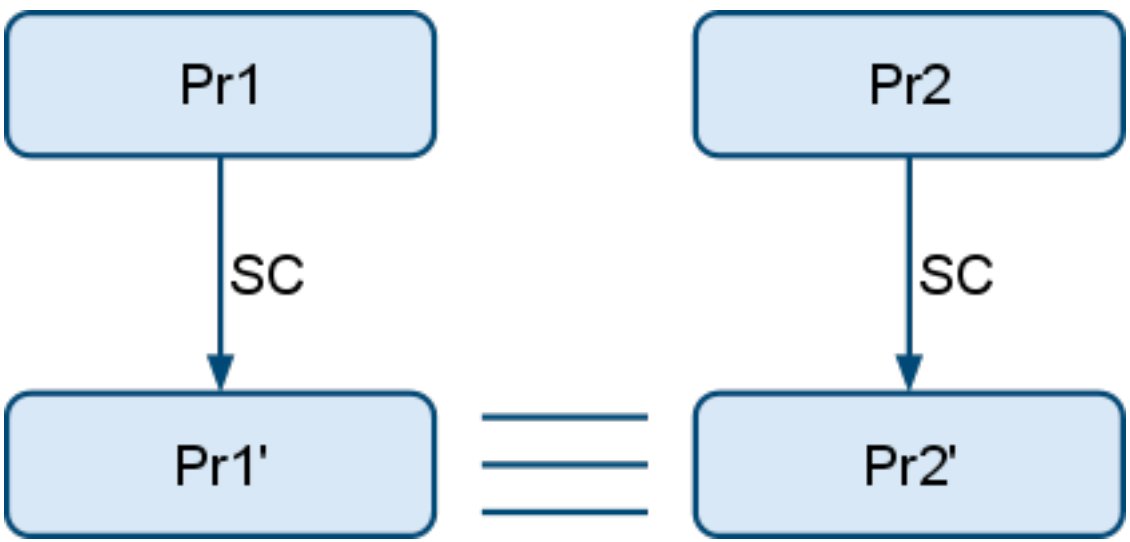
Klyuchnikov I.G. Supercompiler HOSC: proof of correctness. KIAM Preprint № 31, Moscow, 2010.

Outline

1. Program analysis by transformation
   - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
   - Checking correctness of monads
5. Higher-level supercompilation
   - Detecting improvement lemmas
   - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

Proving equivalence of expressions

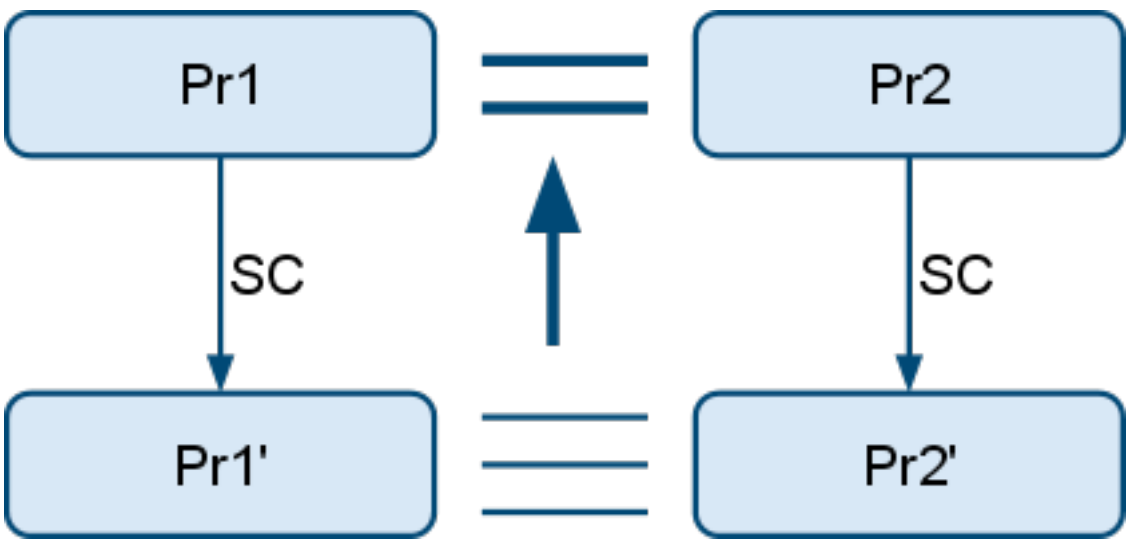
The basic idea



A.P. Lisitsa and M. Webster. Supercompilation for Equivalence Testing in Metamorphic Computer Viruses Detection. Proceedings of the First International Workshop on Metacomputation in Russia, 2008

Proving equivalence of expressions

The basic idea



It works only if SC preserves equivalence

(Multi-result supercompilation.)

Example: proof of equivalence
(how to prove?)

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

map = \f xs -> case xs of {
  Nil -> Nil;
  Cons y ys -> Cons (f y) (map f ys);
};

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

Proposition:

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

(for any functions **f** and **g**, any list, even infinite one)

Example: proof of equivalence
(normalizing by supercompilation!)

```
map f (map g xs)                    map (comp f g) xs
```


```
letrec h = \ys -> case ys of {
    Nil -> Nil;
    Cons z zs -> Cons (f (g z)) (h zs);
} in h xs
```

Proof of equivalence

Pros:

- Non-terminating calculations do not need special consideration
- Reasoning about infinite data
- Equivalence of functions may be prooved

Alternative approach:

```
equals(f(x), g(x)) => True
```

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.

HOSC 0: Testing on examples

HOSC 0 proved only 6 of 25 simple equalities (from the first chapter)

HOSC 1: proven equivalences

HOSC 1, using extended homeomorphic embedding was able to prove 25 of 25 equivalences

```
concatR xs == foldR NilR catR xs
listR (cross (P f g)) (zipR xs ys)
    == uncurry zipR (cross (P (listR f) (listR g)) (P xs ys))
listR f xs == foldR NilR (\a x -> Cons (f a) x) xs
filterR p xs
    == listR outl (filterR outr (uncurry zipR (pair (P id (listR p)) xs)))
listL (listL f) (inits xs) == inits (listL f xs)
listR f (concatR xs) == concatR (listR (listR f) xs)
mult1 x y == foldN Z (\n -> plus1 n y) x
mult1 x y == foldN Z (plus y) x
mult x y == foldN Z (plus x) y
plus1 x y == foldN y succ x
plus x y == foldN x succ y
filterR p xs ==
    foldR NilR (curry (cond (compose p outl) (uncurry cons) outr)) xs
filterR p xs == foldR1 NilR (cond (compose p outl) (uncurry cons) outr) xs
```

HOSC 1: proven equivalences

```
filterR p xs
    == (compose (foldR NilR catR)
        (foldR NilR (\a x -> Cons ((cond p wrapR nilpR) a) x))) xs
filterR p xs ==
    (compose (foldR NilR catR) (listR (cond p wrapR nilpR))) xs
filterR p xs == (compose concatR (listR (cond p wrapR nilpR))) xs
lengthR xs == (compose sum (listR (\x -> S Z))) xs
lengthR xs == foldR Z (\a n -> (S n)) xs
appendL xs ys == foldL1 snoc xs ys
appendL xs ys == foldL id (\f x a -> snoc (f a) x) ys xs
appendR xs ys == foldR id (\x f a -> cons x (f a)) xs ys
appendL xs ys == foldL xs snoc ys
appendR xs ys == foldR ys cons xs
appendL (appendL xs ys) zs == appendL xs (appendL ys zs)
appendR (appendR xs ys) zs == appendR xs (appendR ys zs)
```

Outline

1. Program analysis by transformation
   - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
   - Checking correctness of monads
5. Higher-level supercompilation
   - Detecting improvement lemmas
   - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

Application: checking monadic laws

1. `join (return a) k == k a`
2. `join m (\x -> join (k x) h) == join (join m k) h`
3. `join m return == id m`
4. `fmap (compose f g) xs == compose (fmap f) (fmap g) xs`
5. `fmap id xs == id xs`
6. `fmap f xs == join xs (compose return f)`
7. `join mzero f == mzero`
8. `bind v mzero == mzero`

Application: checking monadic laws

Maybe monad

```
data Maybe a = Just a | Nothing;
compose = \f g x ->  f (g x);
return = \x -> Just x;
fmap = \f m -> case m of {
    Nothing -> Nothing;
    Just x -> Just (f x);
};
join = \m k -> case m of {
    Nothing -> Nothing;
    Just x -> k x;
};
id = \m -> case m of {
    Nothing -> Nothing;
    Just x -> Just x;
};
gid = \x -> x;
bind = \m k -> join m (\x -> k);
mzero = Nothing;
```

Application: checking monadic laws

Maybe Monad

1. +
2. +
3. +
4. +
5. +
6. +
7. +
8. -

List Monad

1. +
2. +
3. +
4. +
5. +
6. +
7. +
8. -

State Monad

1. +
2. +
3. +
4. +
5. +
6. +
7. +
8. +

Detecting "errors"

8. Maybe

```
bind v mzero
```


```
case  v  of {
  Nothing  -> Nothing;
  Just s -> Nothing;}
```

=?=

```
mzero
```


```
Nothing
```

8. List

```
bind v mzero
```


```
letrec f= \x -> case x of {
   Nil  -> Nil;
   Cons y z -> (f z); }
in f v
```

=?=

```
mzero
```

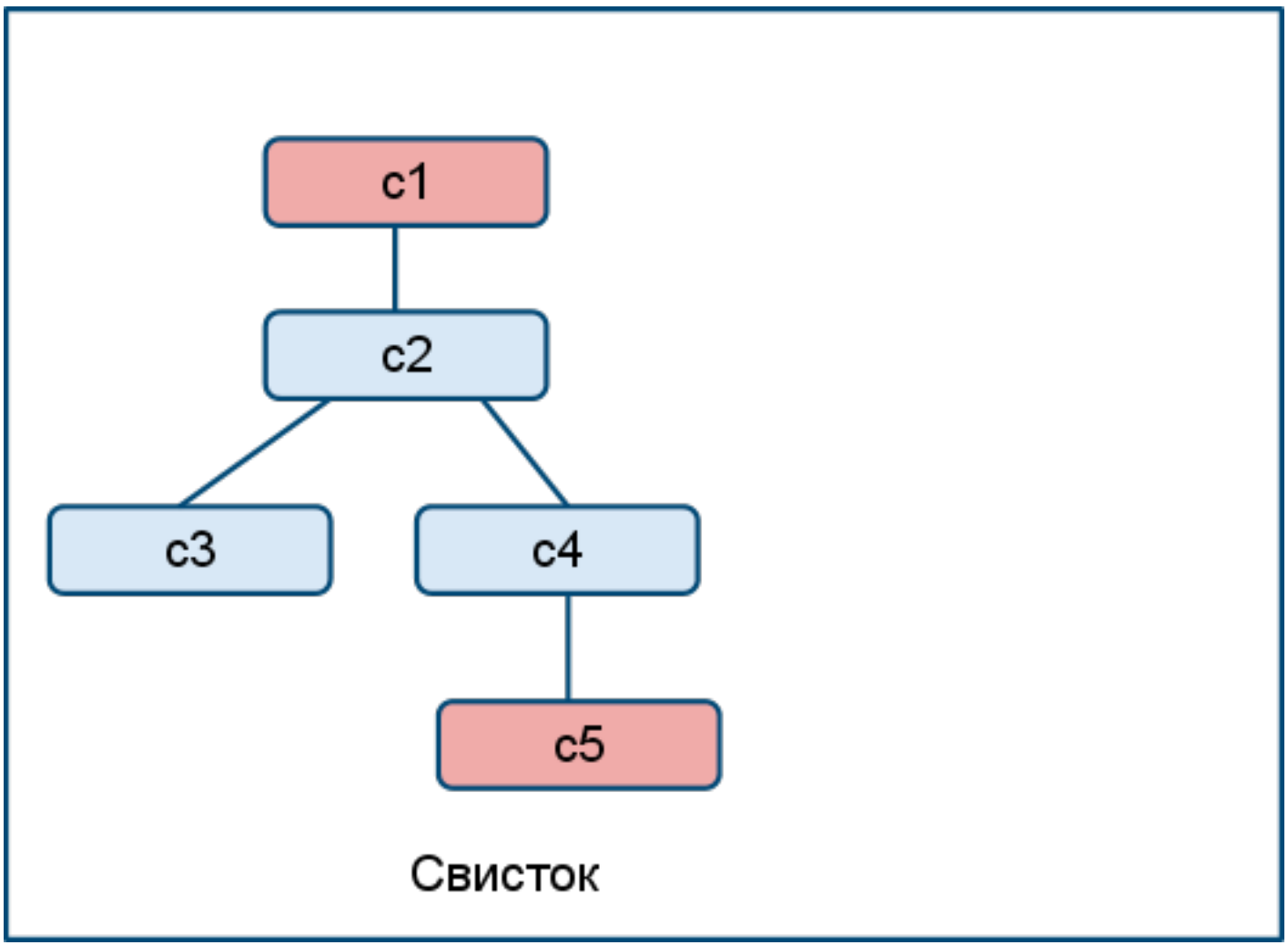

```
Nil
```

Residual program may supply useful information

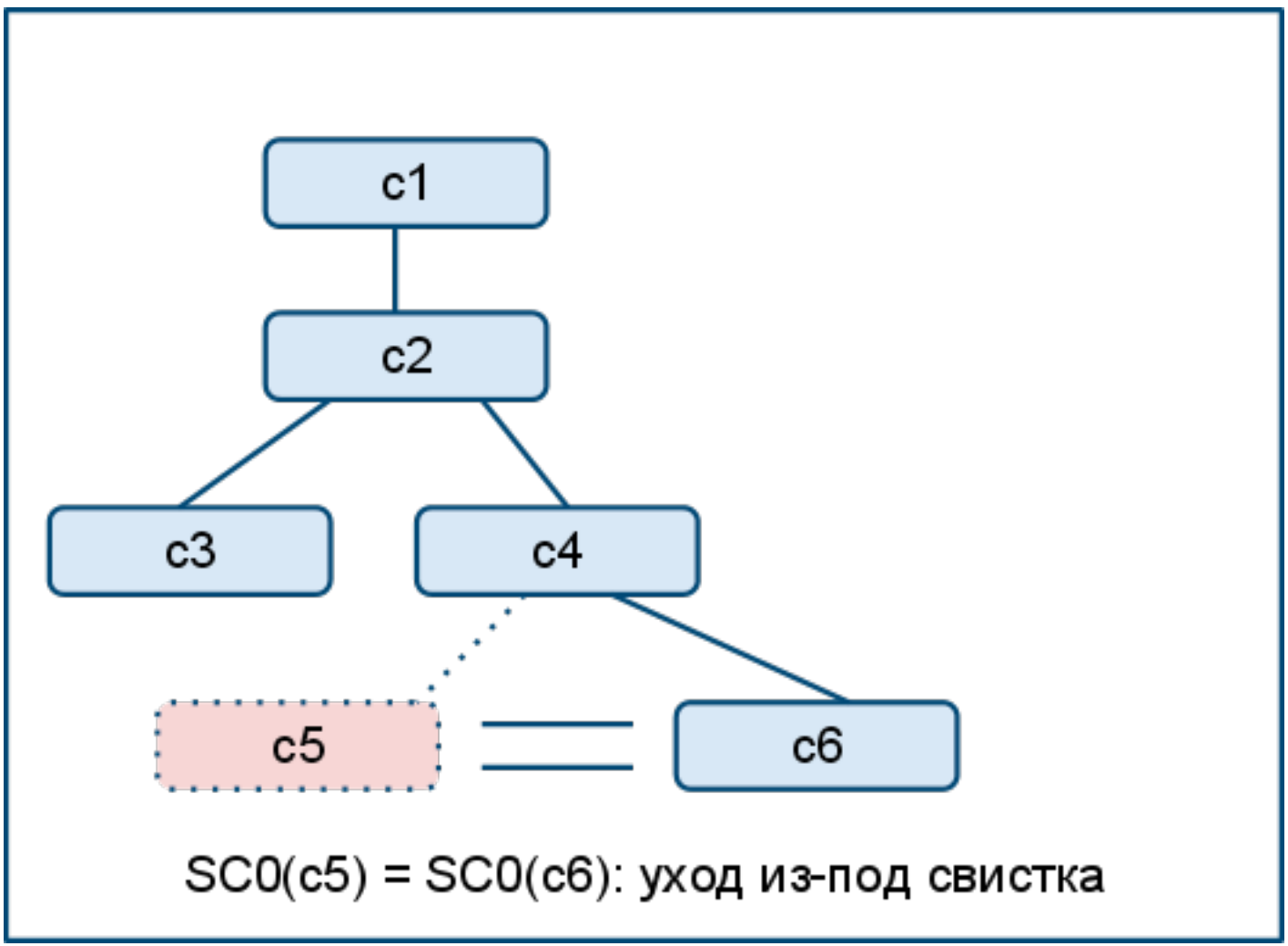
Outline

1. Program analysis by transformation
   - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
   - Checking correctness of monads
5. Higher-level supercompilation
   - Detecting improvement lemmas
   - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

Higher-level supercompilation



Higher-level supercompilation



Higher-level supercompilation

Base supercompiler:

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

The second-level supercompiler:

```
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
}
```

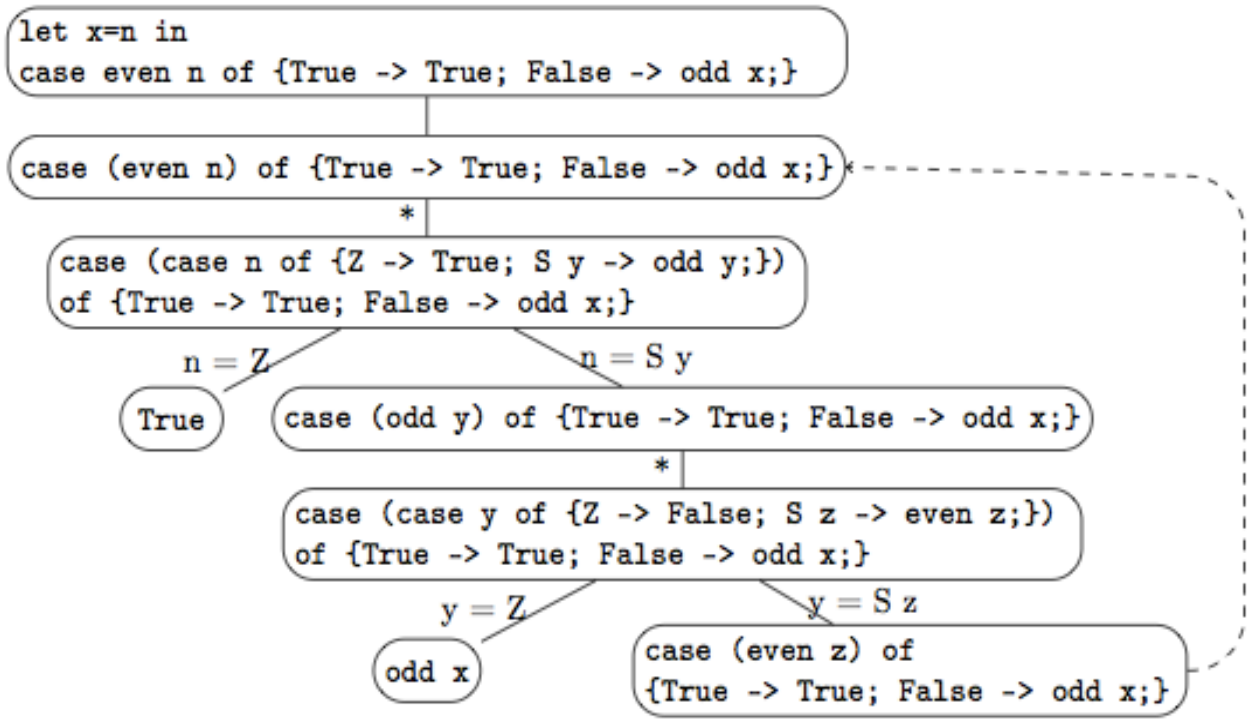
The correctness of higher-level supercompilation

Theorem (Sands, 1996). When performing fold/unfold transformation it is safe to replace e1 by e2 if e2 is a strong improvement of e1.

If e2 if a strong improvement of e1, then
(e1, e2) is an **improvement lemma**

Comparing costs of computations

Annotating a partial process tree



Propagating annotations into a 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
```

The correctness of higher-level supercompilation

Theorem (Klyuchnikov). Let

prog'1 - a residual expression for e1
prog'2 - a residual expression for e2

If prog'1 syntactically equivalent to prog'2 (ignoring annotations) and prog'2 is embedded into prog'1 by annotations then e2 is a strong improvement of e1 and it is correct to replace e1 на e2. when constructing a partial process tree.

Ilya Klyuchnikov and Sergei Romanenko. Towards Higher-Level Supercompilation. In: *Second International Workshop on Metacomputation in Russia*. 2010.

Outline

1. Program analysis by transformation
   - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
   - Checking correctness of monads
5. Higher-level supercompilation
   - Detecting improvement lemmas
   - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

Higher-level supercompilation

Example 1. Accumulating parameter. 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));};
```

Higher-level supercompilation

Example 1. Accumulating parameter.

Classical supercompilation:

```
letrec f=\w2 p2-> case  w2  of {
  Z -> letrec g=\r2-> 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
```

Higher-level supercompilation:

```
case  x  of {
  Z -> True;
  S y1 -> letrec f=\t2->
     case  t2  of {Z  -> True; S u2 -> f u2;}
    in f y1;
}
```

Higher-level supercompilation

Example 2 - Non-linear expression. Input

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

or (even m) (odd m) where

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

or = \x y -> case x of { True -> True; False -> y;};
```

Higher-level supercompilation

Example 2 - Non-linear expression

Classical supercompilation:

```
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 m;
    S x -> f x;};} in f m
```

Higher-level supercompilation:

```
letrec f=\w-> case w of {
     Z -> True;
     S x -> case x  of { Z  -> True; S z -> f z;};
  } in f m
```

Higher-level supercompilation

Example 3 - Improving the asymptotic. Non-optimal parser. Complexity: O(n2)

```
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;};
```

Higher-level supercompilation

Example 3 - Improving the asymptotic. Classical supercompilation: - complexity: O(n2)

```
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);
}
```

Higher-level supercompilation

Example 3 - Improving the asymptotic. Higher-level supercompilation. 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
```

Higher-level supercompilation

Example 3 - Improving the asymptotic

Input:

```
p = a p a | empty.
```

Output:

```
p' = a a p' | empty.
```

Equivalence of grammars is shown!
The general idea:

- Coding knowledge as a program.
- Analysis of a program by transformations.
- Result: inference and proving of properties of objects being modeled

Higher-level supercompilation

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

We have shown that a higher-level supercompiler is able to improve he asymptotic of a program.

Outline

1. Program analysis by transformation
   - Problem definition
2. What is supercompilation
3. Supercompiler HOSC
4. Detecting equivalent expressions
   - Checking correctness of monads
5. Higher-level supercompilation
   - Detecting improvement lemmas
   - Examples of higher-level supercompilation
6. On the equivalence of abstract machines

Equivalence of abstract machines

Two approches:

- State-transition function together with a ‘driver loop’.
- Collection of mutually tail-recursive transition functions mapping a given configuration to a final state.

Olivier Danvy, Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion.
I**nformation Processing Letters**. Volume 106, Issue 3 (**April 2008**).

Equivalence of abstract machines

CEK-machine: definitions

```
data Nat = Z | S Nat;
data List a = Nil | Cons a (List a);
data Term = Var Nat | Lam Term | App Term Term;
data Val = Clo Term (List Val);
data RC = RC0 | RC1 RC Term (List Val) | RC2 Val RC;
data Conf = Eval Term (List Val) RC | Apply RC Val;
data State = Final Val | Inter Conf;

lookup = \i env ->
  case env of {
    Cons n env1 ->
      case i of {
        Z -> n;
        S i1 -> lookup i1 env1;
      };
  };
```

Equivalence of abstract machines

CEK-machine: small-step semantics

```
drive (Inter (Eval t Nil RC0)) where

move = \conf -> case conf of {
  Eval t e c ->
    case t of {
      Var i -> Inter (Apply c (lookup i e));
      Lam t0 -> Inter (Apply c (Clo t0 e));
      App t0 t1 -> Inter (Eval t0 e (RC1 c t1 e));};
  Apply c v ->
    case c of {
      RC0 -> Final v;
      RC1 c1 t1 e -> Inter (Eval t1 e (RC2 v c1));
      RC2 v1 c1 -> case v1 of {
        Clo t2 e2 -> Inter (Eval t2 (Cons v e2) c1);};};};};

drive = \state -> case state of {
  Final v -> v;
  Inter conf -> drive (move conf);};
```

Equivalence of abstract machines

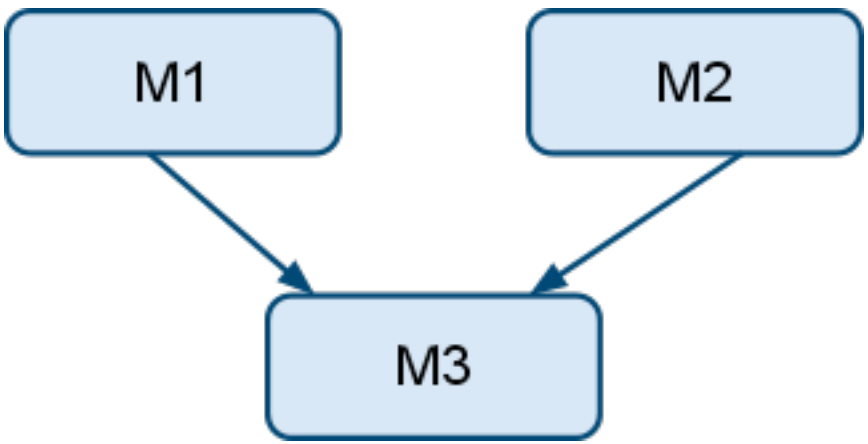
CEK-machine: big-step semantics

```
eval t Nil RC0 where

eval = \t e c ->
  case t of {
    Var i -> apply c (lookup i e);
    Lam t1 -> apply c (Clo t1 e);
    App t0 t1 -> eval t0 e (RC1 c t1 e);
  };

apply = \t v ->
  case t of {
    RC0 -> v;
    RC1 c1 t1 e1 -> eval t1 e1 (RC2 v c1);
    RC2 v1 c1 -> case v1 of {
      Clo t2 e2 -> eval t2 (Cons v e2) c1;
    };
  };
```

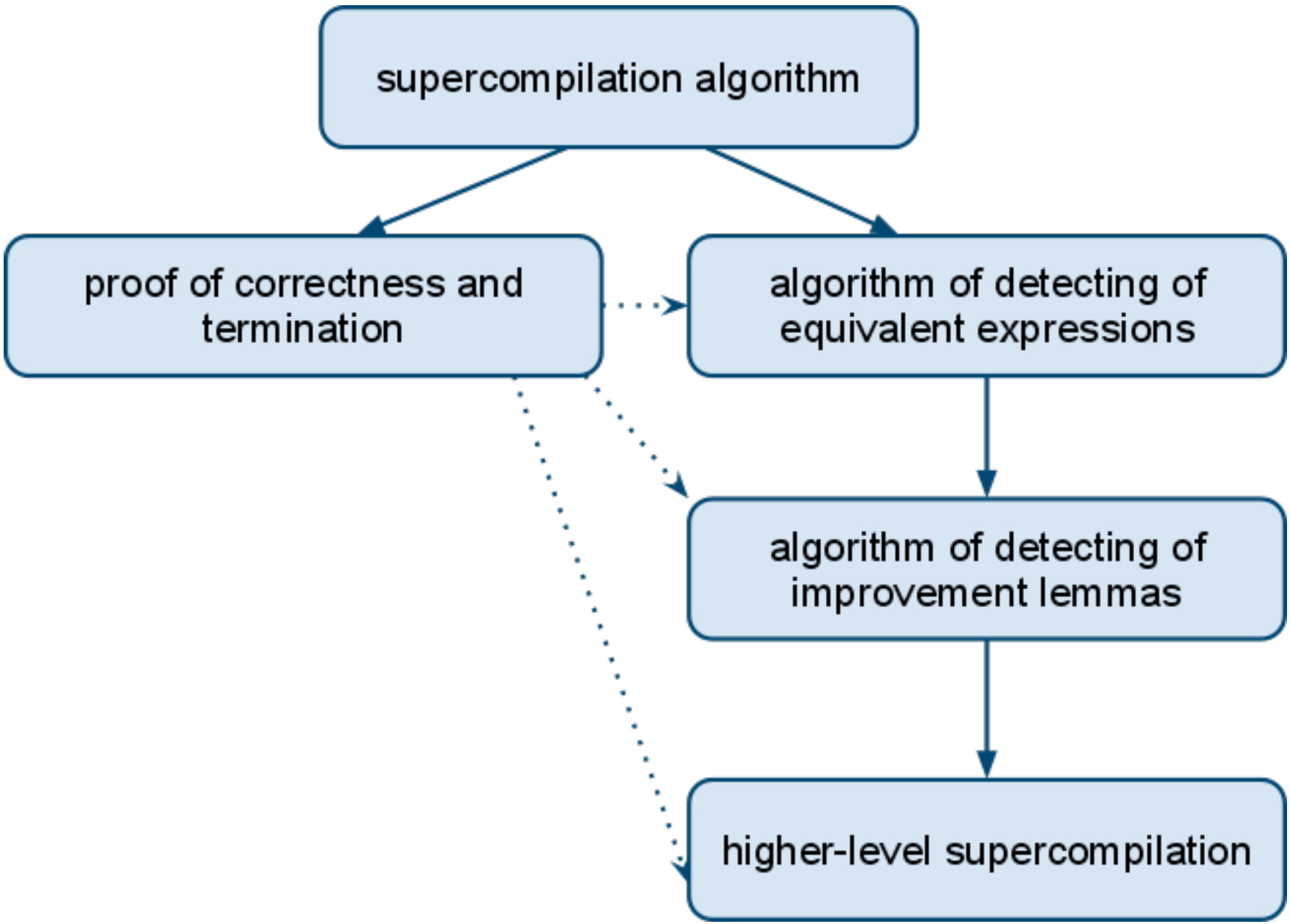
Equivalence of abstract machines

Danvy, Millikin:

Normalization by supercompilation



Mind-map of methods



Contribution

- The algorithm of higher-order supercompilation is developed. This algorithm takes into account properties of bound variables:
  - The homeomorphic embedding relation was extended.
  - A new algorithm of generalization of coupled expressions.
  - Proof of termination, proof of correctness.
- A method of higher-level supercompilation (capable to improve the asymptotic behavior of programs):
  - The algorithm for detecting the equivalence of higher-order expressions.
  - The algorithm for detecting improvement lemmas
- Applicability of suggested methods to program analysis by supercompilation was demonstrated.

Publications

1. **Илья Ключников, Сергей Романенко. SPSC: Суперкомпилятор на языке Scala. // Программные продукты и системы. 2009. №2 (86)**
2. Ilya Klyuchnikov and Sergei Romanenko. SPSC: a Simple Supercompiler in Scala. In: International Workshop on Program Understanding 19-23 June, Altai Mountains, Russia. 2009
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.**
4. Ilya Klyuchnikov and Sergei Romanenko. Towards Higher-Level Supercompilation. In: Second International Workshop on Metacomputation in Russia. 2010

Preprints

1. Klyuchnikov I.G. Supercompiler HOSC 1.0: under the hood KIAM Preprint № 63, Moscow, 2009.
2. Klyuchnikov I.G. Supercompiler HOSC 1.1: proof of termination. KIAM Preprint № 21, Moscow, 2010.
3. Klyuchnikov I.G. Supercompiler HOSC: proof of correctness. KIAM Preprint № 31, Moscow, 2010.