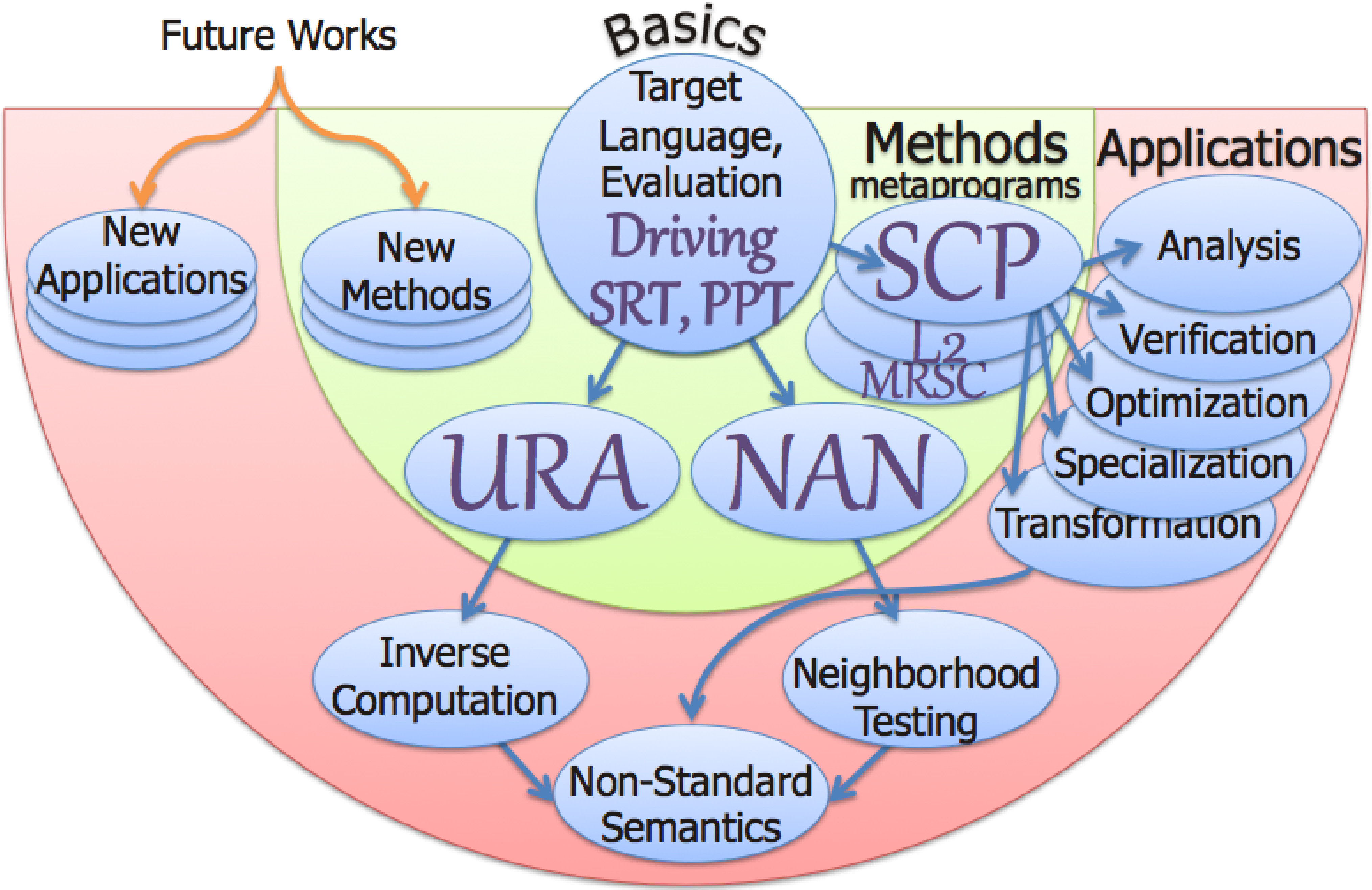
Metacomputation
A Gentle Introduction to Advanced Topics

Ilya Klyuchnikov

Keldysh Institute of Applied Mathematics
(Russian Academy of Sciences)

Metacomputation: The Big Picture



Future Works

Basics

Target Language, Evaluation

Driving

SRT, PPT

Methods

metaprograms

SCP

L2

MRSC

Applications

Analysis

Verification

Optimization

Specialization

Transformation

New Applications

New Methods

URA

NAN

Inverse Computation

Neighborhood Testing

Non-Standard Semantics

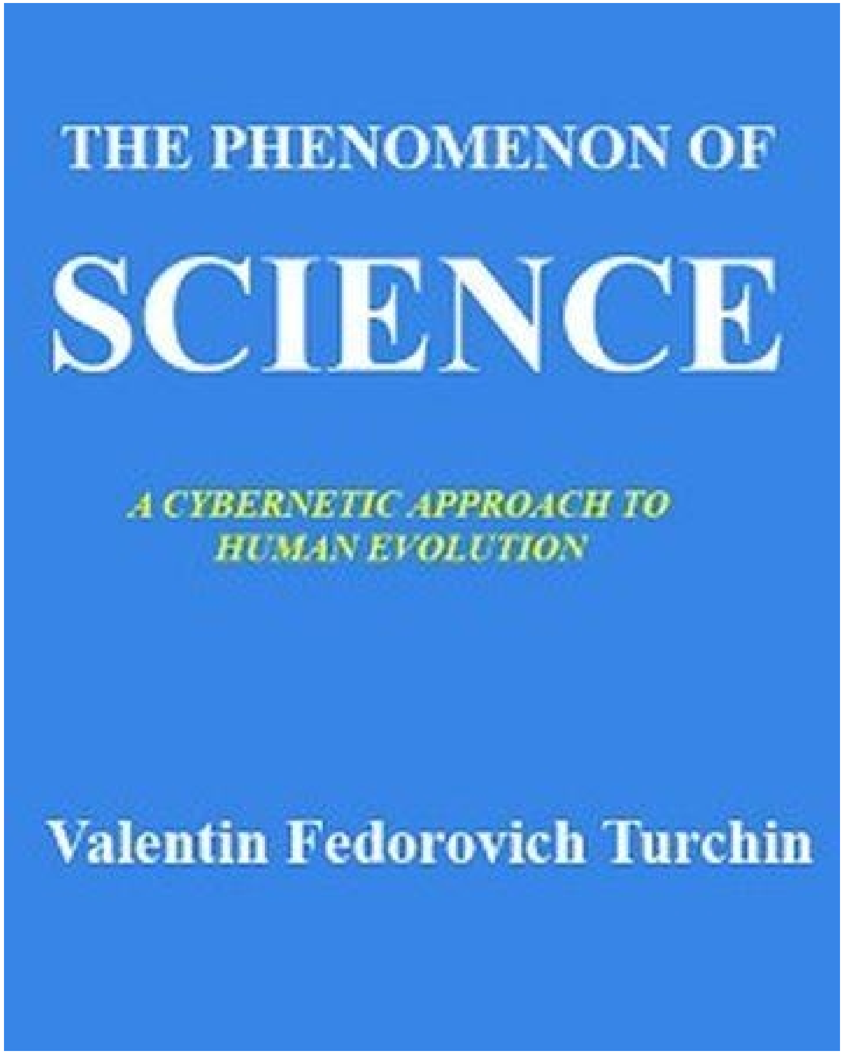
Disclaimer Notice

The following presentation simplifies technical
stuff A LOT in order to fit 1.5 hours
and give you a taste of the area.

Examples are also small for the same reason.

Please consult references for details.

Valentin Turchin (1931-2010)



THE PHENOMENON OF

SCIENCE

A CYBERNETIC APPROACH TO HUMAN EVOLUTION

Valentin Fedorovich Turchin

- The concept of metasystem transition
- The concept of supercompilation

These two concepts are related (I will try to show this at the end of the talk).

The Plan

- Supercompilation in a nutshell
- Optimization vs Analysis
- Analyzing supercompilation (HOSC)
- Two-level supercompilation
- Multi-result supercompilation (MRSC)
- Finding a minimal proof by multi-result supercompilation
- On metasystem transitions

The Plan

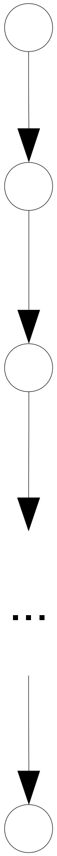
- Supercompilation in a nutshell
- Optimization vs Analysis
- Analyzing supercompilation (HOSC)
- Two-level supercompilation
- Multi-result supercompilation (MRSC)
- Finding a minimal proof by multi-result supercompilation
- On metasystem transitions

Supercompilation in a Nutshell

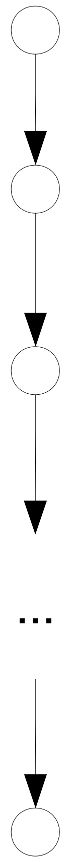
- Driving
- Folding
- Whistle
- Generalization

V. Turchin. The concept of a supercompiler / 1986
M. Sørensen, R. Glück, and N. Jones. A Positive Supercompiler /1996
M. Sørensen and R. Glück. Introduction to Supercompilation / 1998

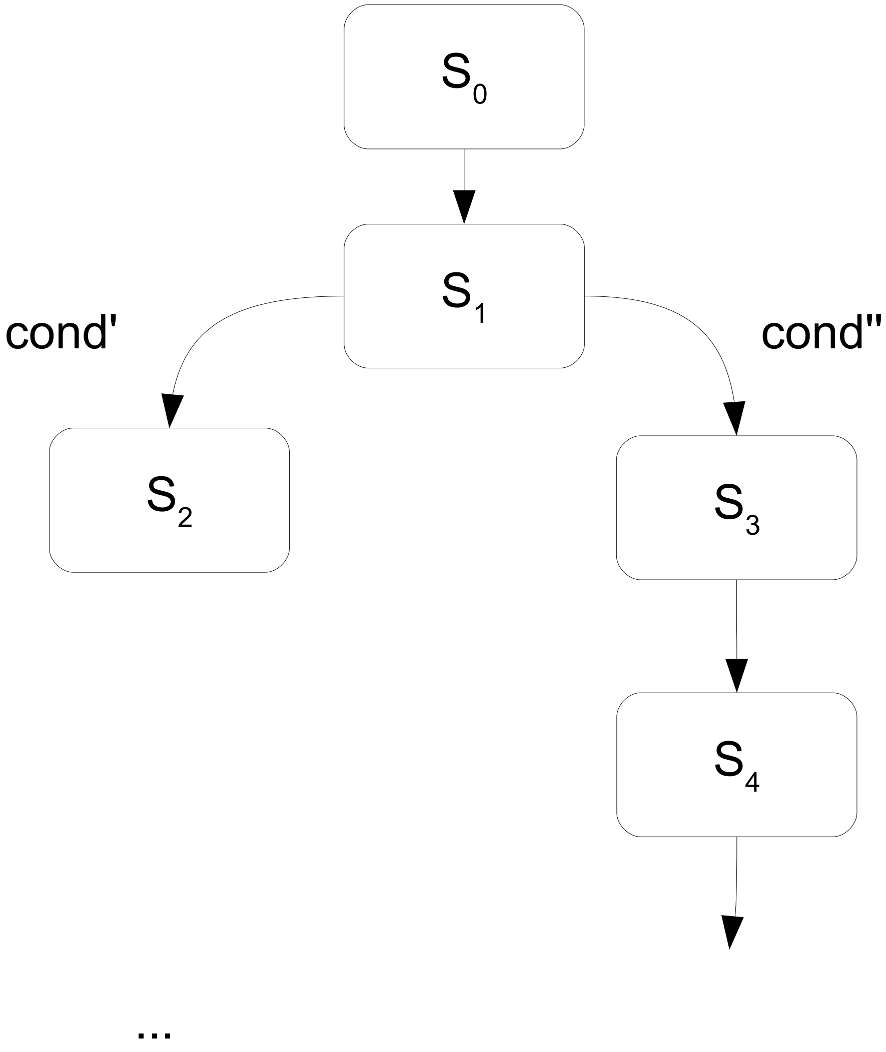
Execution



Execution

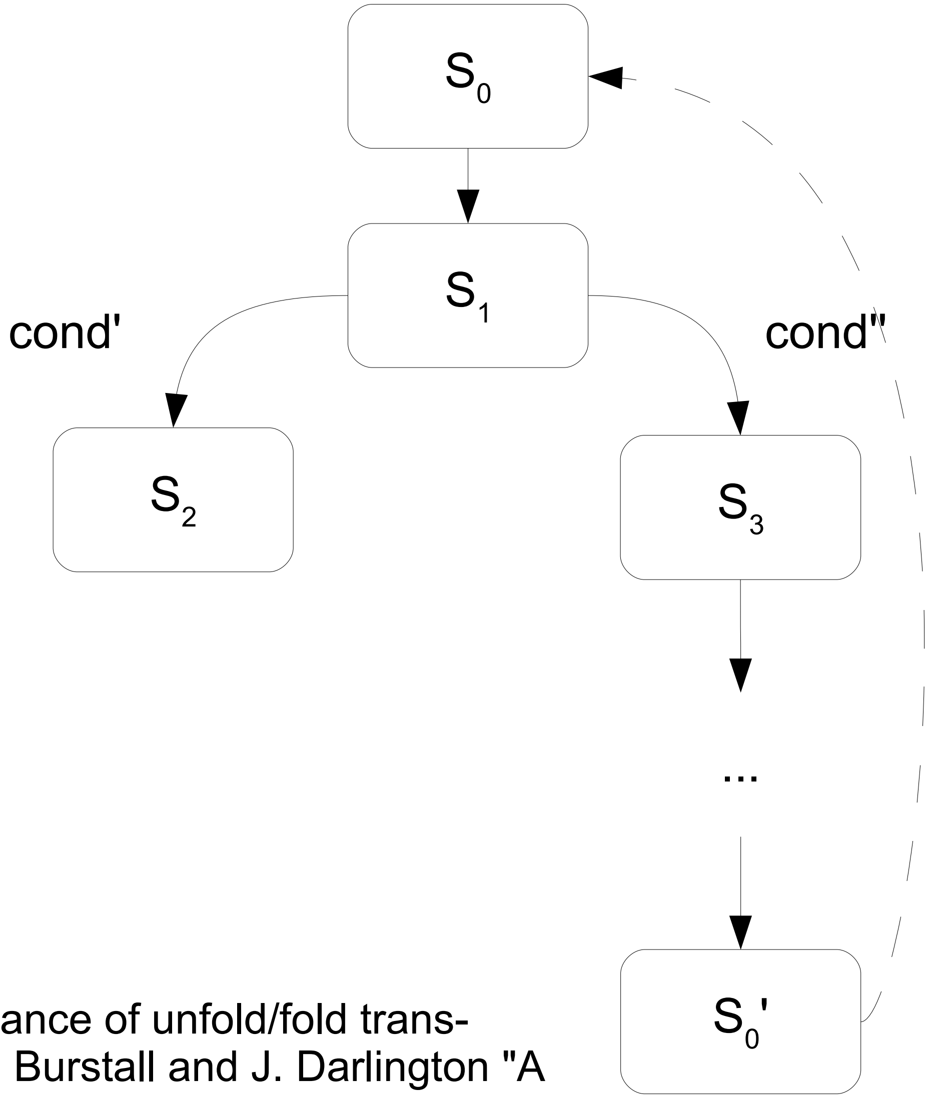


Driving



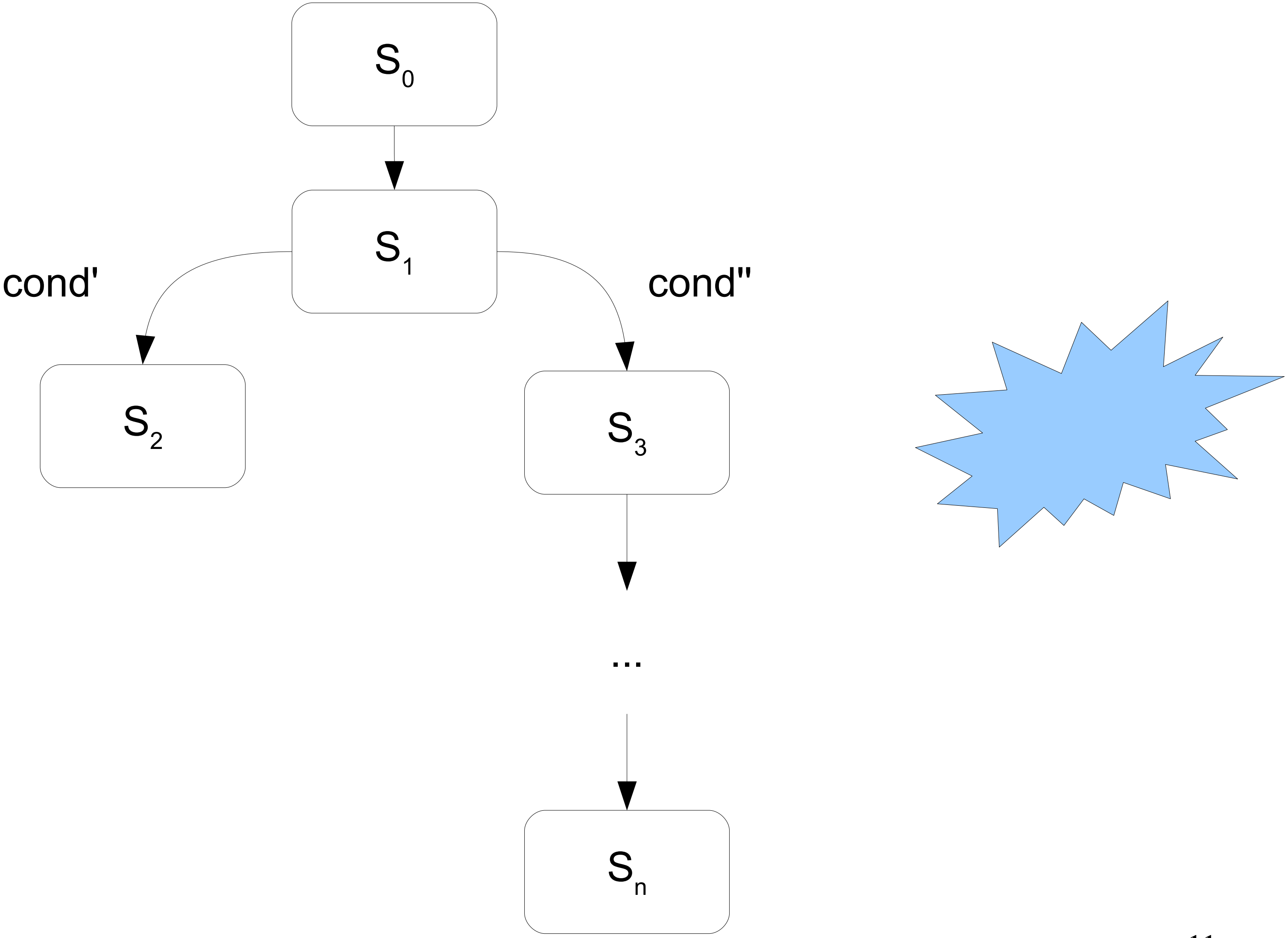
A. Klimov, R. Glück. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree / 1993
N. Jones. The Essence of Program Transformation by Partial Evaluation and Driving / 1999

Folding



Supercompilation is an instance of unfold/fold transformation defined in: R. M. Burstall and J. Darlington "A Transformation System for Developing Recursive Programs" / 1977

Whistle



S_0

S_1

cond'

cond''

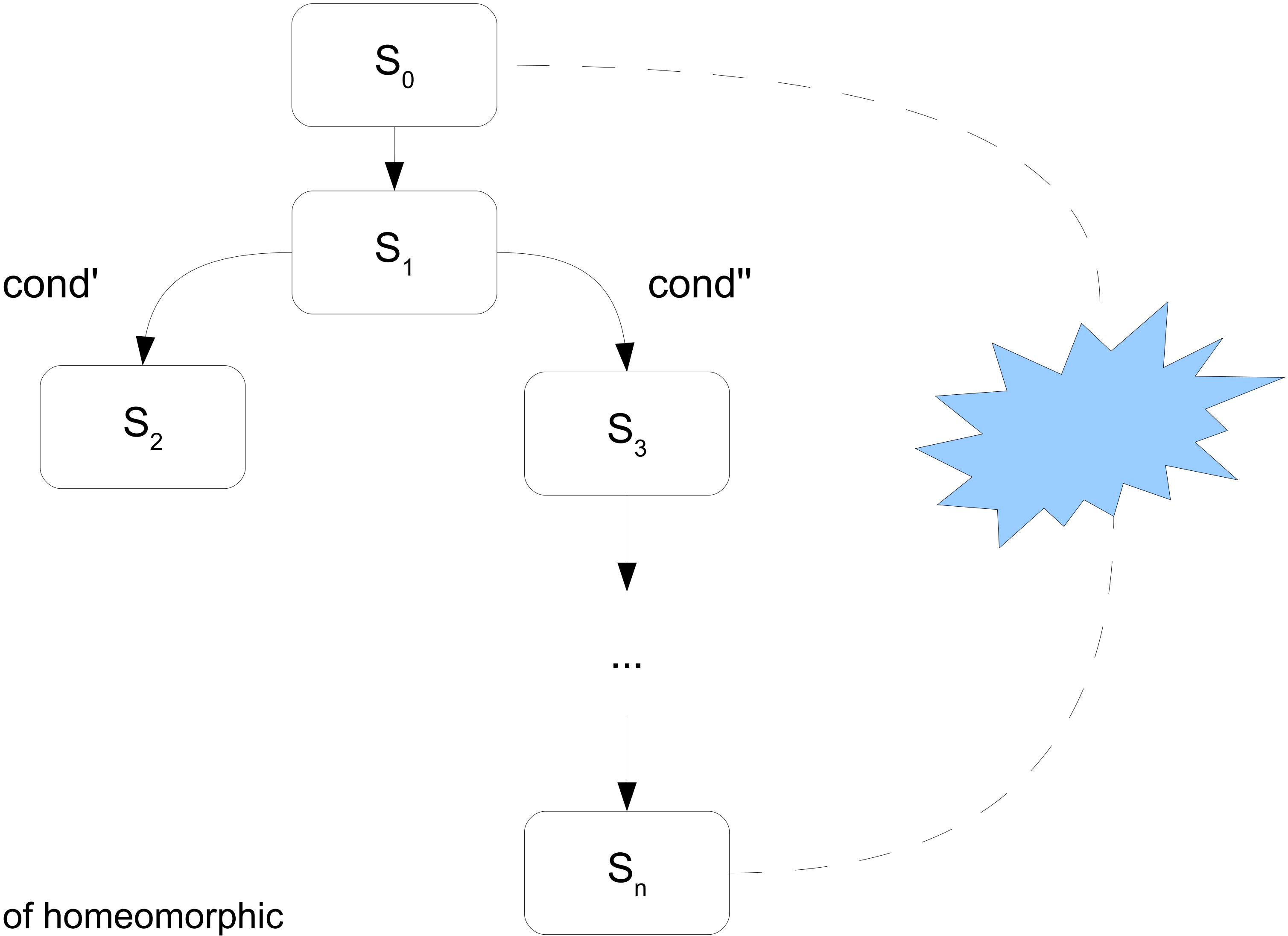
S_2

S_3

...

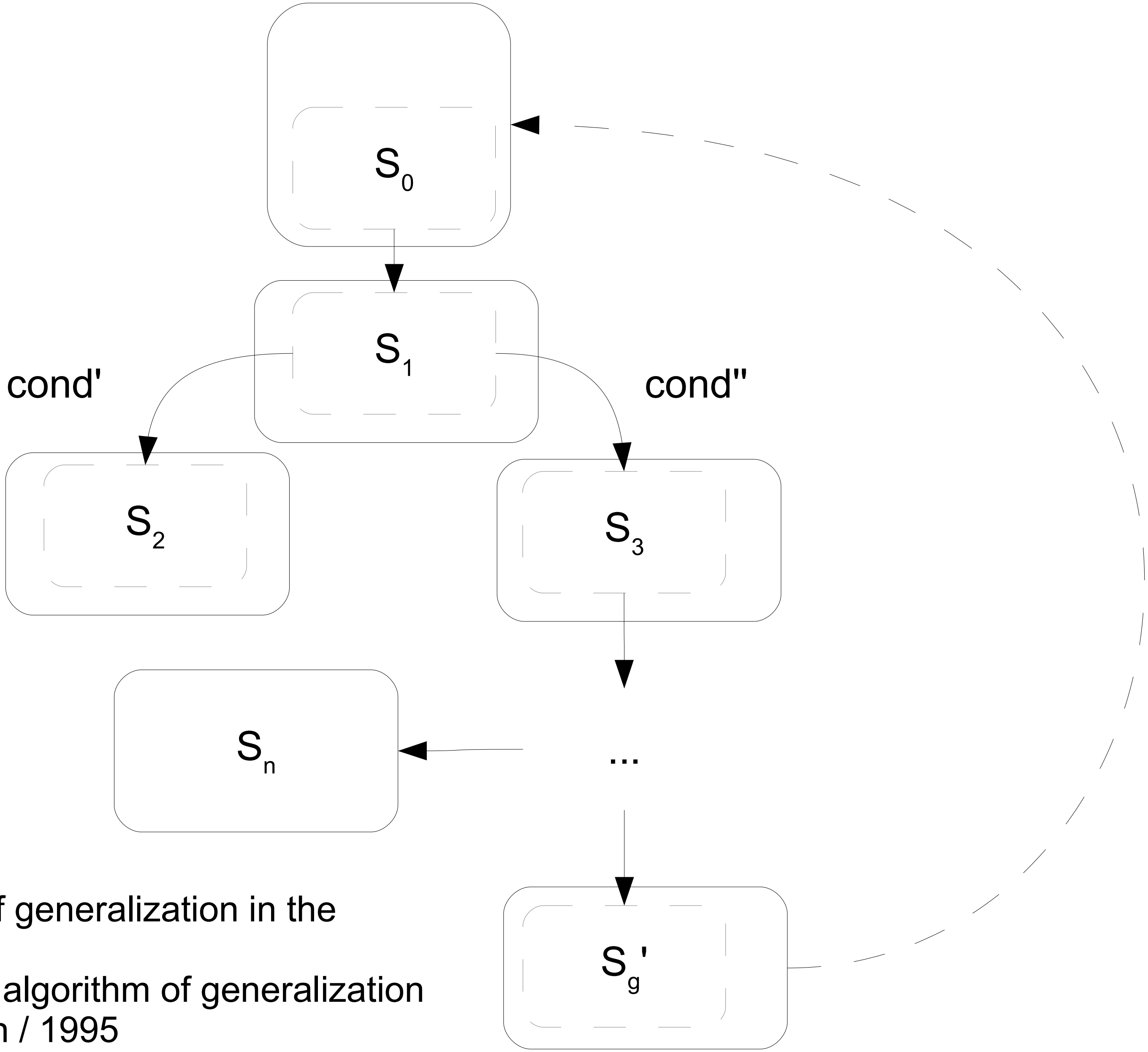
S_n

Binary Whistle (standard approach)



M. Leuschel. On the power of homeomorphic
embedding for online termination. 1998

Generalization



V. Turchin. The algorithm of generalization in the supercompiler / 1988
M. Sørensen, R. Glück. An algorithm of generalization in positive supercompilation / 1995

History of Supercompilation

- 1970-1990s - Supercompilation for Refal Language (V. Turchin et al)
- 1990s – Supercompilation of First-Order Functional languages
- 2000s – Supercompilation of Higher-Order Functional languages

There are 2 trends in supercompilation community: program optimization, program analysis.

Existing Supercompilers

- SCP4 (1990s)
- SCP for TSG (2000s)
- Jscp (2000s)
- SCP for Timber (2007)
- Supero (2007)
- SPSC (2008)
- HOSC (2008)
- Optimusprime (2009/10)
- CHSC (2010)
- Distiller (2009/10)
- MRSC (2011)

I. Klyuchnikov. The ideas and methods of supercompilation. Practice of functional programming. Vol.7. 2011 [In Russian].

“Analyzing” Supercompilers

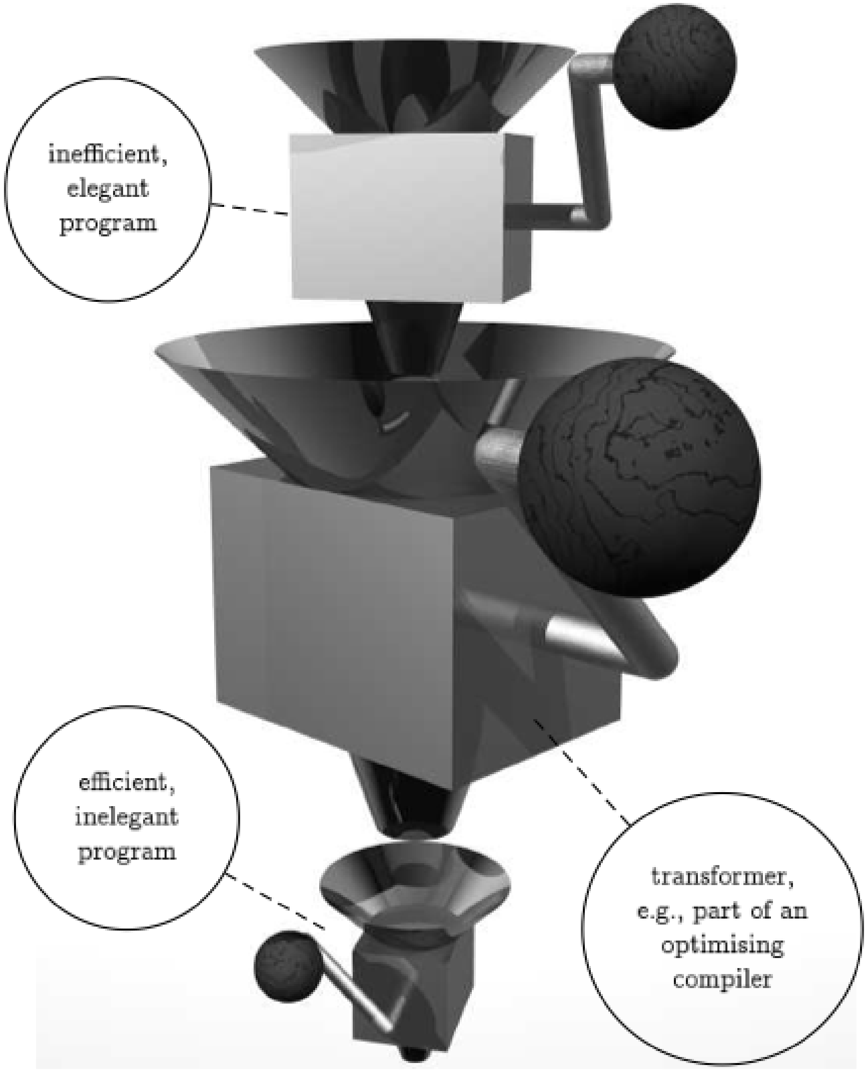
- SCP4 (1990s)
- SCP for TSG (2000)
- Jscp (2000)
- SCP for Timber (2007)
- Supero (2007)
- SPSC (2008)
- HOSC (2008)
- Optimusprime (2009/10)
- CHSC (2010)
- Distiller (2009/10)
- MRSC (2011)

I. Klyuchnikov. The ideas and methods of supercompilation. Practice of functional programming. Vol.7. 2011 [In Russian].

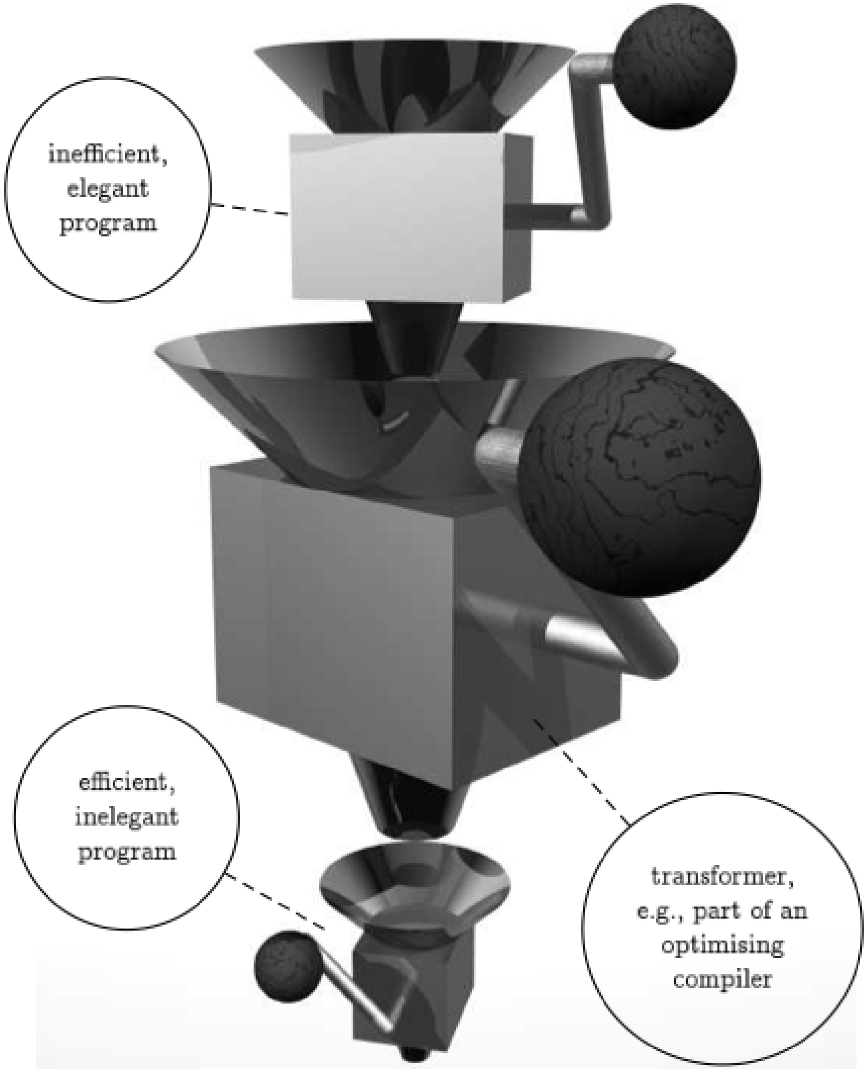
The Plan

- Supercompilation in a nutshell
- Optimization vs Analysis ←
- Analyzing supercompilation (HOSC)
- Two-level supercompilation
- Multi-result supercompilation (MRSC)
- Finding a minimal proof by multi-result supercompilation
- On metasystem transitions

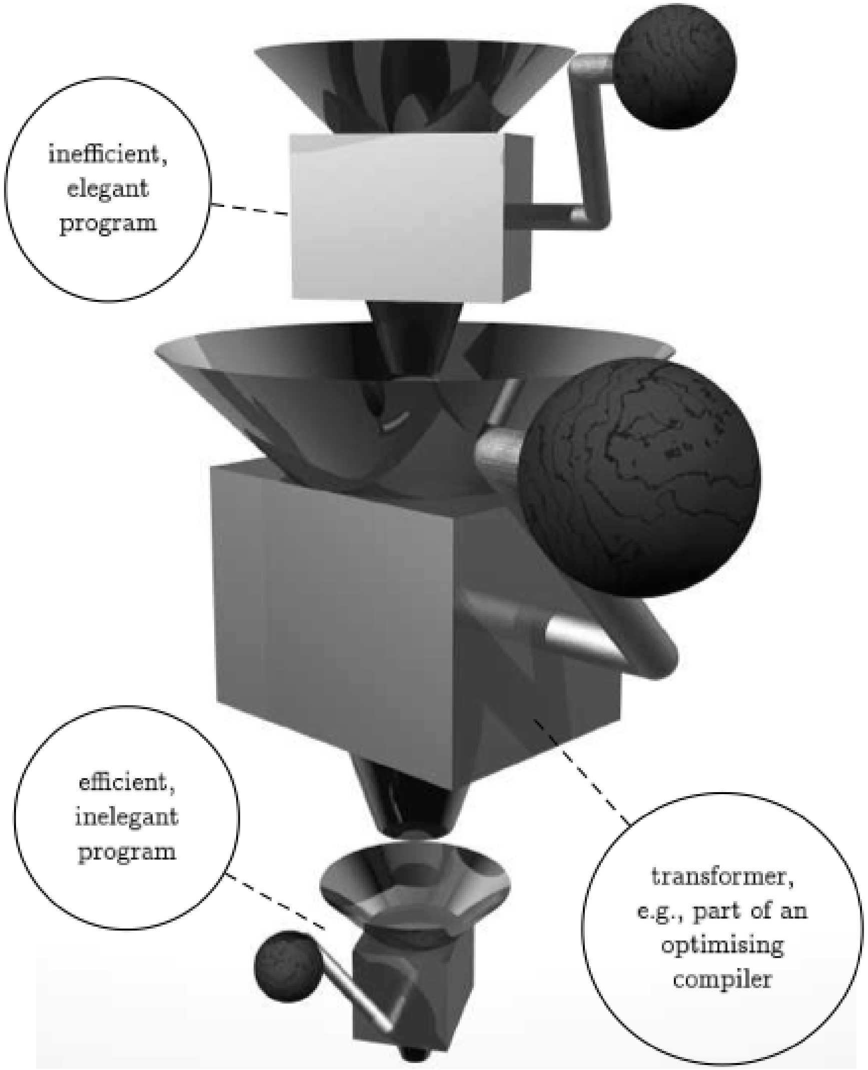
Optimization vs Analysis

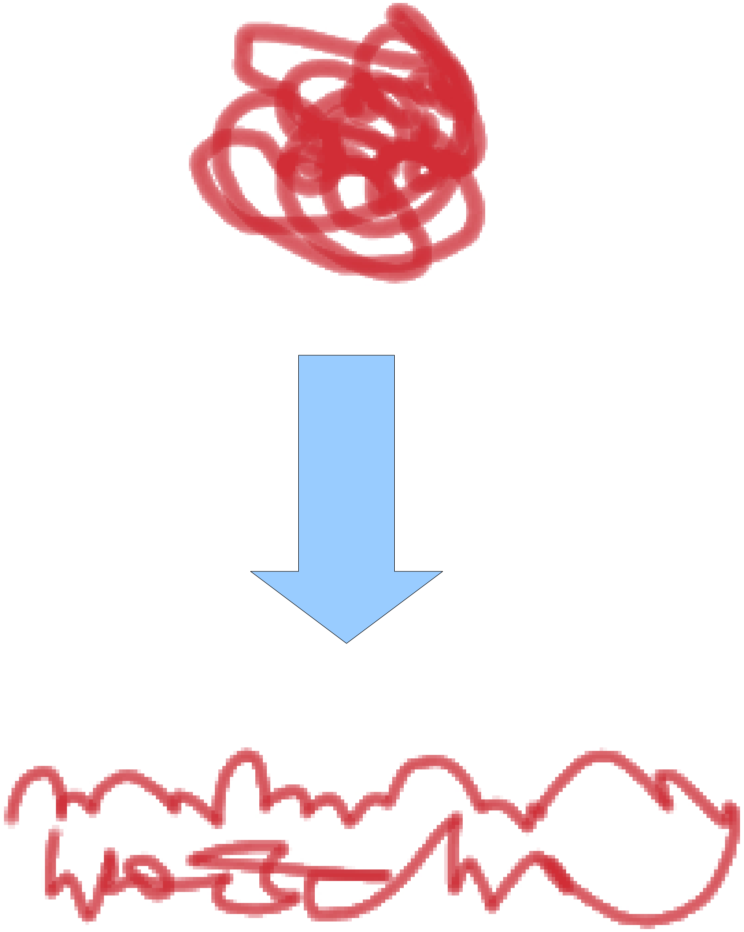


Optimization vs Analysis



Optimization vs Analysis





Optimization vs Analysis

- Reducing execution time
- Reducing code size

- Simplifying the structure
- Revealing hidden properties

Optimization vs Analysis

- Reducing execution time
- Reducing code size

The “best” ouput program.

- Simplifying the structure
- Revealing hidden properties

The set of output programs.

The story of development of analyzing supercompilers

From HOSC (2008) to MRSC (2011)

The Plan

- Supercompilation in a nutshell
- Optimization vs Analysis
- Analyzing supercompilation (HOSC)
- Two-level supercompilation
- Multi-result supercompilation (MRSC)
- Finding a minimal proof by multi-result supercompilation
- On metasystem transitions

From HOSC to MRSC

HOSC (Higher-Order Supercompiler) –
an analyzing supercompiler for core Haskell:
http://code.google.com/p/hosc/

MRSC (Multi-Result Supercompiler) –
a framework for rapid development of different supercompilers:
https://github.com/ilya-klyuchnikov/mrsc

The HOSC Supercompiler (2008)

HOSC is intended for program analysis rather than for program optimization:

- Code duplication is allowed
- “Controversial” (from optimization point) transformation is allowed

The trick: we treat call-by-need programs as call-by-name ones.

The consequence: tendency “to normalize” programs.

I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. PSI-2009.

Example #1. Church numbers

```
0 = \s z -> z

1 = \s z -> s z

2 = \s z -> s (s z)

3 = \s z -> s (s (s z))

...
```

`n = \s z ->` s^n `z`

f^{m+n} `z ->` s^m `(`s^n `z)`

f^{m*n} `z ->` $((s^n)^m)$ `z`

Example #1. Church numbers

```
data Nat = Z | S Nat;

foldn = \h z s -> case x of { Z -> z; S n1 -> s (foldn s z n1); };

add = \x y -> foldn S y x;

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

church = \n -> foldn (\m f x -> f (m f x)) (\f x -> x) n;

unchurch = \n -> n S Z;

churchMult = \m n f -> m (n f);


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

Example #1. Church numbers

```
data Nat = Z | S Nat;

foldn = \h z s -> case x of { Z -> z; S n1 -> s (foldn s z n1);};

add = \x y -> foldn S y x;

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

church = \n -> foldn (\m f x -> f (m f x)) (\f x -> x) n;

unchurch = \n -> n S Z;

churchMult = \m n f -> m (n f);


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


letrec f = \m n -> case m of {

  Z  -> Z;

  S m1 -> letrec g = \z -> case z of { S v -> S (g v); Z -> f m1 n; } in g n;

} in f x y
```

Example #1. Church numbers

```
data Nat = Z | S Nat;

foldn = \h z s -> case x of { Z -> z; S n1 -> s (foldn s z n1);};

add = \x y -> foldn S y x;

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

church = \n -> foldn (\m f x -> f (m f x)) (\f x -> x) n;

unchurch = \n -> n S Z;

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

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

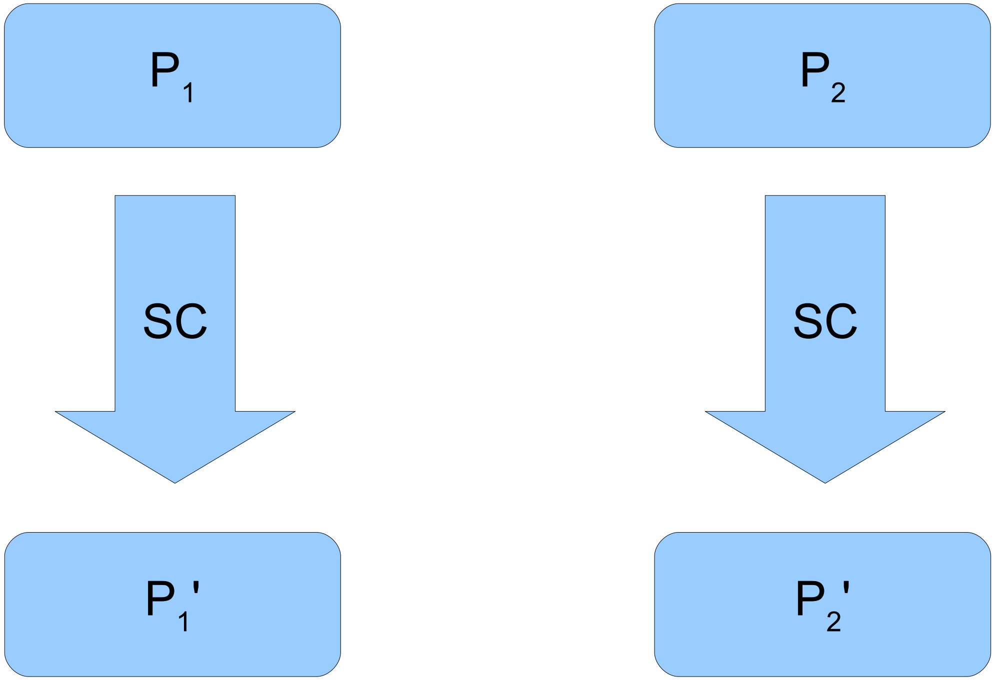
```
letrec f = \m n -> case m of {
  Z  -> Z;
  S m1 -> letrec g = \z -> case z of { S v -> S (g v); Z -> f m1 n; } in g n;
} in f x y
```

Inferring the equivalence of programs

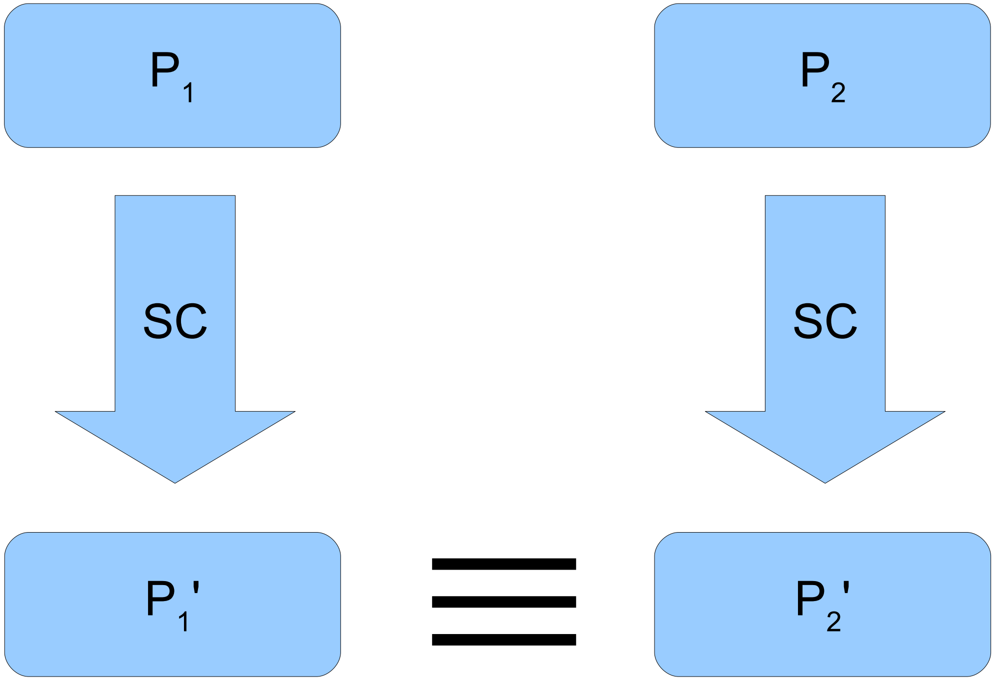
P_1

P_2

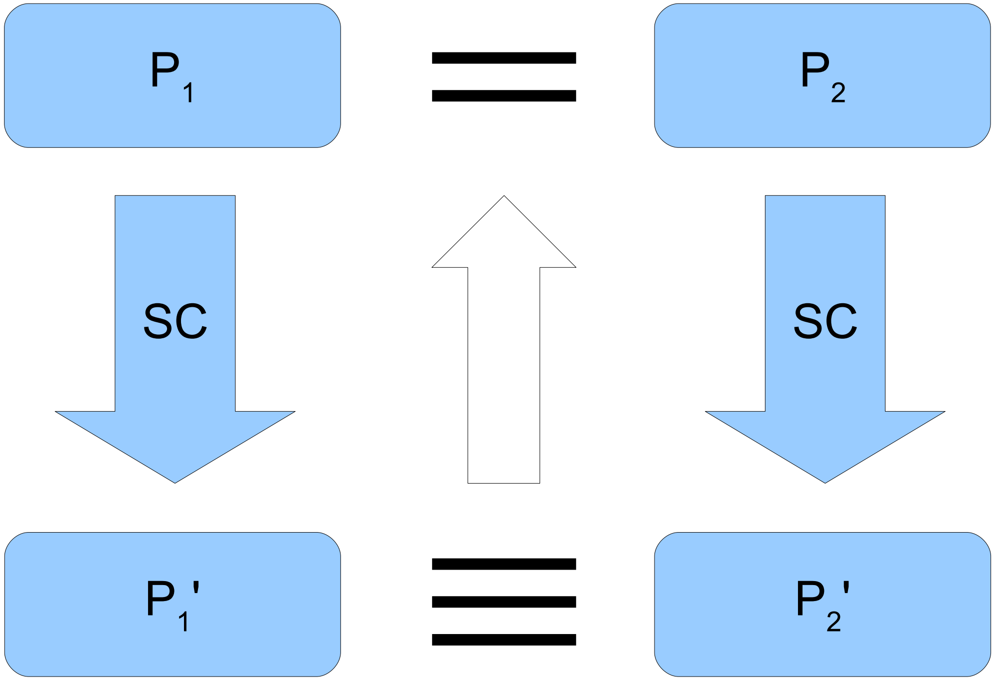
Inferring the equivalence of programs



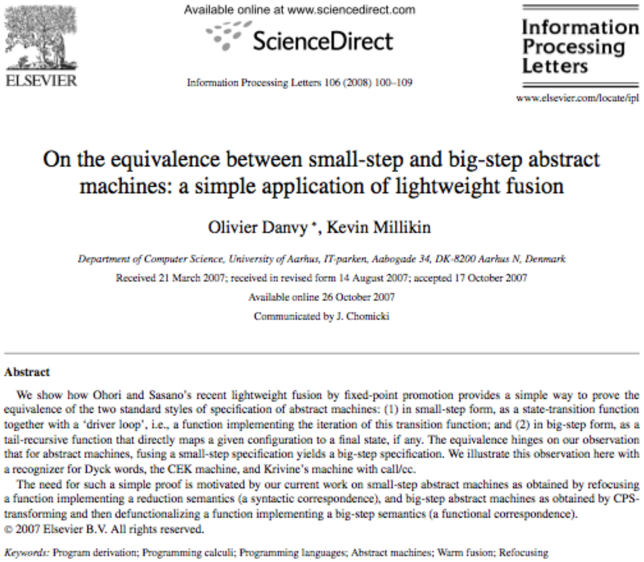
Inferring the equivalence of programs



Inferring the equivalence of programs



Example #2. Abstract machines



Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 106 (2008) 100–109
Information Processing Letters
www.elsevier.com/locate/ipl

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy *, Kevin Millikin

Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Received 21 March 2007; received in revised form 14 August 2007; accepted 17 October 2007
Available online 26 October 2007
Communicated by J. Chomicki

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop', i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

Keywords: Program derivation; Programming calculi; Programming languages; Abstract machines; Warm fusion; Refocusing

O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion / 2008

Example #2. Abstract machines

am$_1$

am$_2$

Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 106 (2008) 100–109
ELSEVIER
Information Processing Letters
www.elsevier.com/locate/ipl

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy *, Kevin Millikin

Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Received 21 March 2007; received in revised form 14 August 2007; accepted 17 October 2007
Available online 26 October 2007
Communicated by J. Chomicki

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop', i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

Keywords: Program derivation; Programming calculi; Programming languages; Abstract machines; Warm fusion; Refocusing

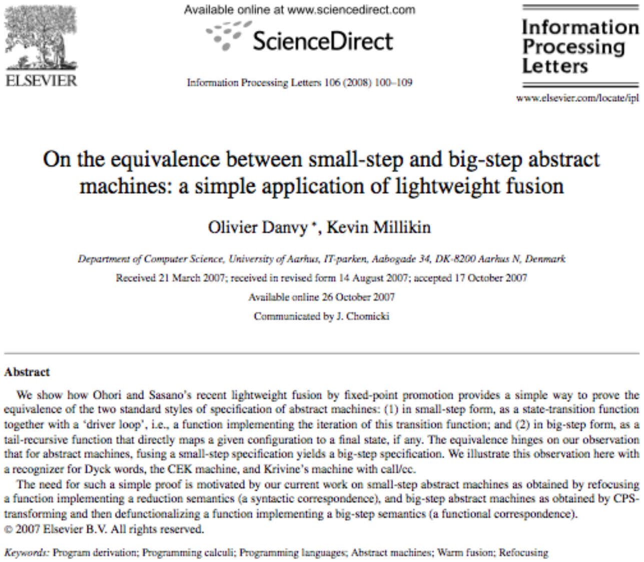
O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion / 2008

Example #2. Abstract machines

Danvy, Millikin (by hand):

am$_1$

am$_2$



Available online at www.sciencedirect.com

ScienceDirect

Information Processing Letters 106 (2008) 100–109

Information Processing Letters

www.elsevier.com/locate/ipl

ELSEVIER

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy *, Kevin Millikin

Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Received 21 March 2007; received in revised form 14 August 2007; accepted 17 October 2007

Available online 26 October 2007

Communicated by J. Chomicki

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop', i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

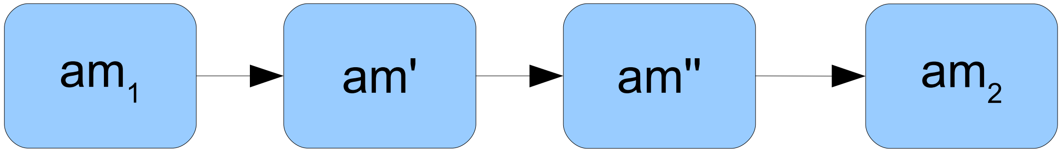
© 2007 Elsevier B.V. All rights reserved.

Keywords: Program derivation; Programming calculi; Programming languages; Abstract machines; Warm fusion; Refocusing

O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion / 2008

Example #2. Abstract machines

Danvy, Millikin (by hand):



am_1 → am' → am'' → am_2

ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Information Processing Letters 106 (2008) 100–109

Information Processing Letters

www.elsevier.com/locate/ipl

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy *, Kevin Millikin

Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Received 21 March 2007; received in revised form 14 August 2007; accepted 17 October 2007

Available online 26 October 2007

Communicated by J. Chomicki

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop', i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

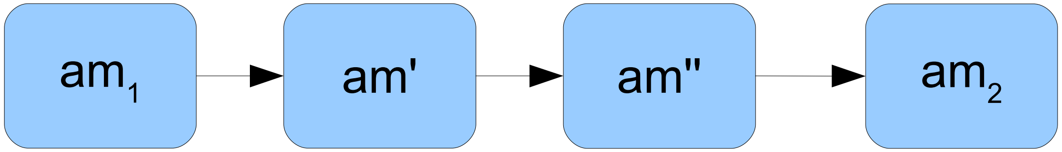
The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

Keywords: Program derivation; Programming calculi; Programming languages; Abstract machines; Warm fusion; Refocusing

O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion / 2008

Example #2. Abstract machines

Danvy, Millikin (by hand):



am$_1$

am'

am''

am$_2$

HOSC (automatically):

am$_1$

am$_2$

Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 106 (2008) 100–109
ELSEVIER
Information Processing Letters
www.elsevier.com/locate/ipl

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy *, Kevin Millikin

Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Received 21 March 2007; received in revised form 14 August 2007; accepted 17 October 2007
Available online 26 October 2007
Communicated by J. Chomicki

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop', i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

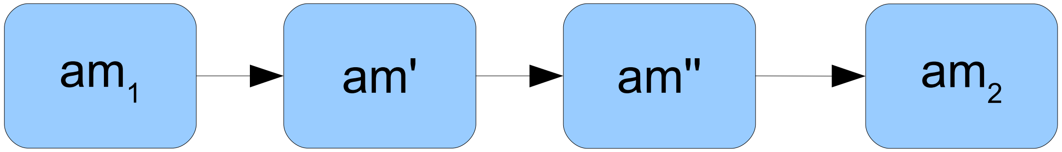
The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

Keywords: Program derivation; Programming calculi; Programming languages; Abstract machines; Warm fusion; Refocusing

O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion / 2008

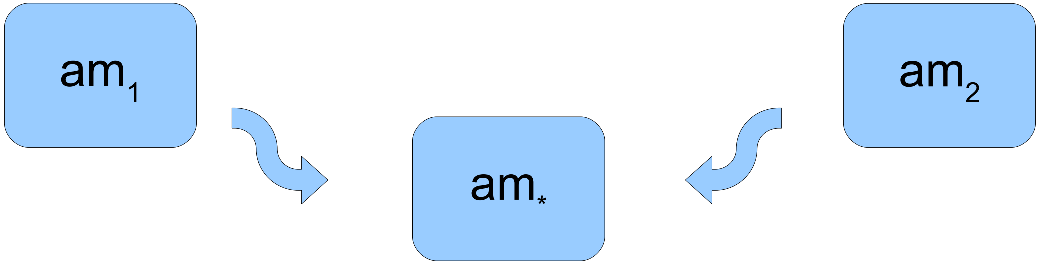
Example #2. Abstract machines

Danvy, Millikin (by hand):



am_1 → am' → am'' → am_2

HOSC (automatically):



am_1

am_2

am_*

ELSEVIER
Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 106 (2008) 100–109
Information Processing Letters
www.elsevier.com/locate/ipl

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy *, Kevin Millikin

Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Received 21 March 2007; received in revised form 14 August 2007; accepted 17 October 2007
Available online 26 October 2007
Communicated by J. Chomicki

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop', i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

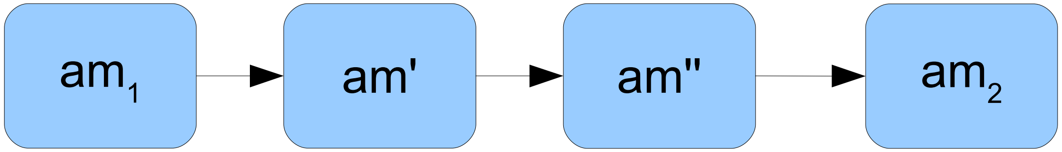
The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

Keywords: Program derivation; Programming calculi; Programming languages; Abstract machines; Warm fusion; Refocusing

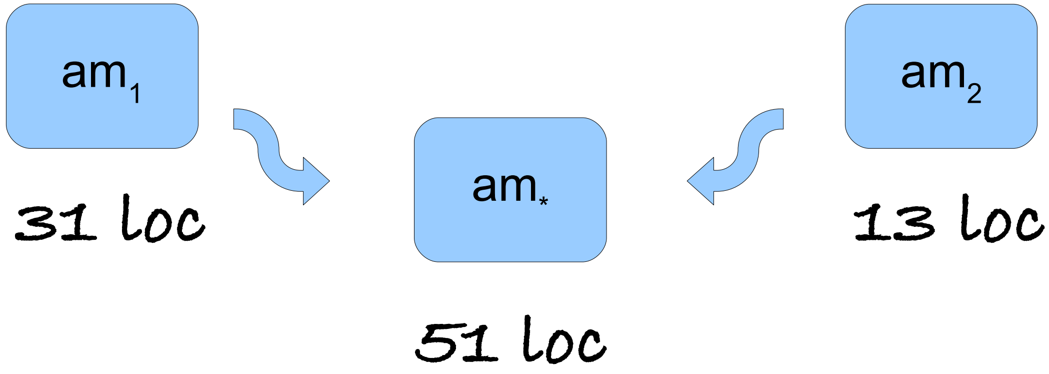
O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion / 2008

Example #2. Abstract machines

Danvy, Millikin (by hand):



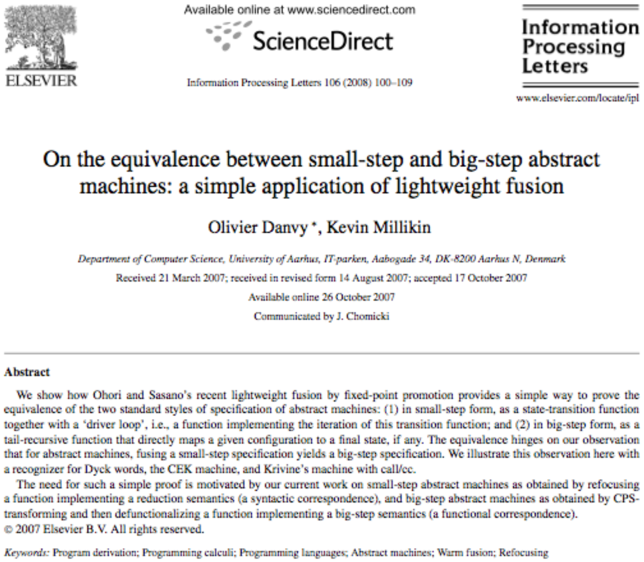
HOSC (automatically):



31 loc

13 loc

51 loc


Available online at www.sciencedirect.com

ScienceDirect

Information Processing Letters 106 (2008) 100–109

ELSEVIER

Information Processing Letters

www.elsevier.com/locate/ipl

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy *, Kevin Millikin

Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Received 21 March 2007; received in revised form 14 August 2007; accepted 17 October 2007

Available online 26 October 2007

Communicated by J. Chomicki

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop', i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

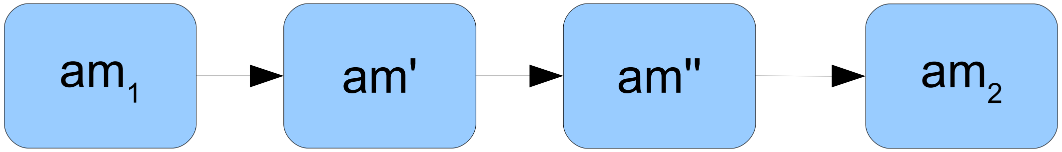
The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

Keywords: Program derivation; Programming calculi; Programming languages; Abstract machines; Warm fusion; Refocusing

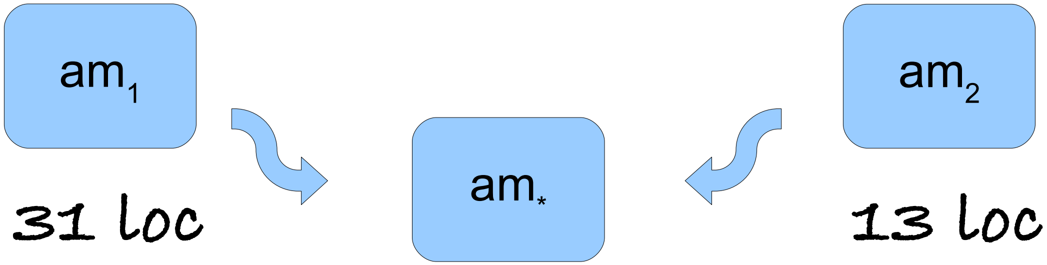
O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion / 2008

Example #2. Abstract machines

Danvy, Millikin (by hand):



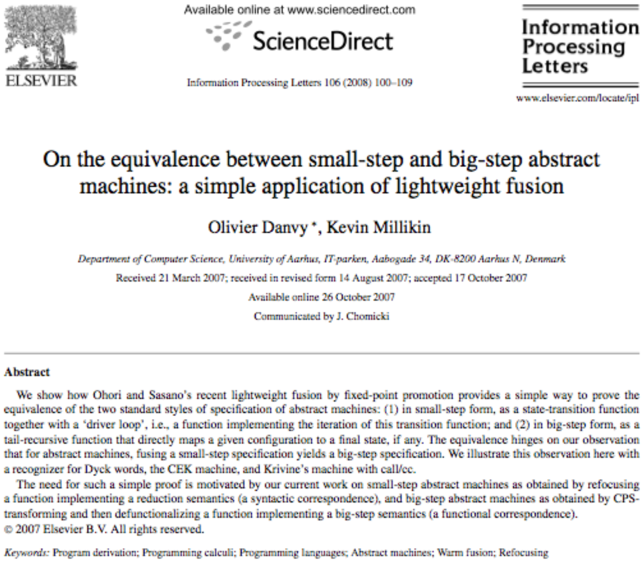
HOSC (automatically):



31 loc

13 loc

51 loc



Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 106 (2008) 100–109
ELSEVIER
Information Processing Letters
www.elsevier.com/locate/ipl

On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion

Olivier Danvy *, Kevin Millikin

Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Received 21 March 2007; received in revised form 14 August 2007; accepted 17 October 2007
Available online 26 October 2007
Communicated by J. Chomicki

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) in small-step form, as a state-transition function together with a 'driver loop', i.e., a function implementing the iteration of this transition function; and (2) in big-step form, as a tail-recursive function that directly maps a given configuration to a final state, if any. The equivalence hinges on our observation that for abstract machines, fusing a small-step specification yields a big-step specification. We illustrate this observation here with a recognizer for Dyck words, the CEK machine, and Krivine's machine with call/cc.

The need for such a simple proof is motivated by our current work on small-step abstract machines as obtained by refocusing a function implementing a reduction semantics (a syntactic correspondence), and big-step abstract machines as obtained by CPS-transforming and then defunctionalizing a function implementing a big-step semantics (a functional correspondence).

Keywords: Program derivation; Programming calculi; Programming languages; Abstract machines; Warm fusion; Refocusing

O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion / 2008

Examples online: http://hosc.appspot.com

Can we reuse this normalization property?

Can we reuse this normalization property?

Self-application???

Approaches to self-application

- Futamura projections
  - sc(int, prog) = prog'
  - sc(sc, int) = compiler
  - sc(sc, sc) = compiler generator

Approaches to self-application

- Futamura projections
  - sc(int, prog) = prog'
  - sc(sc, int) = compiler
  - sc(sc, sc) = compiler generator

The most popular (old) idea

Approaches to self-application

- Futamura projections
  - sc(int, prog) = prog'
  - sc(sc, int) = compiler
  - sc(sc, sc) = compiler generator

The most popular (old) idea

- Distillation
- Two-level supercompilation
- ...

Approaches to self-application

- Futamura projections
  - sc(int, prog) = prog'
  - sc(sc, int) = compiler
  - sc(sc, sc) = compiler generator

The most popular (old) idea

- Distillation
- Two-level supercompilation
- ...

Rather new approaches

Approaches to self-application

- Futamura projections
  - sc(int, prog) = prog'
  - sc(sc, int) = compiler
  - sc(sc, sc) = compiler generator

The most popular (old) idea

- Distillation
- Two-level supercompilation
- ...

Rather new approaches

The Plan

- Supercompilation in a nutshell
- Optimization vs Analysis
- Analyzing supercompilation (HOSC)
- Two-level supercompilation
- Multi-result supercompilation (MRSC)
- Finding a minimal proof by multi-result supercompilation
- On metasystem transitions

Two-Level Supercompilation

The problem:

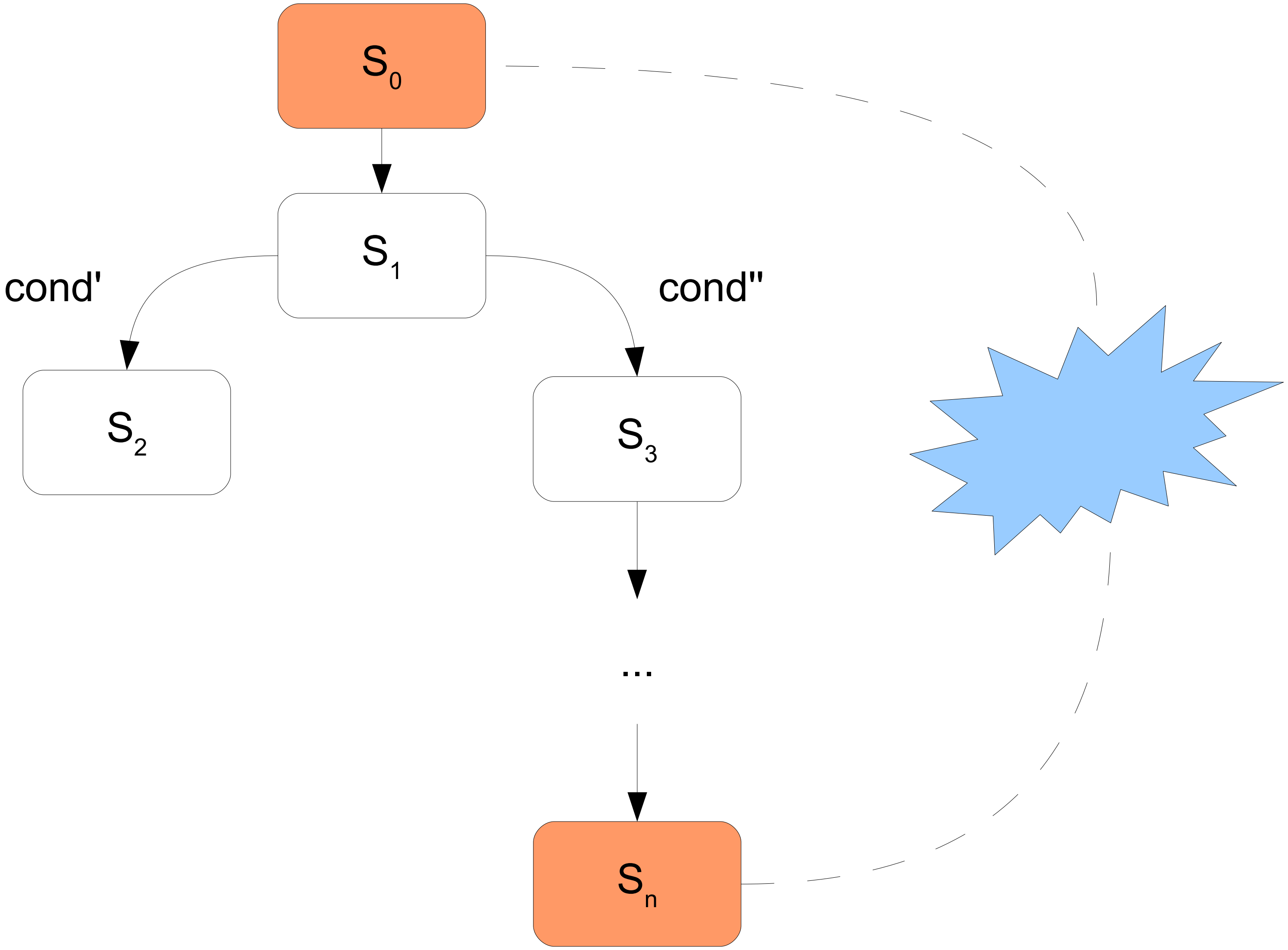
- When whistle blows, we perform generalization.
- Generalization is evil, since it result in loss of information.

The idea:

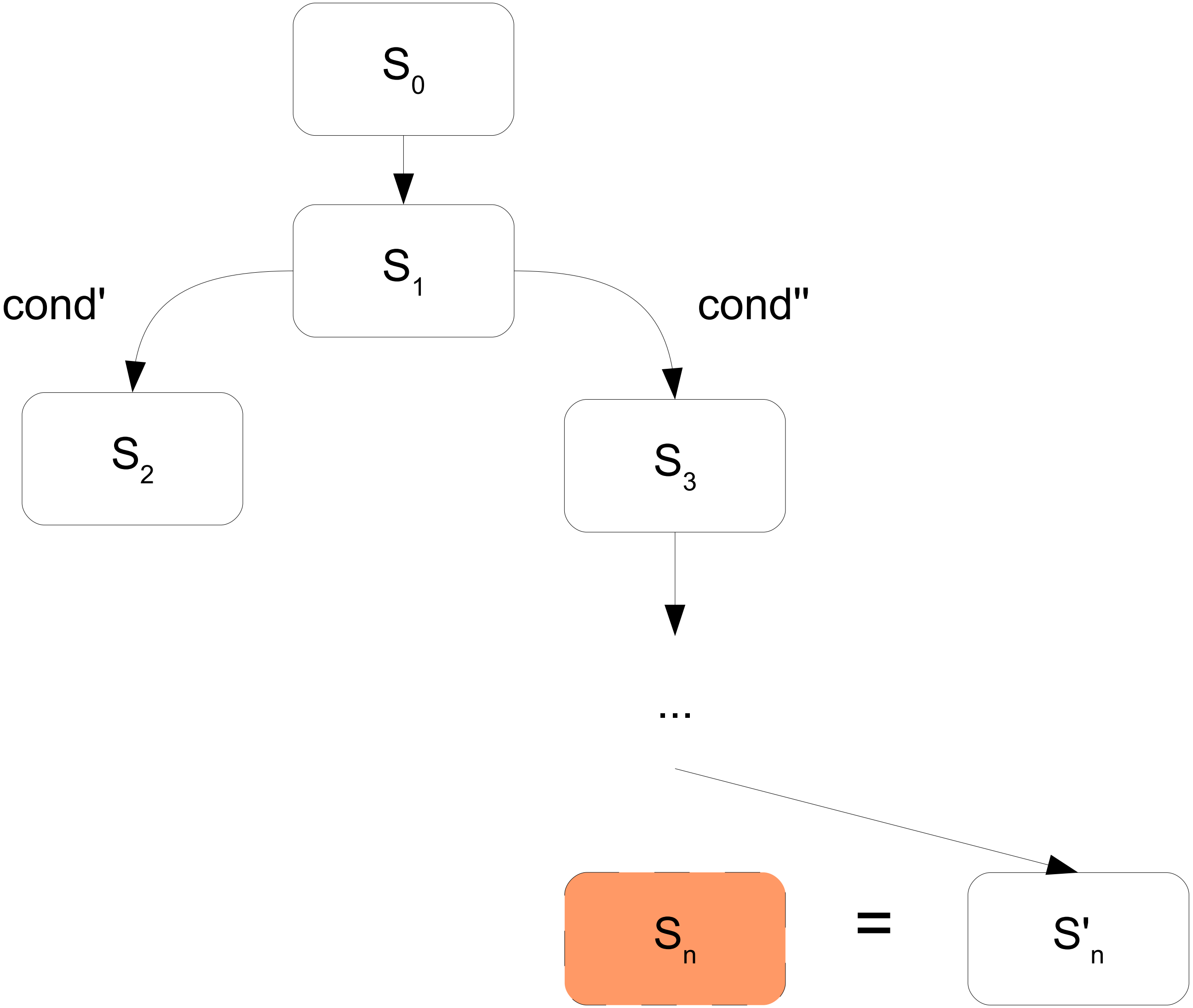
- Escape from whistle

I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. META-2010

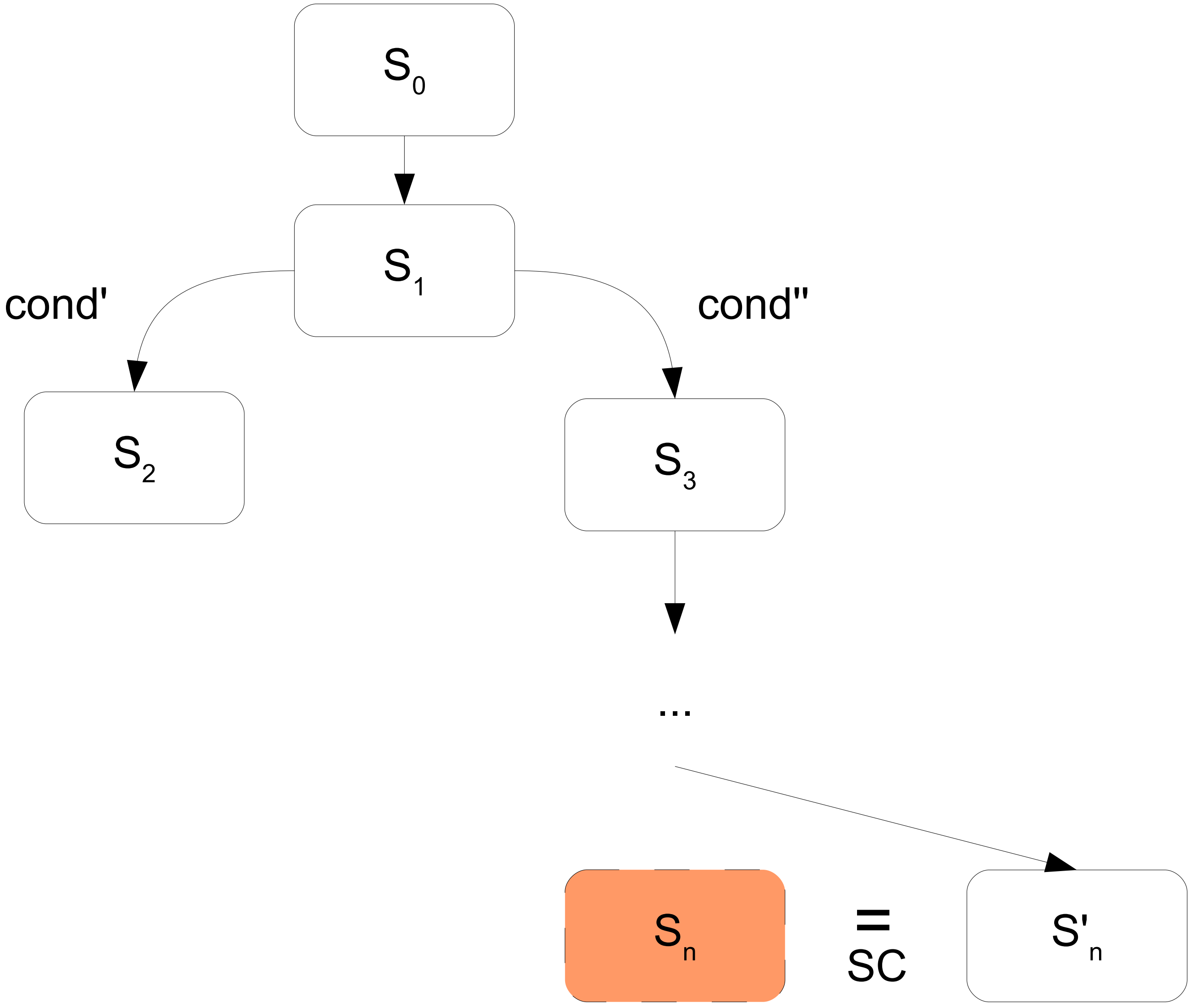
Escape from Whistle



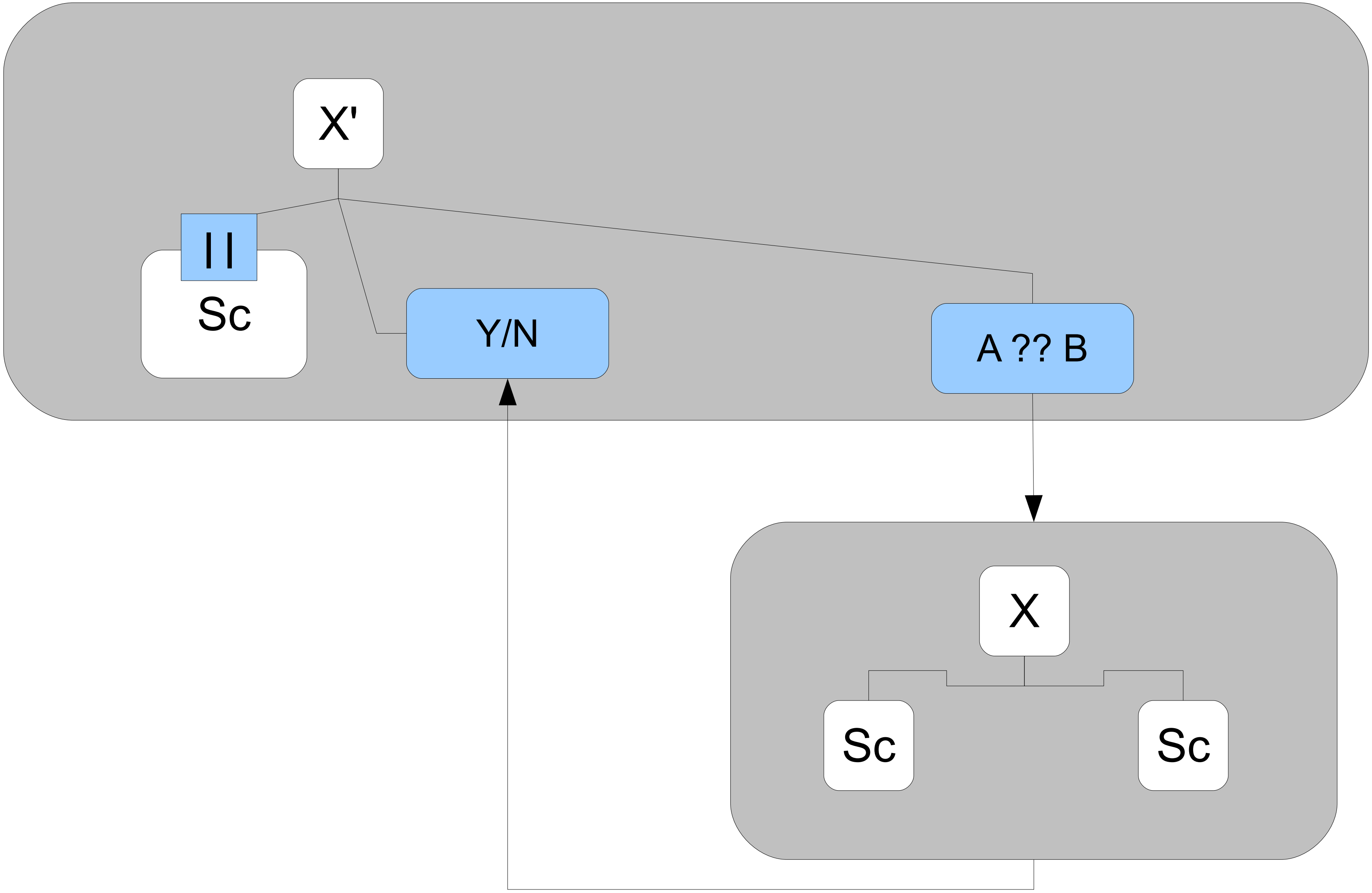
Escape from Whistle



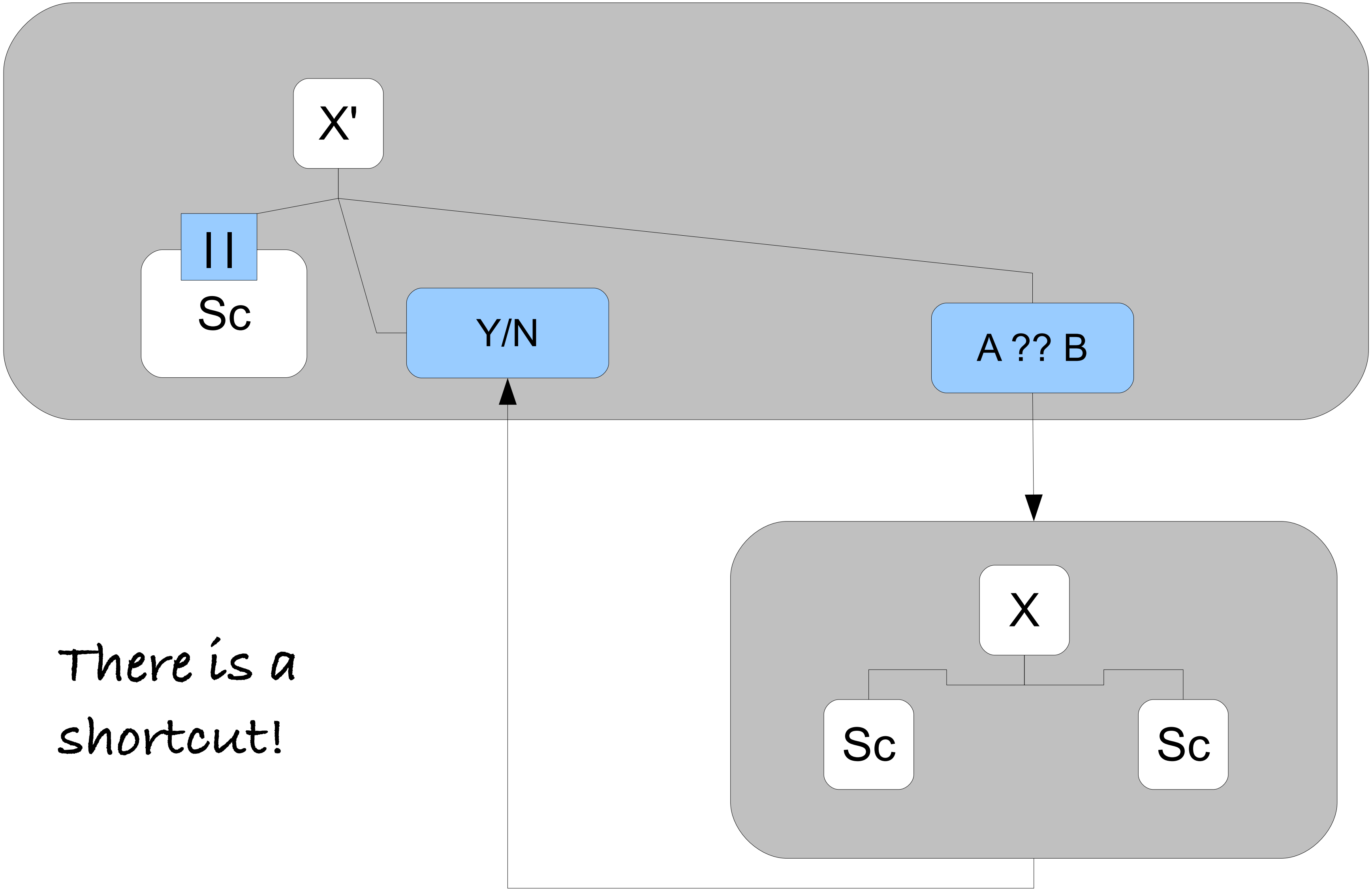
Escape from Whistle



Two-Level Supercompilation

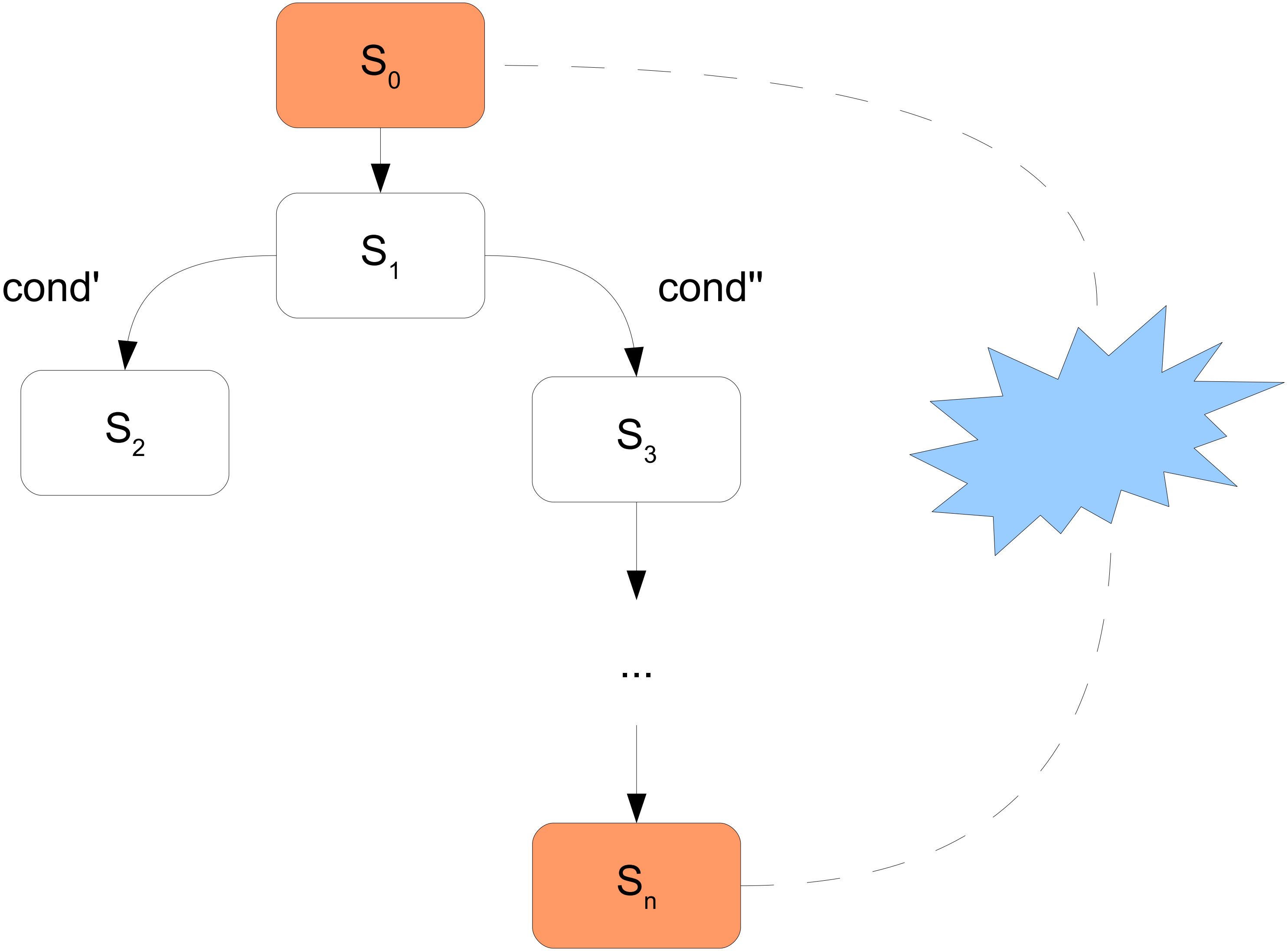


Two-Level Supercompilation

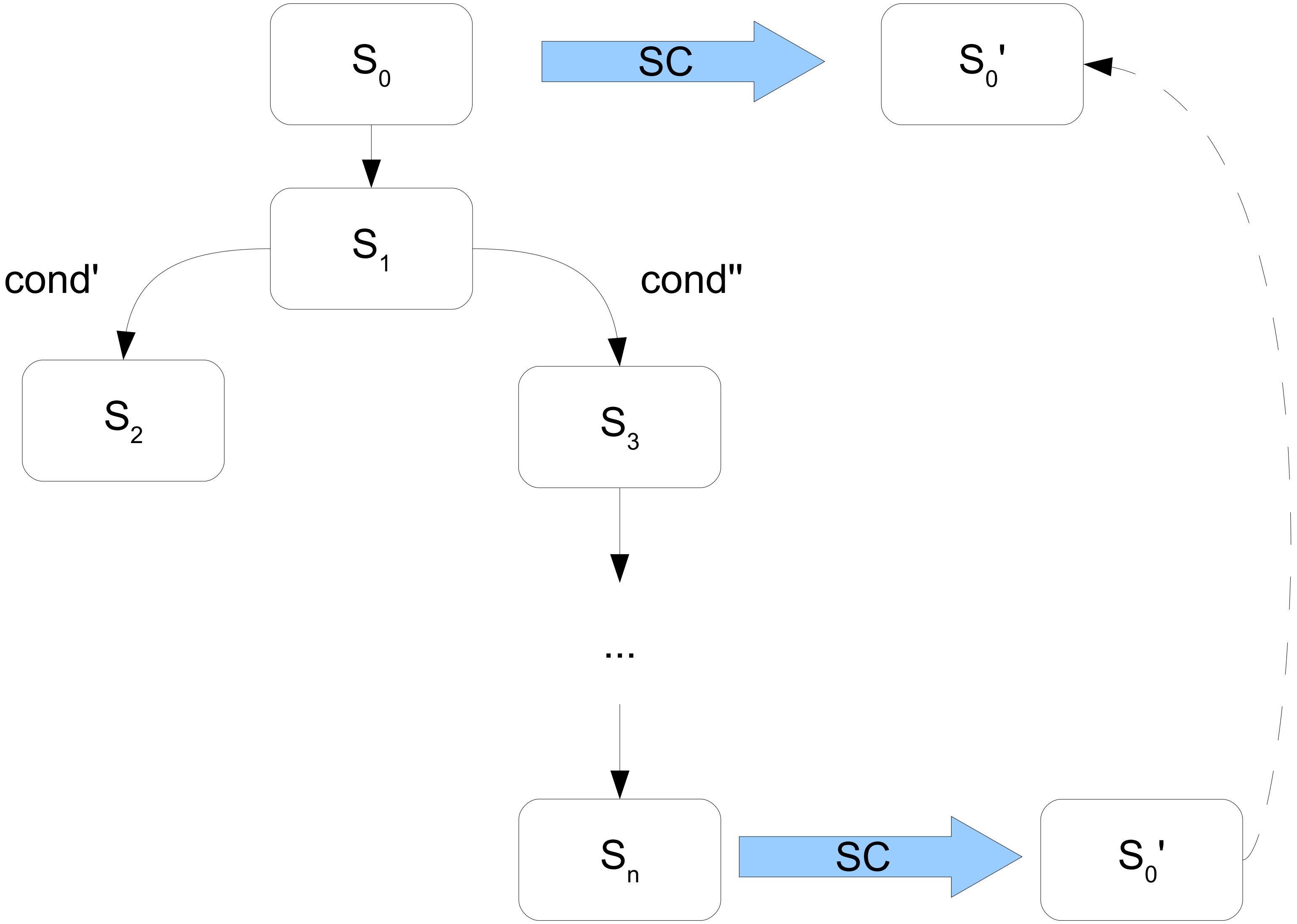


There is a shortcut!

Shortcut Two-Level supercompilation



Shortcut Two-Level supercompilation



I. Klyuchnikov. Towards effective two-level supercompilation. KIAM Preprint #81. 2010

Example #3. Even or odd

```
data Bool = True | False;

data Nat = Z | S Nat;

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; };
```

<u>`or (even m) (odd m)`</u>

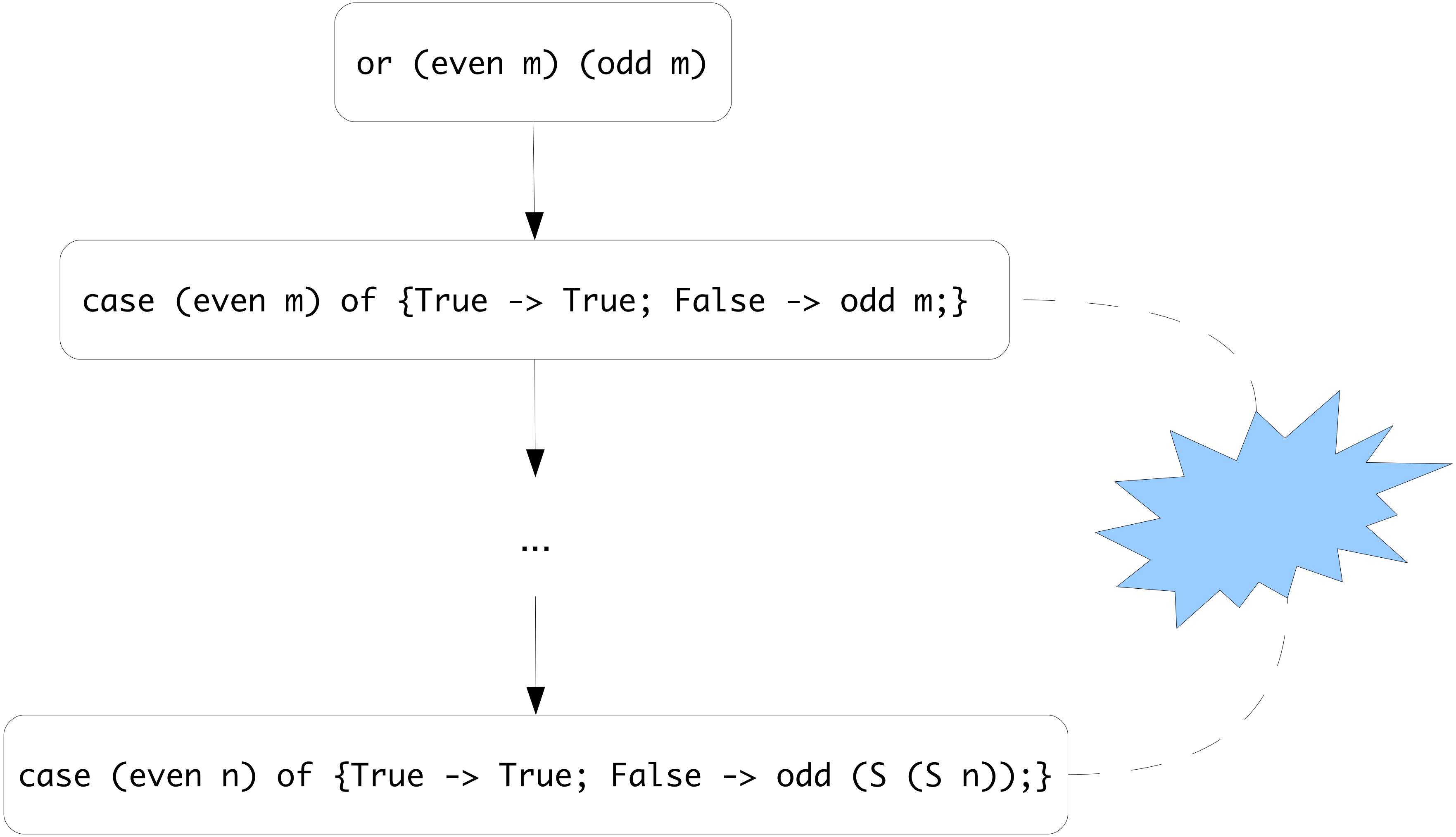
The output:

```
letrec f = \w ->

  case w of { Z -> True; S x -> case x  of { Z  -> True; S z -> f z; };}

in f m
```

Example #3. Even or odd



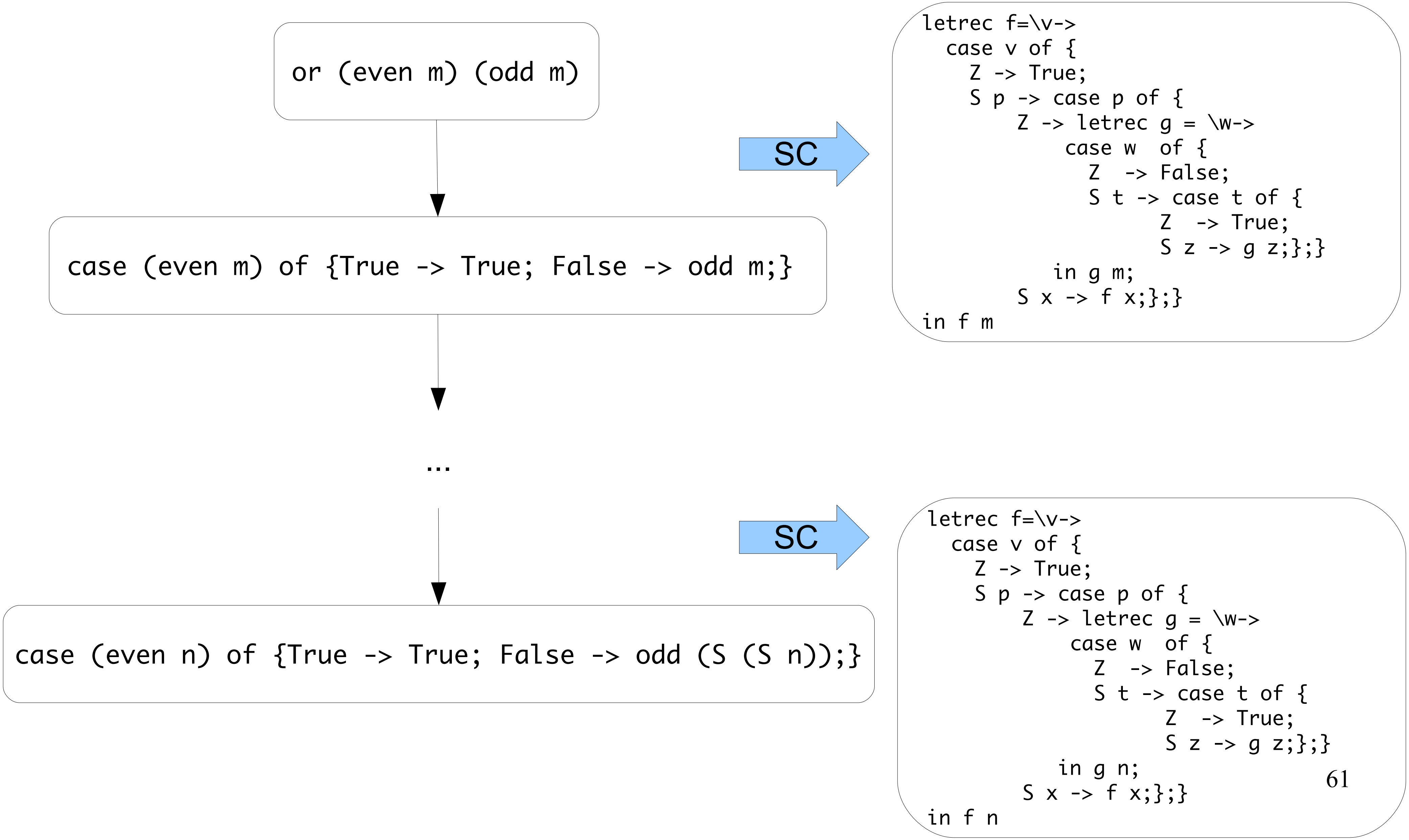
`or (even m) (odd m)`

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

…

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

Example #3. Even or odd



```
or (even m) (odd m)
```

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

...

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

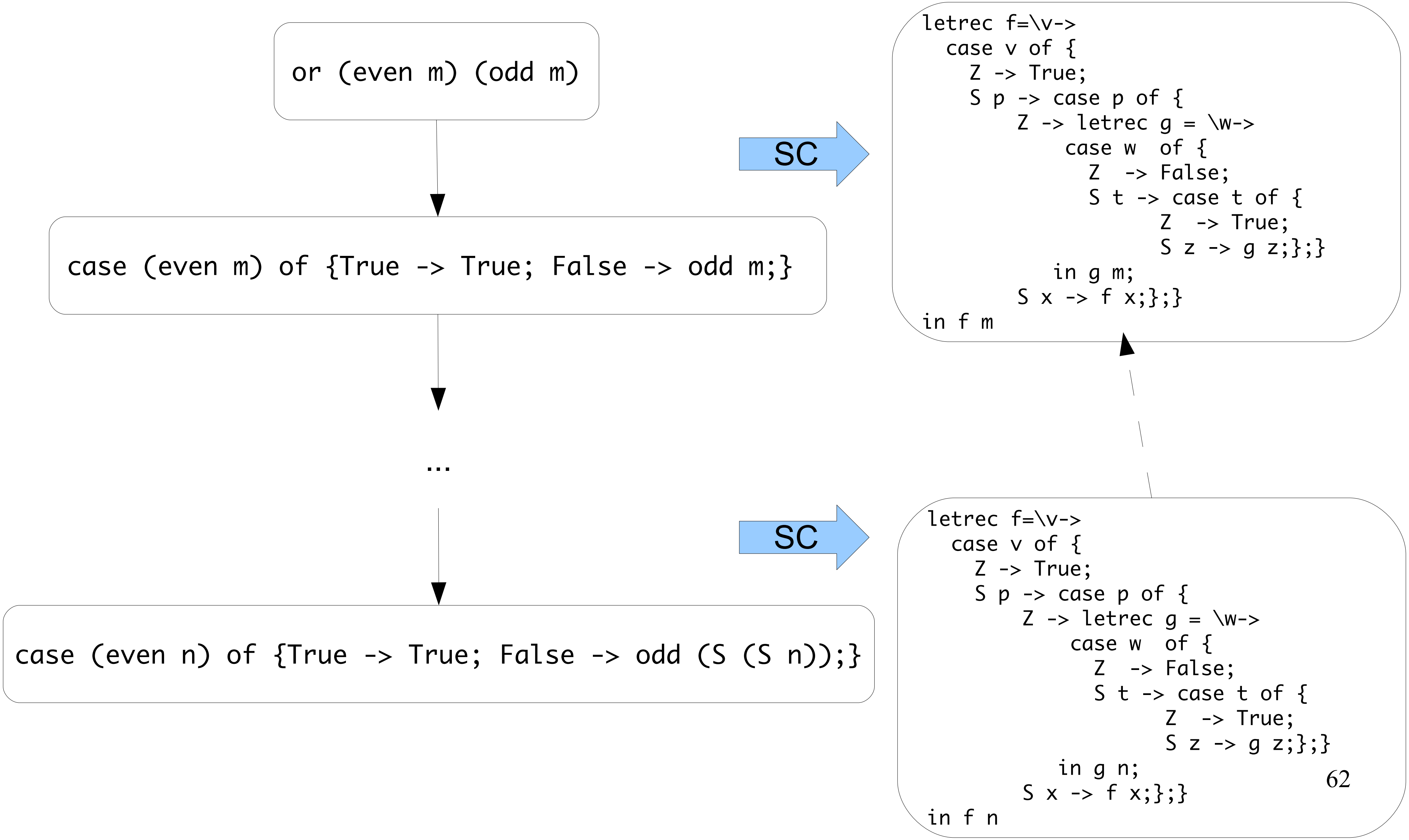
SC

```
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
```

SC

```
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
```

Example #3. Even or odd



`or (even m) (odd m)`

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

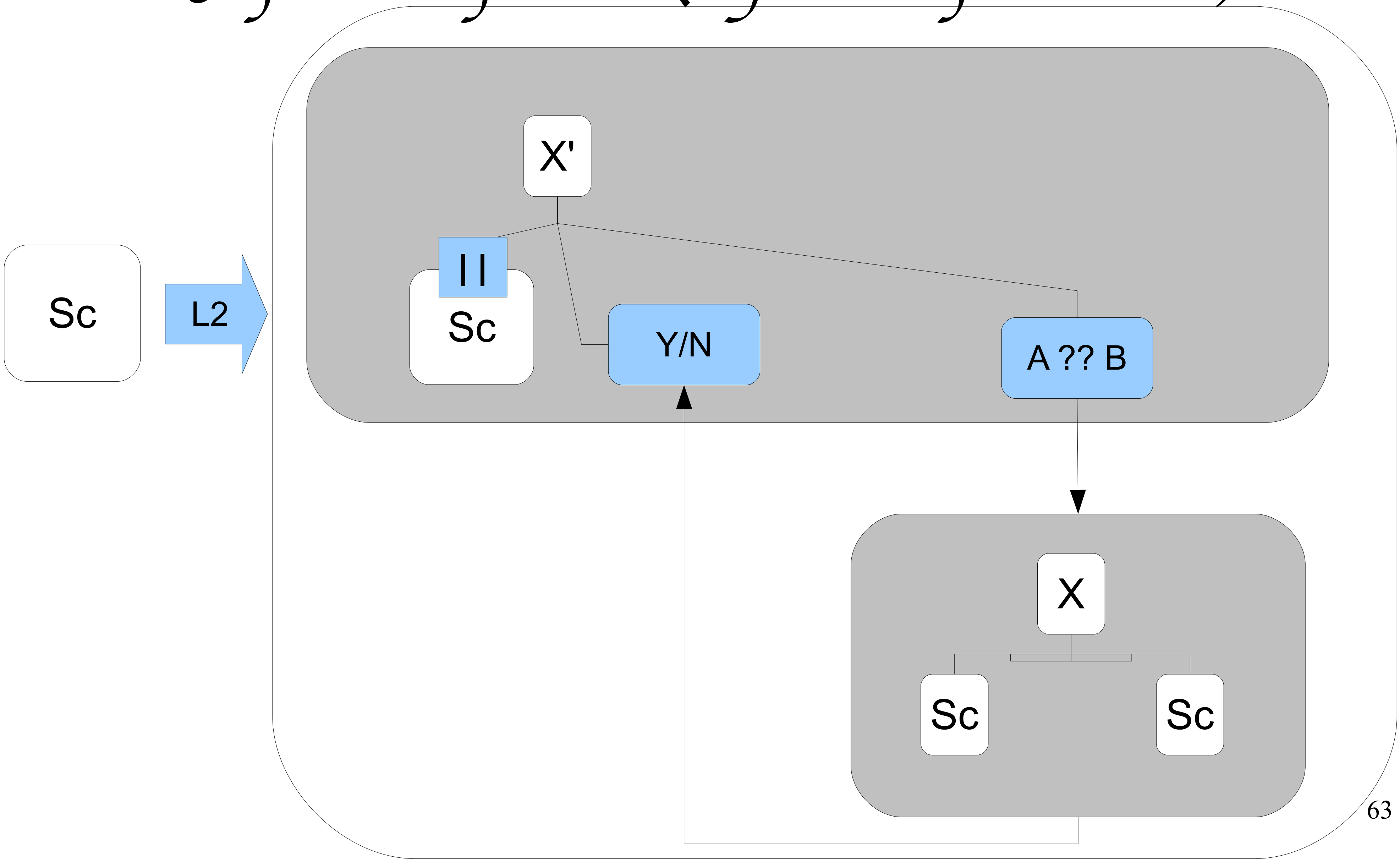
…

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

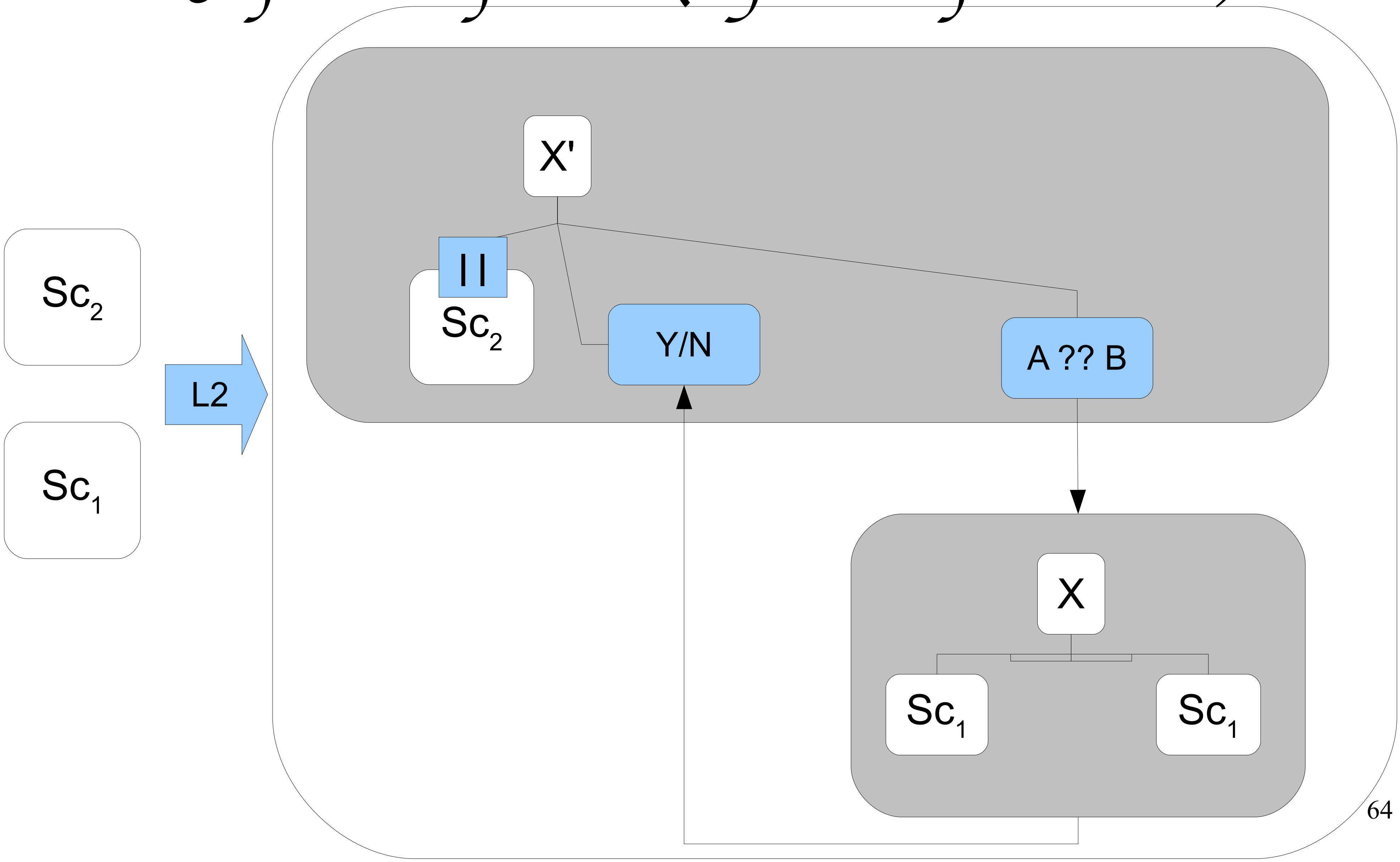
```
letrec f=\v->
 case v of {
  Z -> True;
  S p -> case p of {
    Z -> letrec g = \w->
      case w  of {
       Z  -> False;
       S t -> case t of {
          Z  -> True;
          S z -> g z;};}
     in g m;
    S x -> f x;};}
in f m
```

```
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
```

Making supercompilers from supercompilers (by multiplication)



Making supercompilers from supercompilers (by multiplication)



Is It Worth to Do?

We make a two-level supercompiler from *two different* supercompilers.

Is It Worth to Do?

We make a two-level supercompiler from *two different* supercompilers.

Does this approach make difference?

Is It Worth to Do?

We make a two-level supercompiler from *two different* supercompilers.

Does this approach make difference?
(We could use more powerful low-level supercompilers).

+/- is a feature

The stuff for experiments:
SC_{---}, SC_{+--}, SC_{-+-}, ... - 8 supercompilers

I. Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting

It Is Worth to Do!

The task: grammar transformation

```
doubleA = ϵ | a doubleA a
doubleA = ϵ | a a doubleA
```

I. Klyuchnikov. Towards effective two-level supercompilation. KIAM Preprint #81. 2010

It Is Worth to Do!

The task: grammar transformation

```
doubleA = ϵ | a doubleA a
doubleA = ϵ | a a doubleA
```

L2(SC_{---},SC_{---}), L2(SC_{+--},SC_{+--}), L2(SC_{-+-},SC_{-+-}),
L2(SC_{--+},SC_{--+}), ... FAILURE

I. Klyuchnikov. Towards effective two-level supercompilation. KIAM Preprint #81. 2010

It Is Worth to Do!

The task: grammar transformation

```
doubleA = ϵ | a doubleA a
doubleA = ϵ | a a doubleA
```

L2(SC_{---}, SC_{---}), L2(SC_{+--}, SC_{+--}), L2(SC_{-+-}, SC_{-+-}),
L2(SC_{--+}, SC_{--+}), ... FAILURE

L2(SC_{++-}, SC_{+--}), ... SUCCESS

I. Klyuchnikov. Towards effective two-level supercompilation. KIAM Preprint #81. 2010

It Is Worth to Do!

The task: grammar transformation

```
doubleA = ϵ | a doubleA a
doubleA = ϵ | a a doubleA
```

L2(SC_{---}, SC_{---}), L2(SC_{+--}, SC_{+--}), L2(SC_{-+-}, SC_{-+-}), L2(SC_{--+}, SC_{--+}), ... FAILURE

L2(SC_{++-}, SC_{+--}), ... SUCCESS

Interesting Pattern:
L2(Sc_2, Sc_1) – Sc_2 should be a bit smarter than Sc_1 (A managing person should be **a bit** clever than a person being managed)

I. Klyuchnikov. Towards effective two-level supercompilation. KIAM Preprint #81. 2010

V. Turchin. The phenomenon of Science

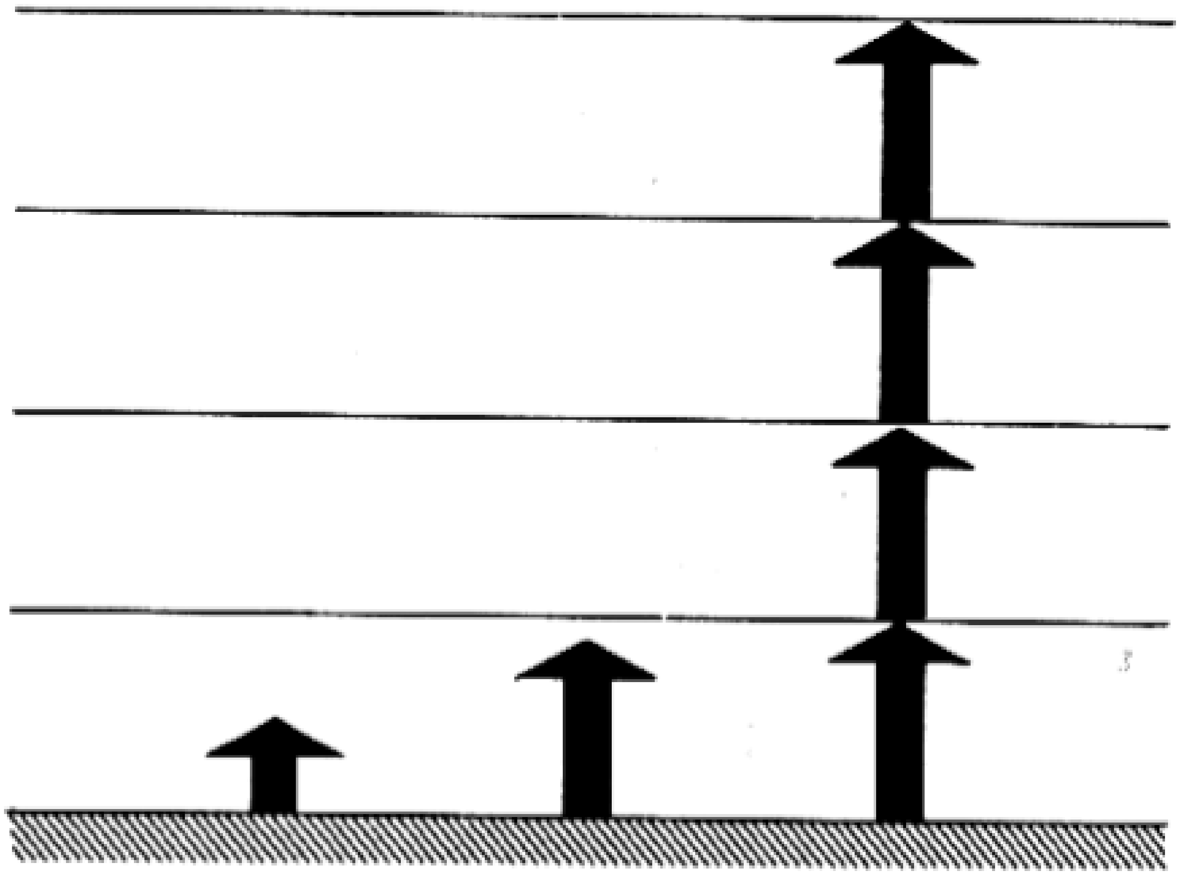
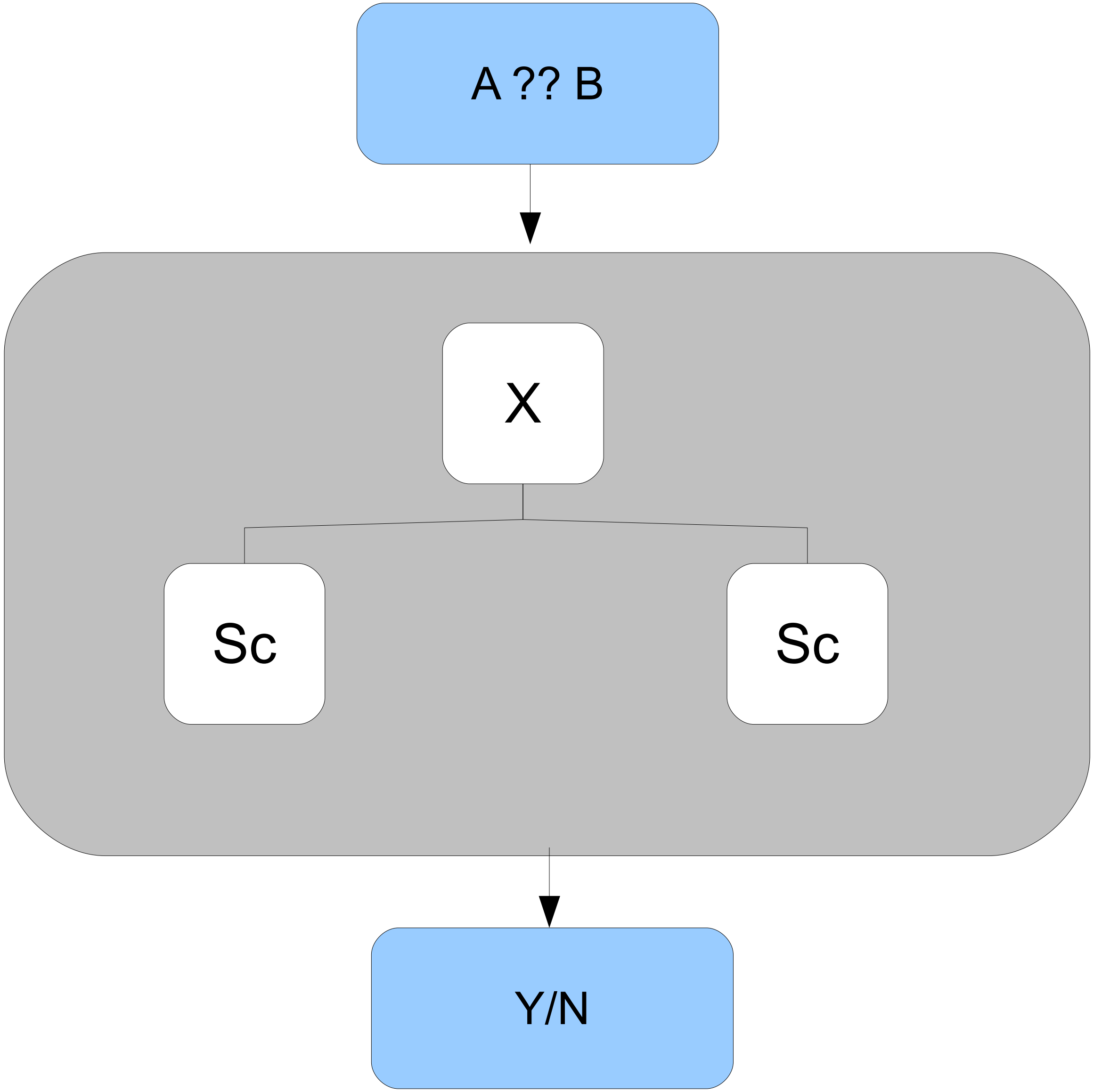
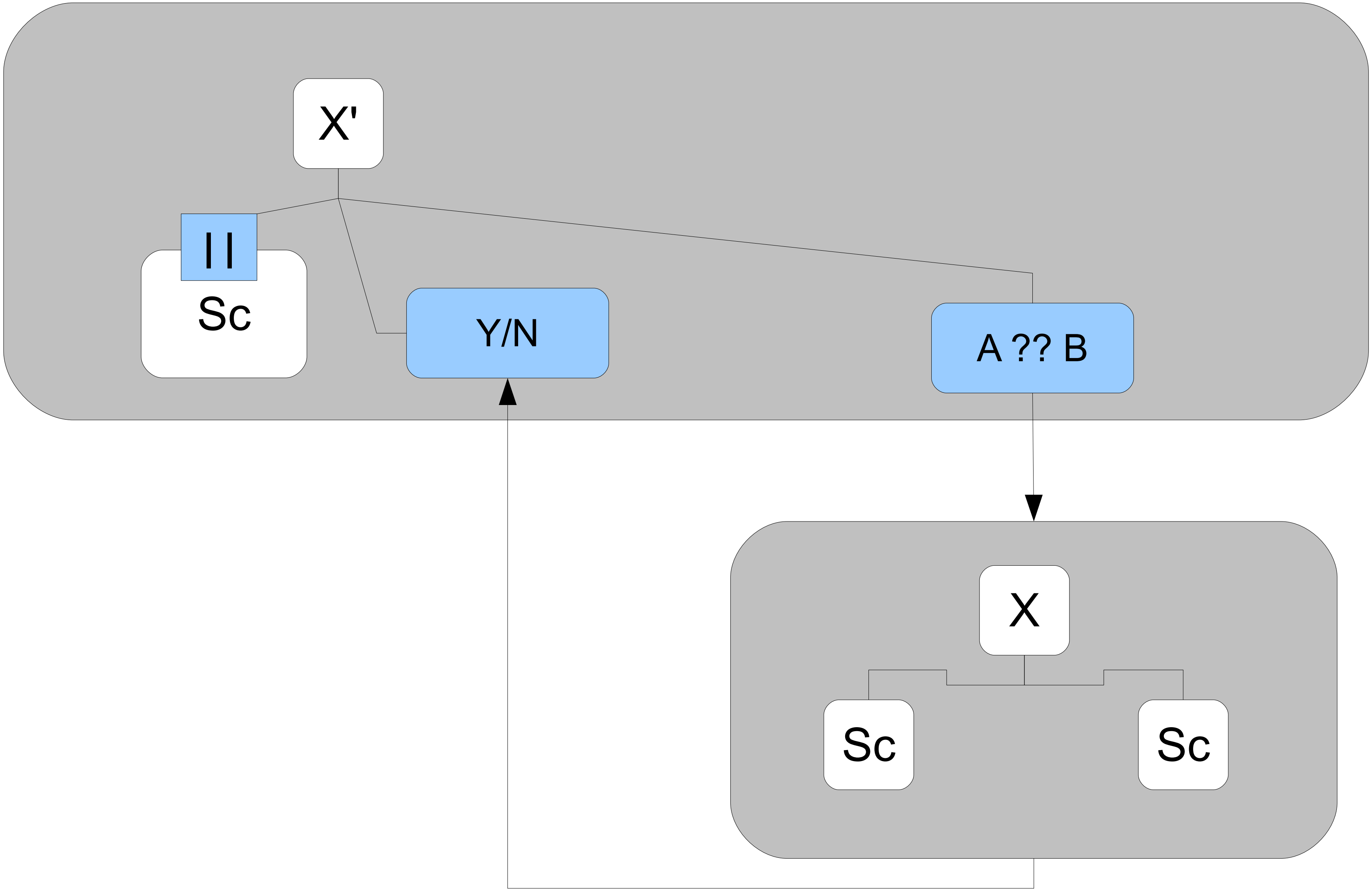


Figure 5.1. The stairway effect.

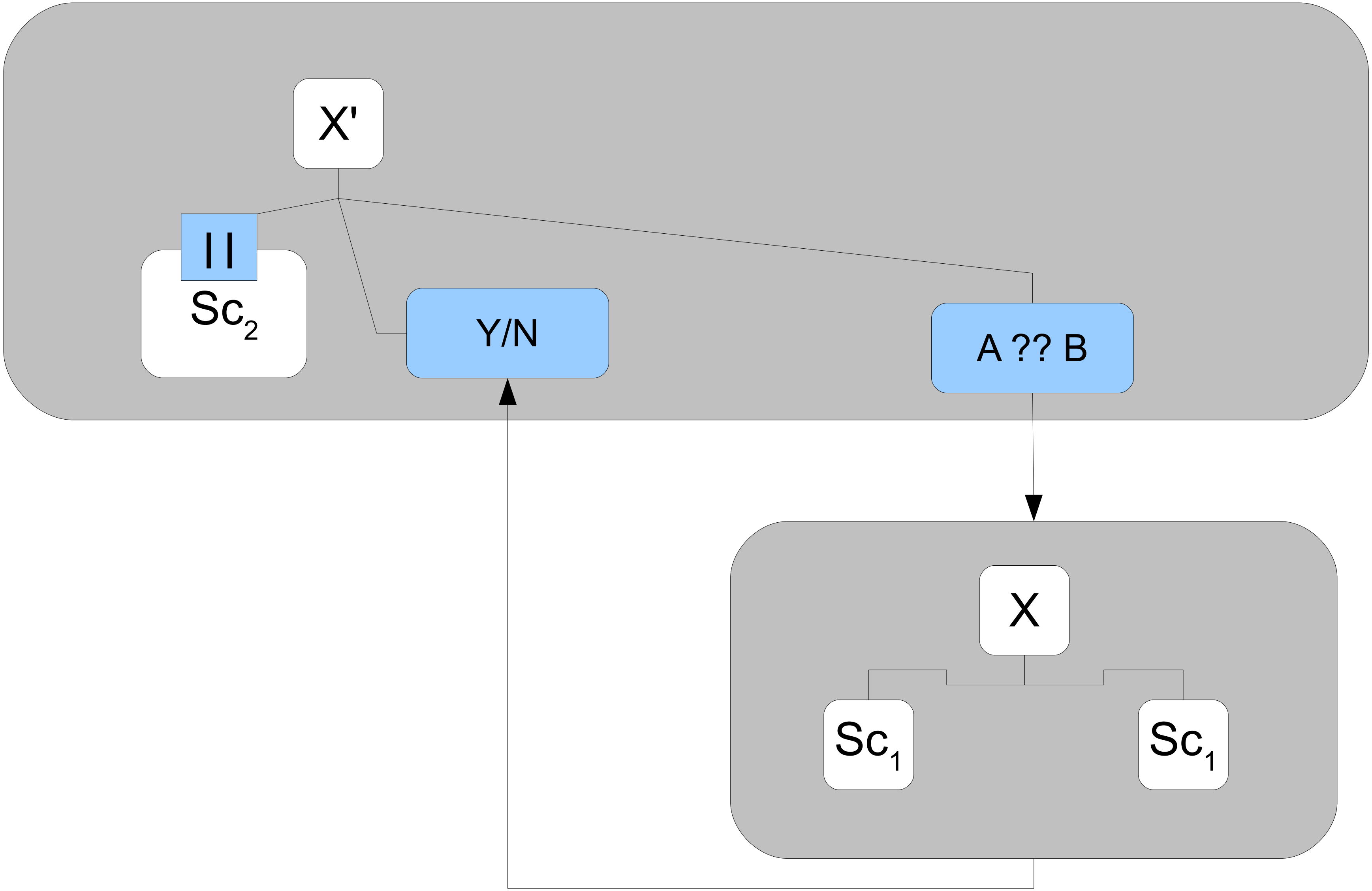
Step #1: 2 Instances of a Supercompiler



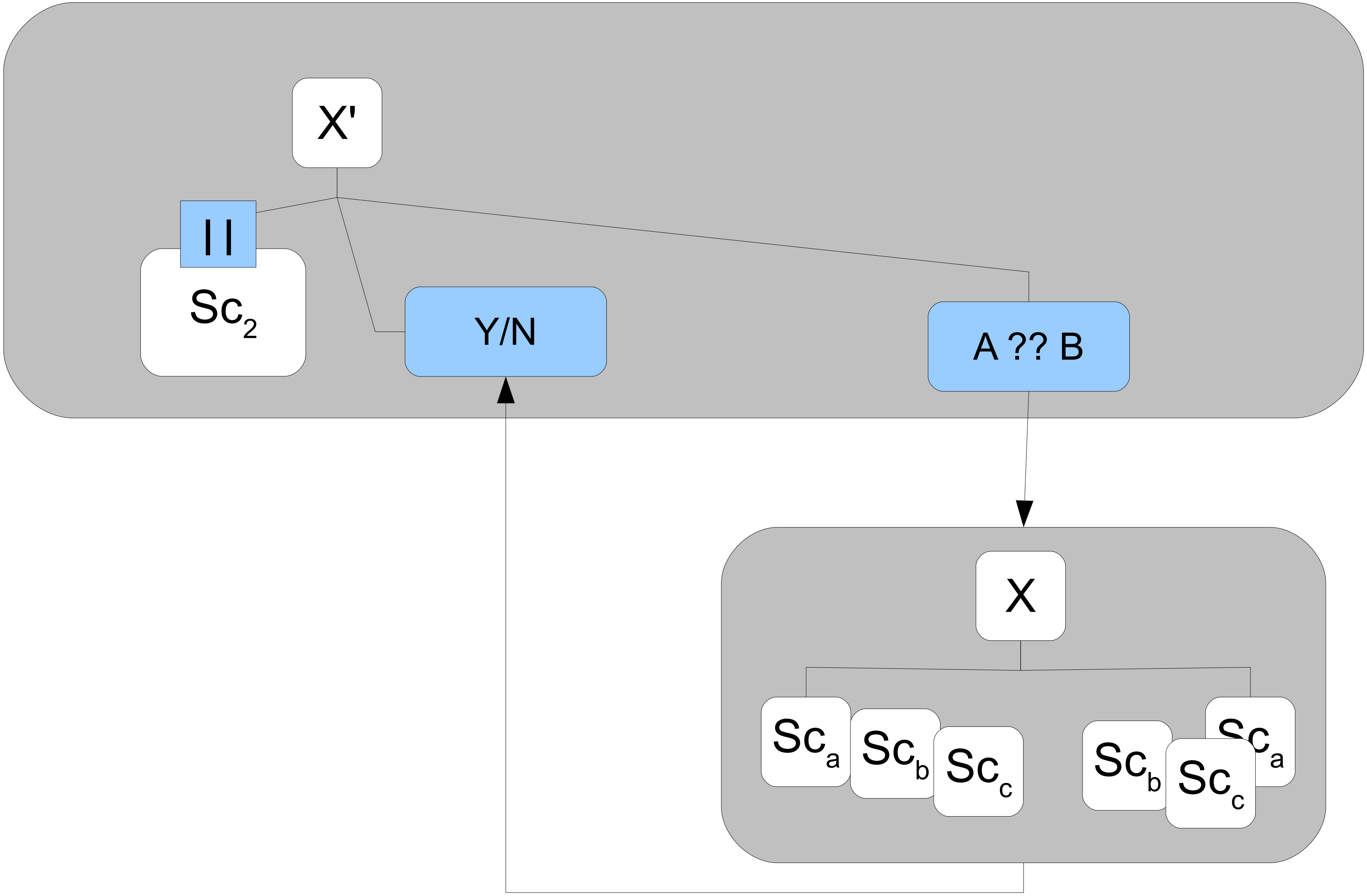
Step #2: 3 Instances of a Supercompiler



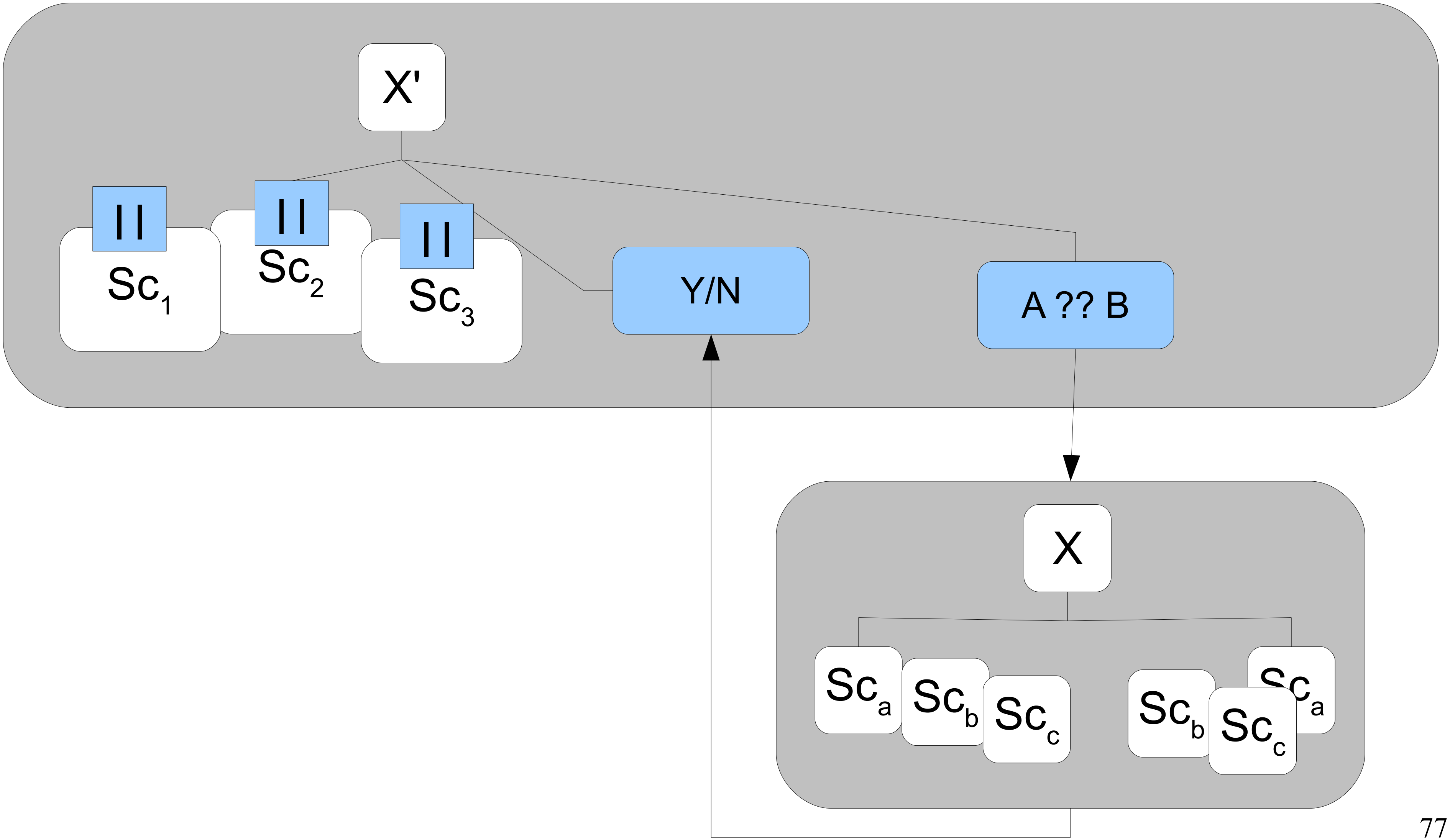
Step #3: Combining 2 Supercompilers



Step #4: Combining Many Supercompilers



Step #5?



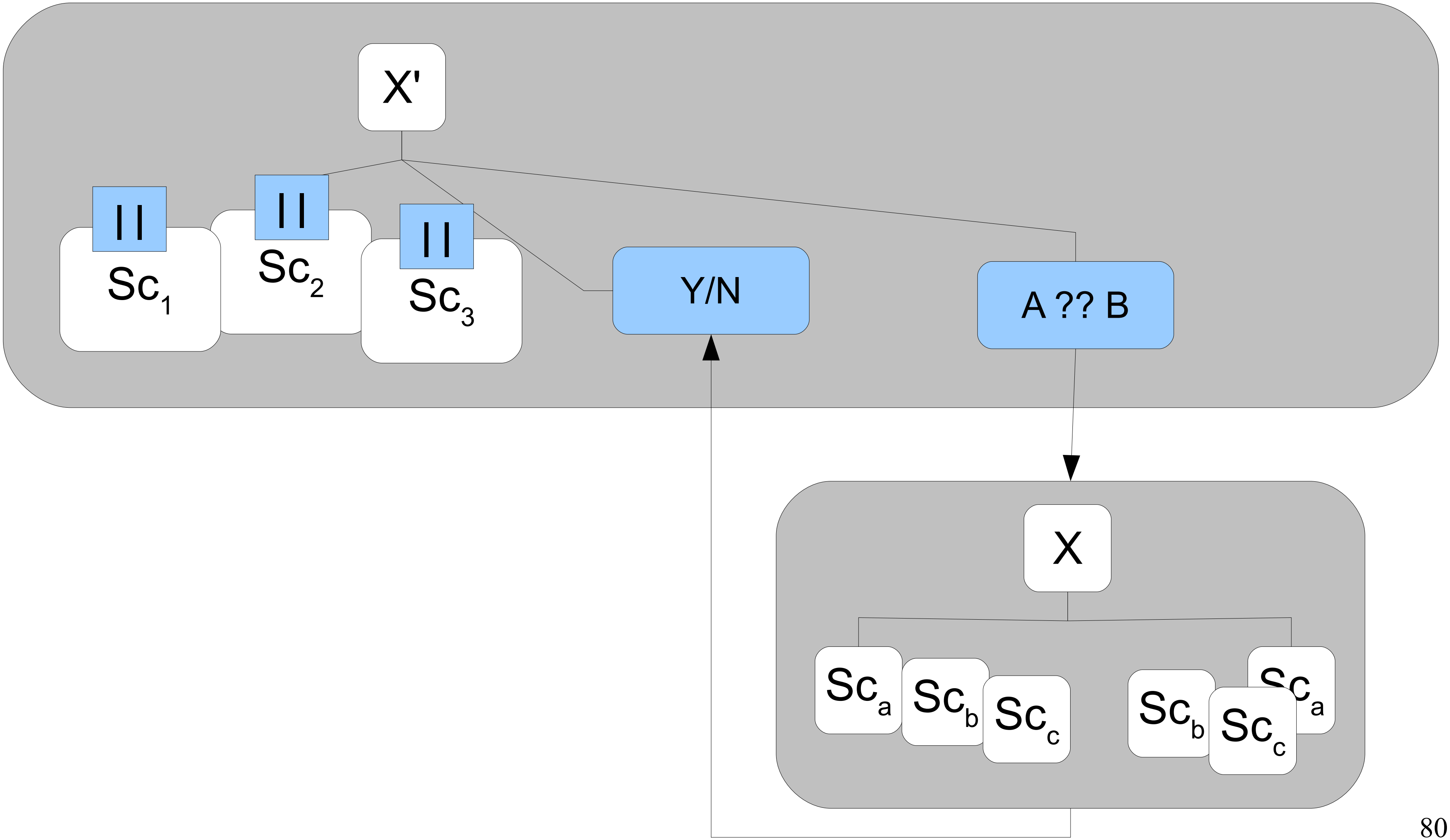
STOP

WE ARE LOOSING CONTROL

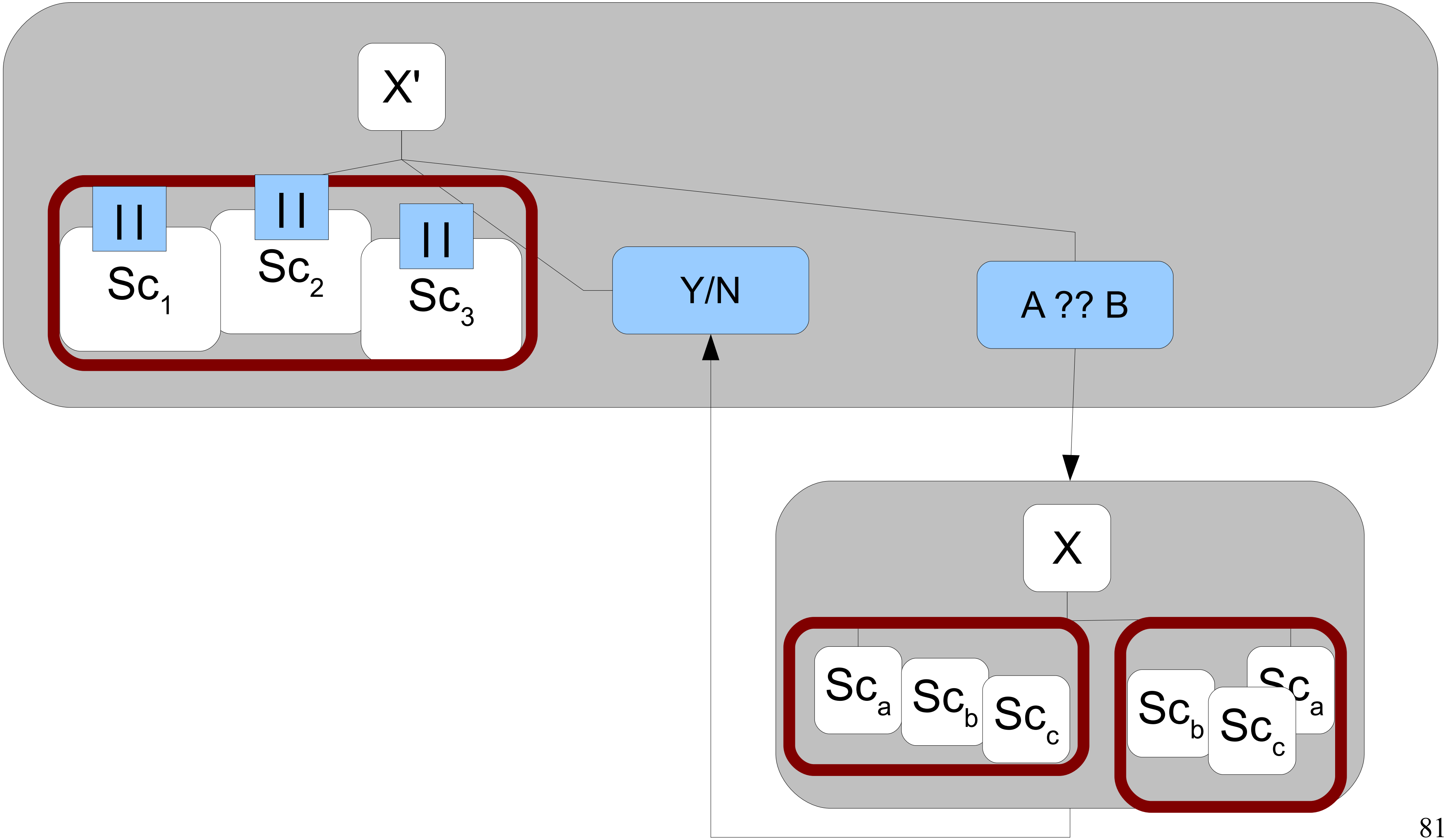
The Plan

- Supercompilation in a nutshell
- Optimization vs Analysis
- Analyzing supercompilation (HOSC)
- Two-level supercompilation
- Multi-result supercompilation (MRSC)
- Finding a minimal proof by multi-result supercompilation
- On metasystem transitions

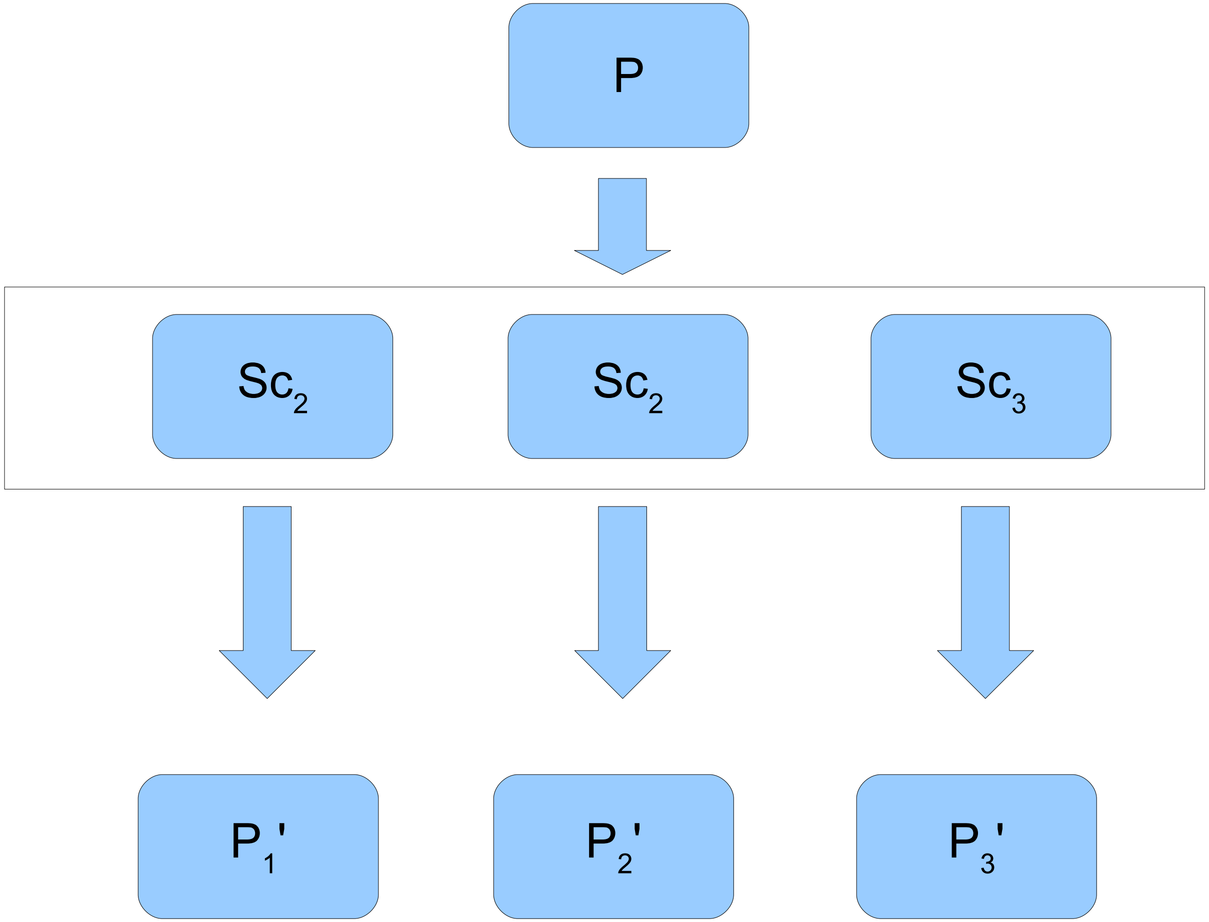
There Is One More Elementary Operation Here:



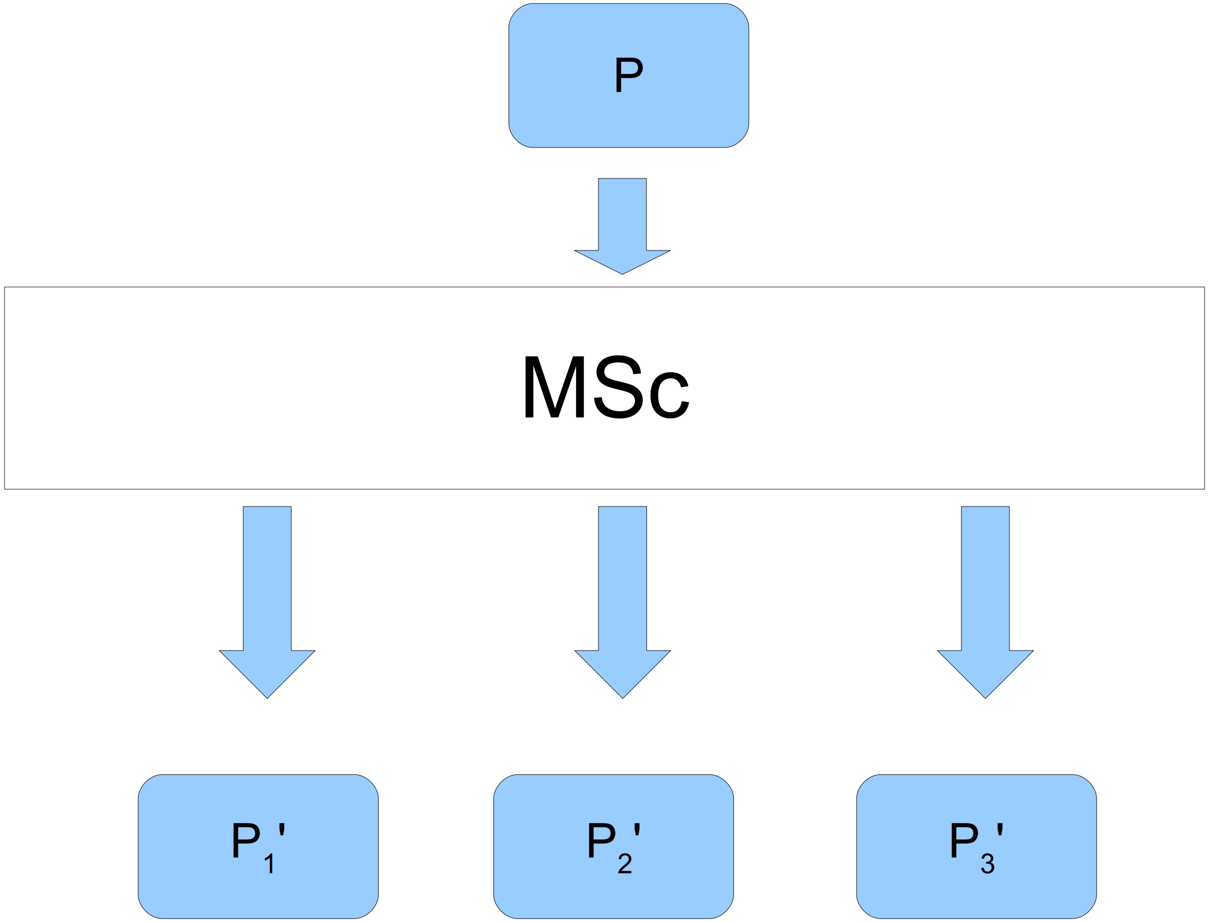
There Is One More Elementary Operation Here: Multiplication of Supercompilers



Let's Treat Many Supercompilers as One Multi-Result Supercompiler



Let's Treat Many Supercompilers as One Multi-Result Supercompiler



P

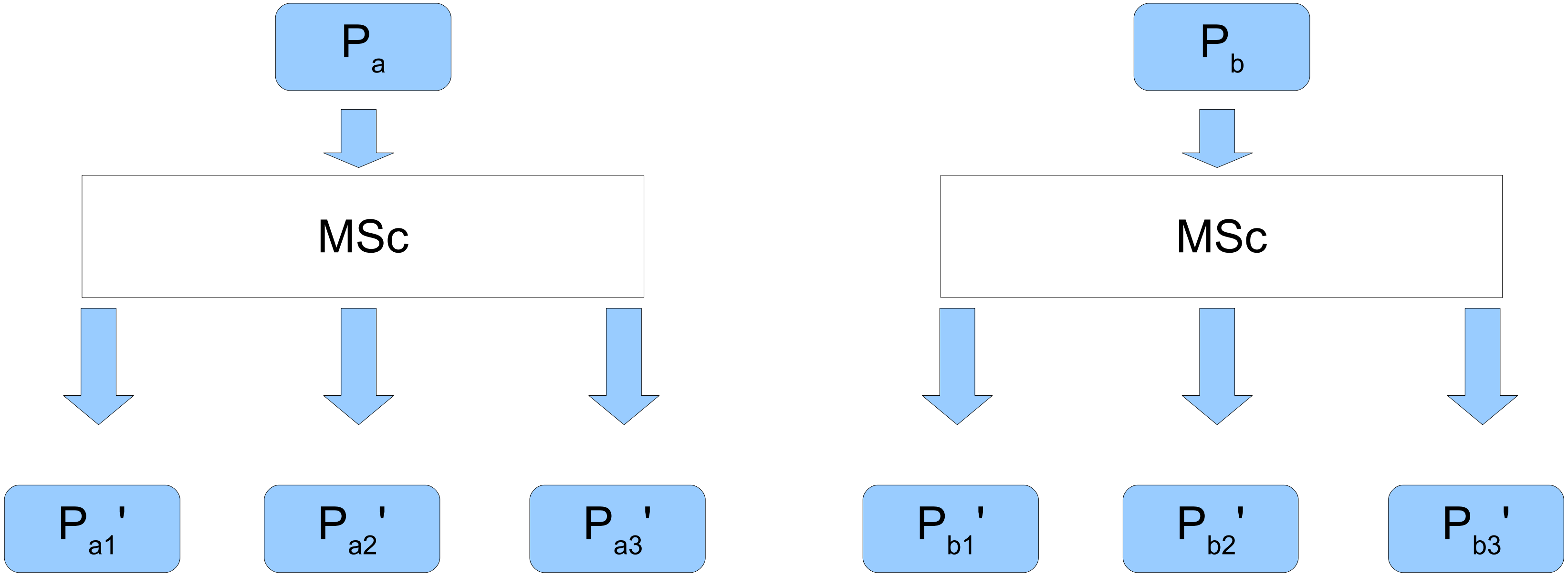
MSc

P_1'

P_2'

P_3'

Checking for the equivalence



P_a

MSc

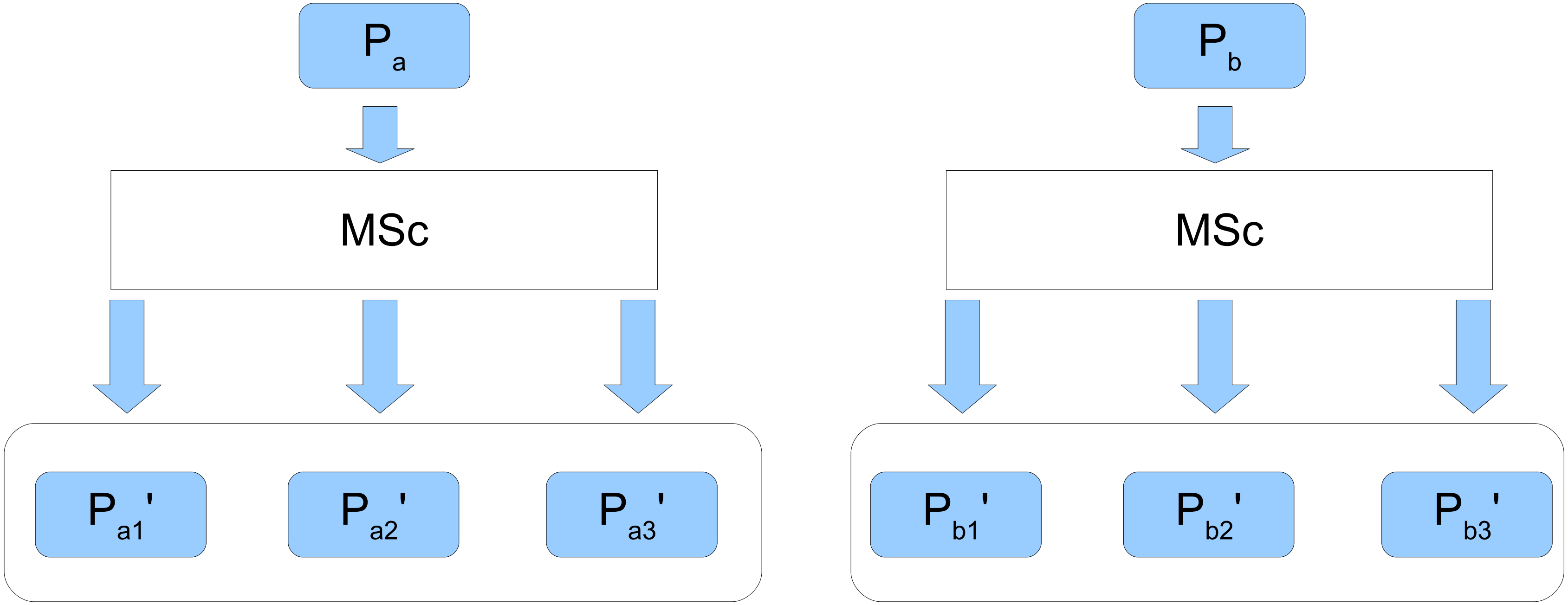
P_{a1}' P_{a2}' P_{a3}'

P_b

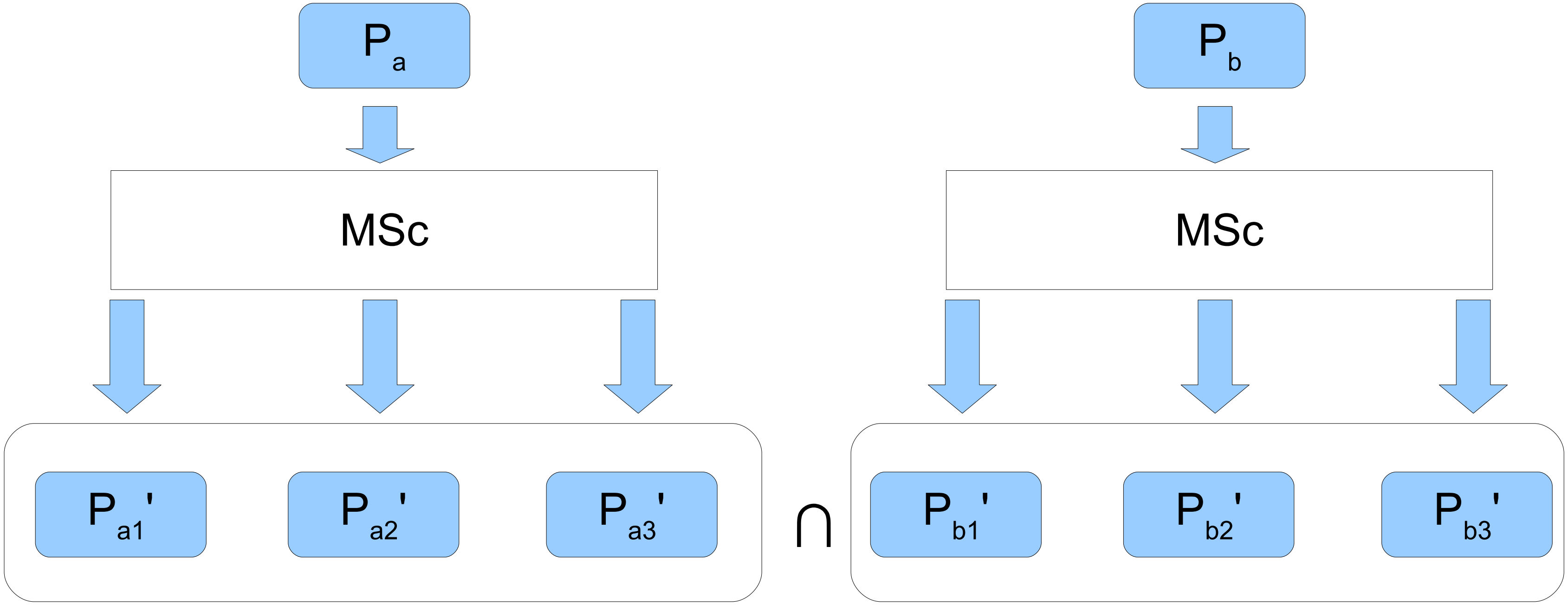
MSc

P_{b1}' P_{b2}' P_{b3}'

Checking for the equivalence



Checking for the equivalence

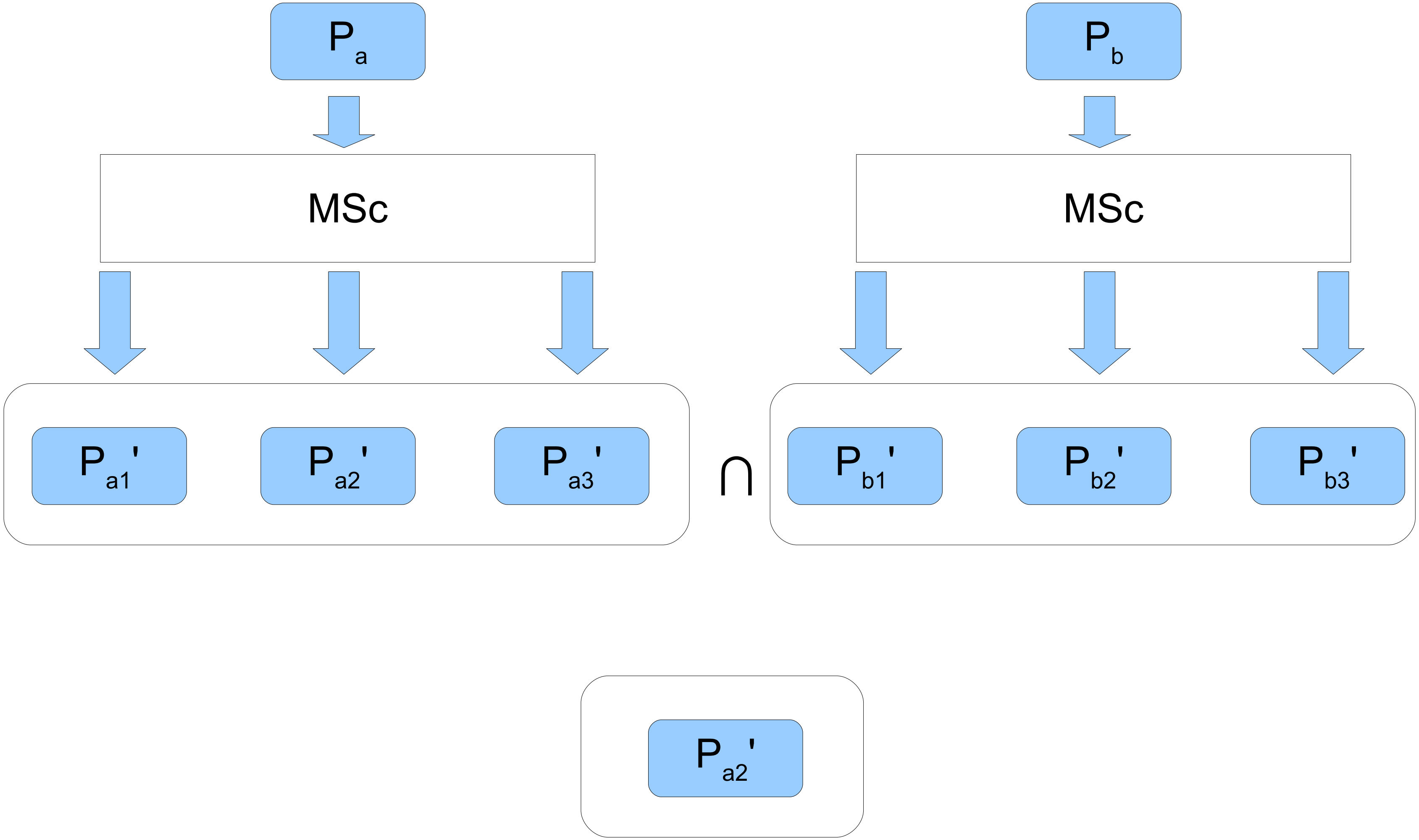


P_a P_b

MSc MSc

P_{a1}' P_{a2}' P_{a3}' $\cap$ P_{b1}' P_{b2}' P_{b3}'

Checking for the equivalence



P_a

MSc

P_{a1}' P_{a2}' P_{a3}'

$\cap$

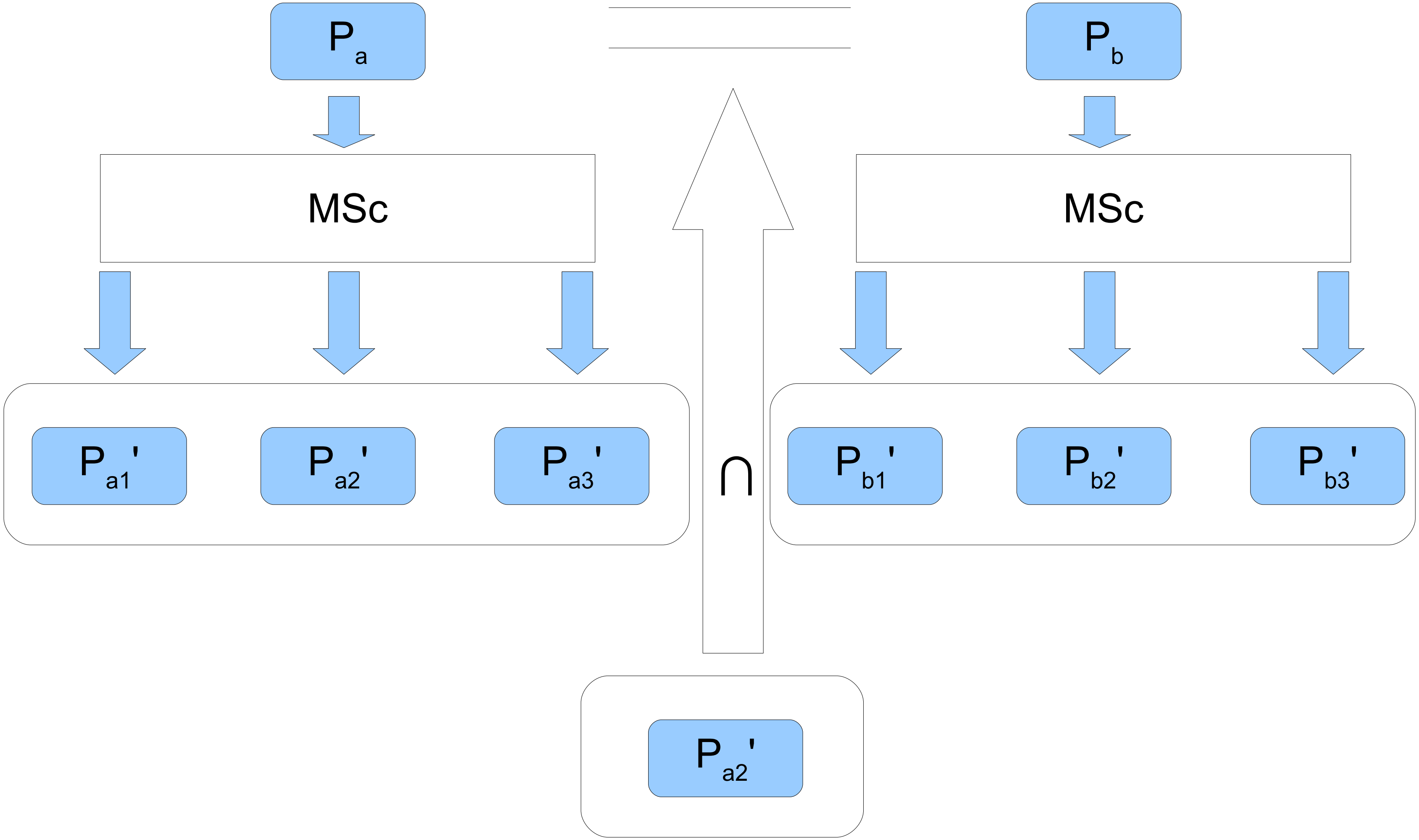
P_b

MSc

P_{b1}' P_{b2}' P_{b3}'

P_{a2}'

Checking for the equivalence



P_a = P_b

MSc

MSc

P_{a1}' P_{a2}' P_{a3}' $\cap$ P_{b1}' P_{b2}' P_{b3}'

P_{a2}'

Supercompiler Combinators

```
type SC  prog = prog → prog
type MSC prog = prog → [prog]


MERGE :: [SC prog] → MSC prog


L2 :: SC prog  → MSC prog → SC prog
```

Supercompiler Combinators

```
type SC  prog = prog → prog
type MSC prog = prog → [prog]

MERGE :: [SC prog]  → MSC prog
MERGE :: [MSC prog] → MSC prog
L2 :: SC prog  → MSC prog → SC prog
L2 :: MSC prog → MSC prog → MSC prog
```

Supercompiler Combinators

```
type SC  prog = prog → prog
type MSC prog = prog → [prog]

MERGE :: [SC prog]  → MSC prog
MERGE :: [MSC prog] → MSC prog
L2 :: SC prog  → MSC prog → SC prog
L2 :: MSC prog → MSC prog → MSC prog

BOOTSTRAP :: SC prog → MSC prog
```

The Recipe

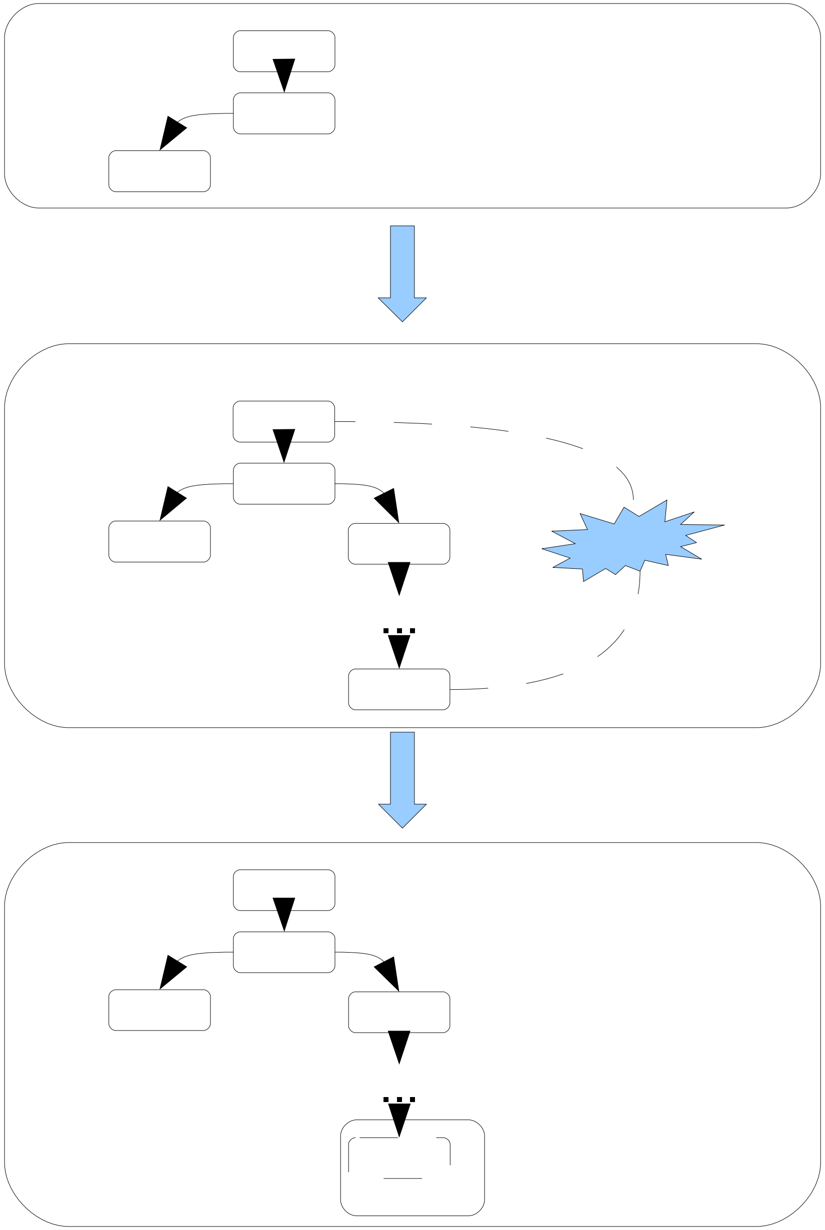
- Driving
- Folding
- Whistle
- Generalization

The Recipe

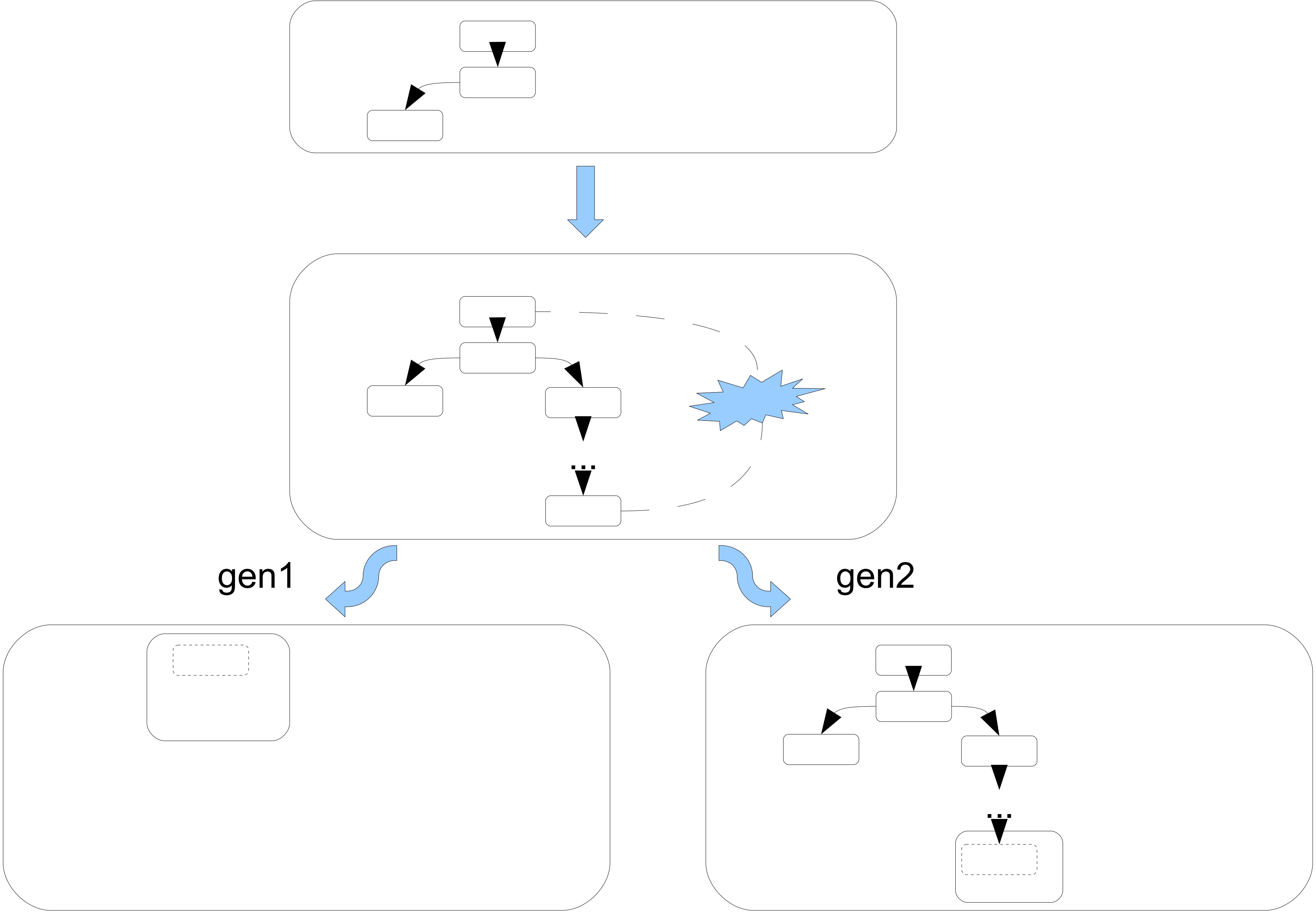
- Driving
- Folding
- Whistle
- Generalization

- Driving
- Folding
- Whistle
- Multi-Generalization

Standard Generalization (Sequence of Trees)



Multi-Generalization (Tree of Trees)



Multi-Generalization

Theorem

If whistle blows at any infinite branch and multi-generalization produces the finite number of variants, then the set of residual programs is finite.

I.Klyuchnikov and S. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. PSI-2011.
I. Klyuchnikov. MRSC: a framework for multi-result supercompilation. Preprint, Keldysh Institute of Applied Mathematics, 2011. To appear.

MRSC (2011)

- MRSC is the framework for constructing (multi-result and two-level) supercompilers by combining strategies for whistle, multi-generalization and escape from whistles.
- https://github.com/ilya-klyuchnikov/mrsc
- The preliminary results are exciting:
  - Automatically finding minimal proofs of the correctness of coherence protocols.
  - Automatic proof of commutativity of addition/multiplication by supercompilation.
  - Automatic proof of correctness of sorting algorithms by supercompilation.

The Taste of MRSC

```
class MultiSC(val ordering: PartialOrdering[SLLExpr])
  extends GenericMultiMachine[Expr, DriveInfo[SLLExpr], Extra]
  with SLLSyntax
  with SLLSemantics
  with SimpleDriving[SLLExpr]
  with Folding[SLLExpr]
  with PartialOrderingTermination[SLLExpr]
  with InAdvanceAllGens[SLLExpr]
```

The Plan

- Supercompilation in a nutshell
- Optimization vs Analysis
- Analyzing supercompilation (HOSC)
- Two-level supercompilation
- Multi-result supercompilation (MRSC)
- Finding a minimal proof by multi-result supercompilation
- On metasystem transitions

Multi-result supercompilation is naturaL for program analysis

Example #4. Finding Minimal Proof

Verification of Protocols

The model of cache-coherence protocols can be seen as a set of transition rules between states represented as a n-tuple of natural numbers.

The problem
Given a safe initial state, prove that unsafe state is unreachable.

The problem is well-studied and there is a lot of **specialized** methods to prove the correctness of protocols automatically.

Verification by Supercompilation

- A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler) / 2007
  - The proof is by program transformation:
    - ∀ `events: safe (go init events) == true`

General-purpose SC

Verification by Supercompilation

- A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler) / 2007
  - The proof is by program transformation:
    - ∀ `events: safe (go init events) == true`

General-purpose SC

- A series of works by A. Klimov (2010/2011)
  - Solving Coverability Problem for Monotonic Counter Systems by Supercompilation
  - Yet another algorithm for solving coverability problem for Monotonic Counter Systems

Minimalistic domain-specific SC

Verification by Supercompilation

- A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler) / 2007
  - The proof is by program transformation:
    - ∀ `events: safe (go init events) == true`

General-purpose SC

- A series of works by A. Klimov (2010/2011)
  - Solving Coverability Problem for Monotonic Counter Systems by Supercompilation
  - Yet another algorithm for solving coverability problem for Monotonic Counter Systems

Minimalistic domain-specific SC

- I. Klyuchnikov (2011)
  - Finding a minimal proof by multi-result supercompilation

DSL-based factory-made multi-result SC

Example #4. MOESI Protocol

```
case object MOESI extends Protocol {
  val start: OmegaConf = List(Omega, 0, 0, 0, 0)

  val rules: List[TransitionRule] = List(
     { case List(i, m, s, e, o) if i>=1   => List(i-1, 0, s+e+1, 0, o+m)   },
     { case List(i, m, s, e, o) if e>=1   => List(i, m+1, s, e-1, o)       },
     { case List(i, m, s, e, o) if s+o>=1 => List(i+m+s+e+o-1, 0, 0, 1, 0) },
     { case List(i, m, s, e, o) if i>=1   => List(i+m+s+e+o-1, 0, 0, 1, 0) })

  def unsafe(c: OmegaConf) = c match {
    case List(i, m, s, e, o) if m>=1 && (e + s + o) >= 1 => true
    case List(i, m, s, e, o) if m>=2                     => true
    case List(i, m, s, e, o) if e>=2                     => true
    case _ => false
  }

}
```

Just mini-DSL in Scala

Supercompiler in 3 slides

```
sealed trait Component {
  def +(comp: Component): Component
  def -(comp: Component): Component
  def >=(i: Int): Boolean
}

case class Value(i: Int) extends Component {
  override def +(comp: Component) = comp match {
    case Omega => Omega
    case Value(j) => Value(i + j)
  }
  override def -(comp: Component) = comp match {
    case Omega => Omega
    case Value(j) => Value(i - j)
  }
  override def >=(j: Int) = i >= j
}

case object Omega extends Component {
  def +(comp: Component) = Omega
  def -(comp: Component) = Omega
  def >=(comp: Int) = true
}
```

Supercompiler in 3 slides

```
trait CountersPreSyntax extends PreSyntax[OmegaConf] {
  val instance = OmegaConfInstanceOrdering
  def rebuildings(c: OmegaConf) = gens(c) - c
  def gens(c: OmegaConf): List[OmegaConf] = c match {
    case Nil => List(Nil)
    case e :: c1 => for (cg <- genComp(e); gs <- gens(c1)) yield cg :: gs
  }
  def genComp(c: Component): List[Component] = c match {
    case Omega => List(Omega)
    case value => List(Omega, value)
  }
}

trait CountersSemantics extends RewriteSemantics[OmegaConf] {
  val protocol: Protocol
  def drive(c: OmegaConf) = protocol.rules.map { _.lift(c) }
}

object OmegaConfInstanceOrdering extends SimplePartialOrdering[OmegaConf] {
  def lteq(c1: OmegaConf, c2: OmegaConf) = (c1, c2).zipped.forall(lteq)
  def lteq(x: Component, y: Component) = (x, y) match {
    case (Omega, _) => true
    case (_, _) => x == y
  }
}
```

Supercompiler in 3 slides

```
trait LWhistle {
  val l: Int
  def unsafe(counter: OmegaConf) = counter exists {
    case Value(i) => i >= l
    case Omega => false
  }
}

case class CounterMultiSc(val protocol: Protocol, val l: Int)
  extends CountersPreSyntax
  with LWhistle
  with CountersSemantics
  with RuleDriving[OmegaConf]
  with SimpleInstanceFoldingToAny[OmegaConf, Int]
  with SimpleUnaryWhistle[OmegaConf, Int]
  with ProtocolSafetyAware
  with SimpleGensWithUnaryWhistle[OmegaConf, Int]
```

Proofs by supercompilation

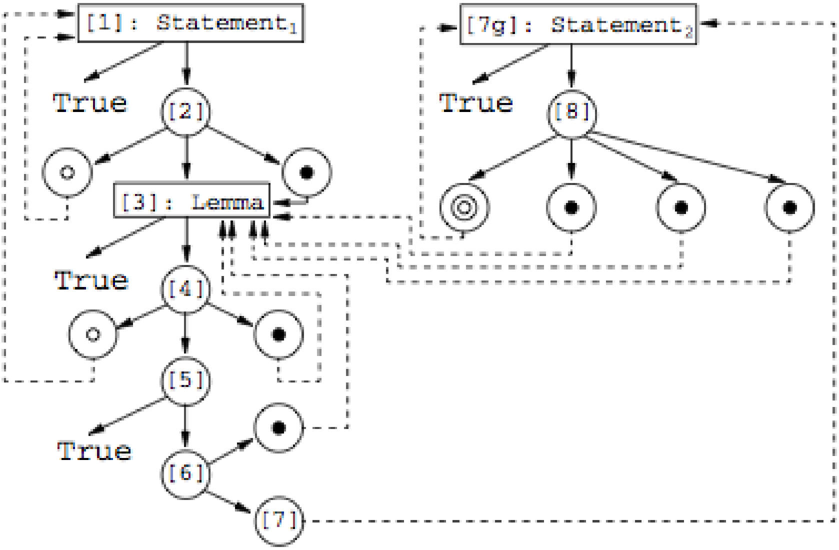


Fig. 3. Graph of the automatic proof.

A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler) / 2007

Proofs by supercompilation

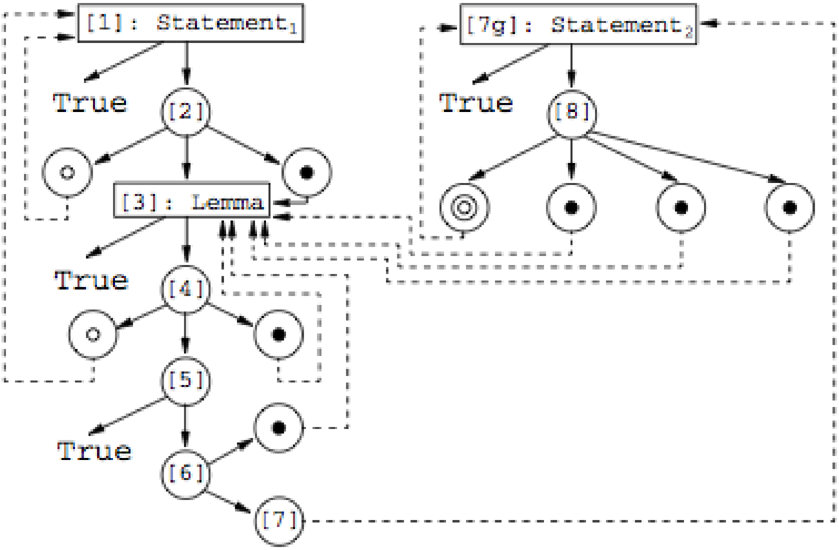
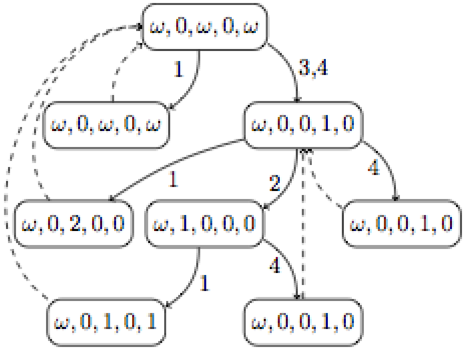


Fig. 3. Graph of the automatic proof.

A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler) / 2007



I. Klyuchnikov. MRSC: a framework for multi-result supercompilation / 2011

Proofs by supercompilation

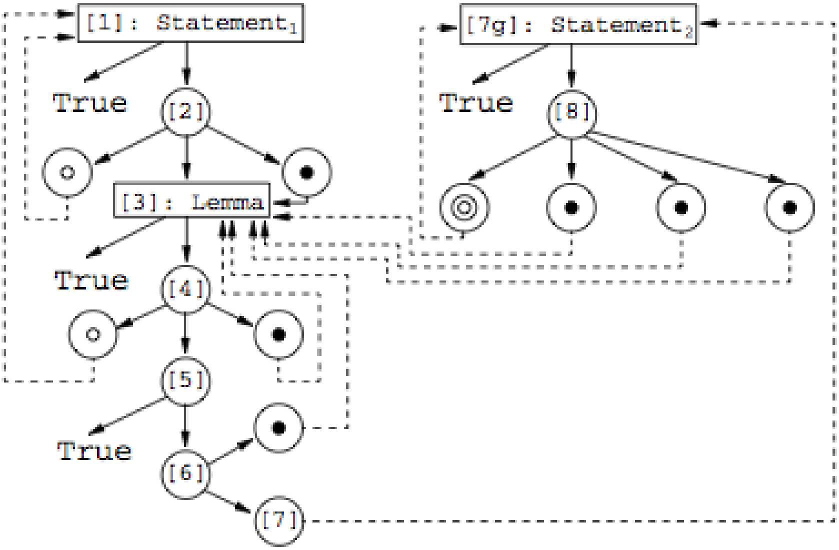
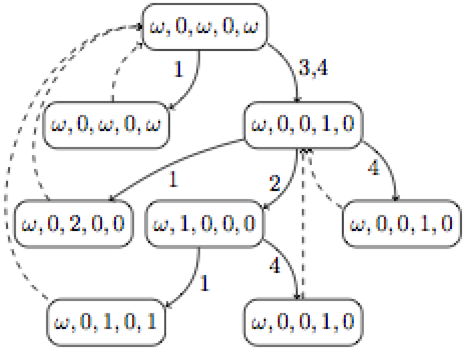


Fig. 3. Graph of the automatic proof.

A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler) / 2007

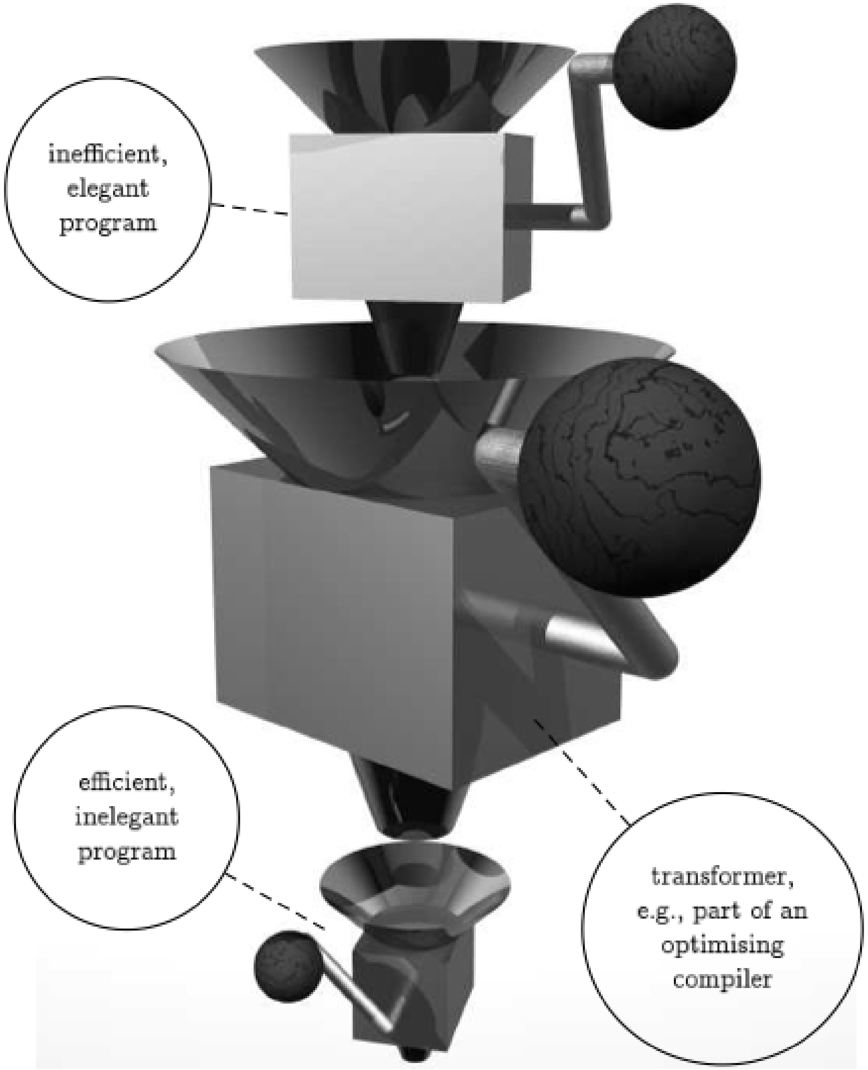
22 steps

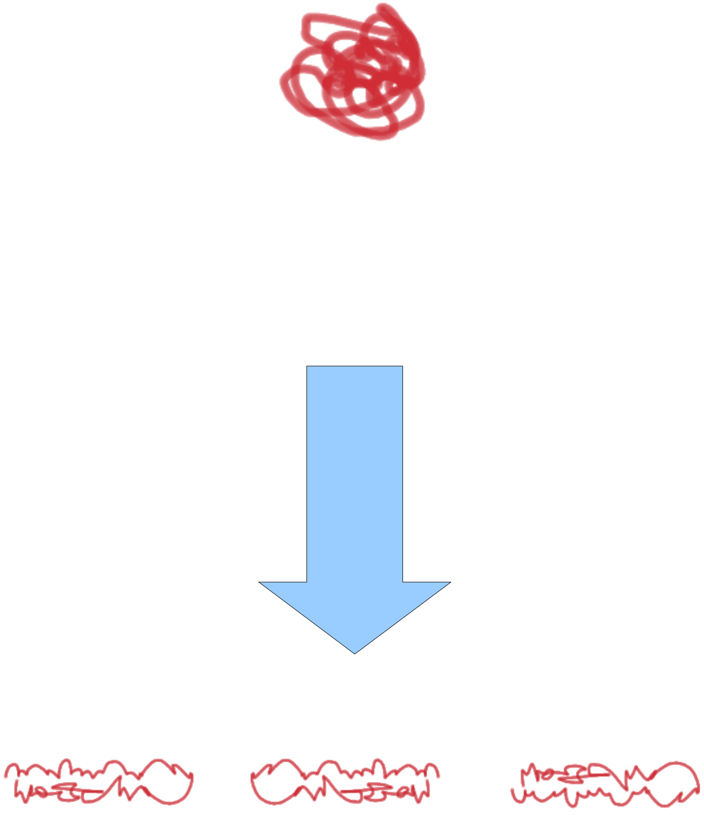


I. Klyuchnikov. MRSC: a framework for multi-result supercompilation / 2011

8 steps

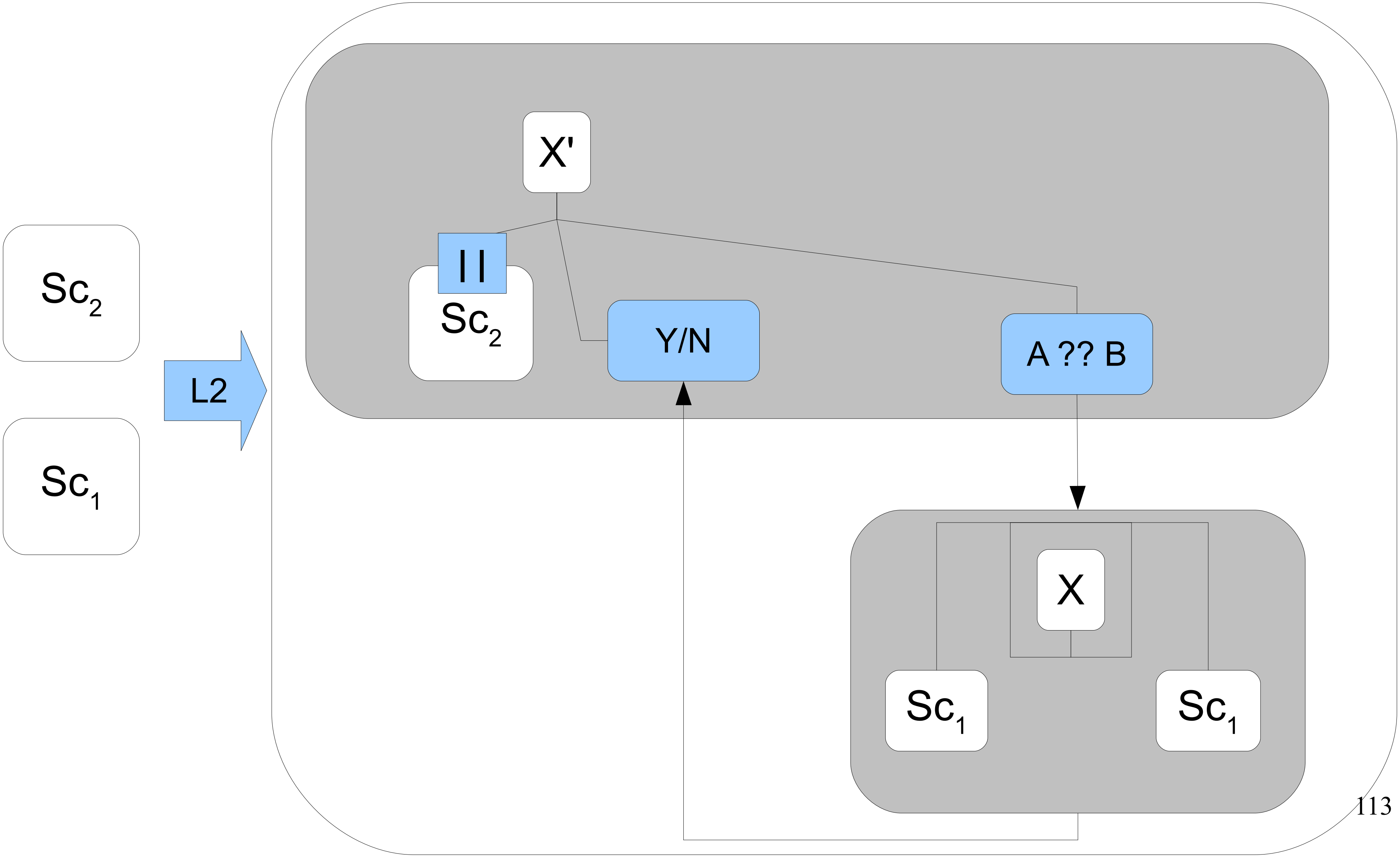
Optimization and Analysis



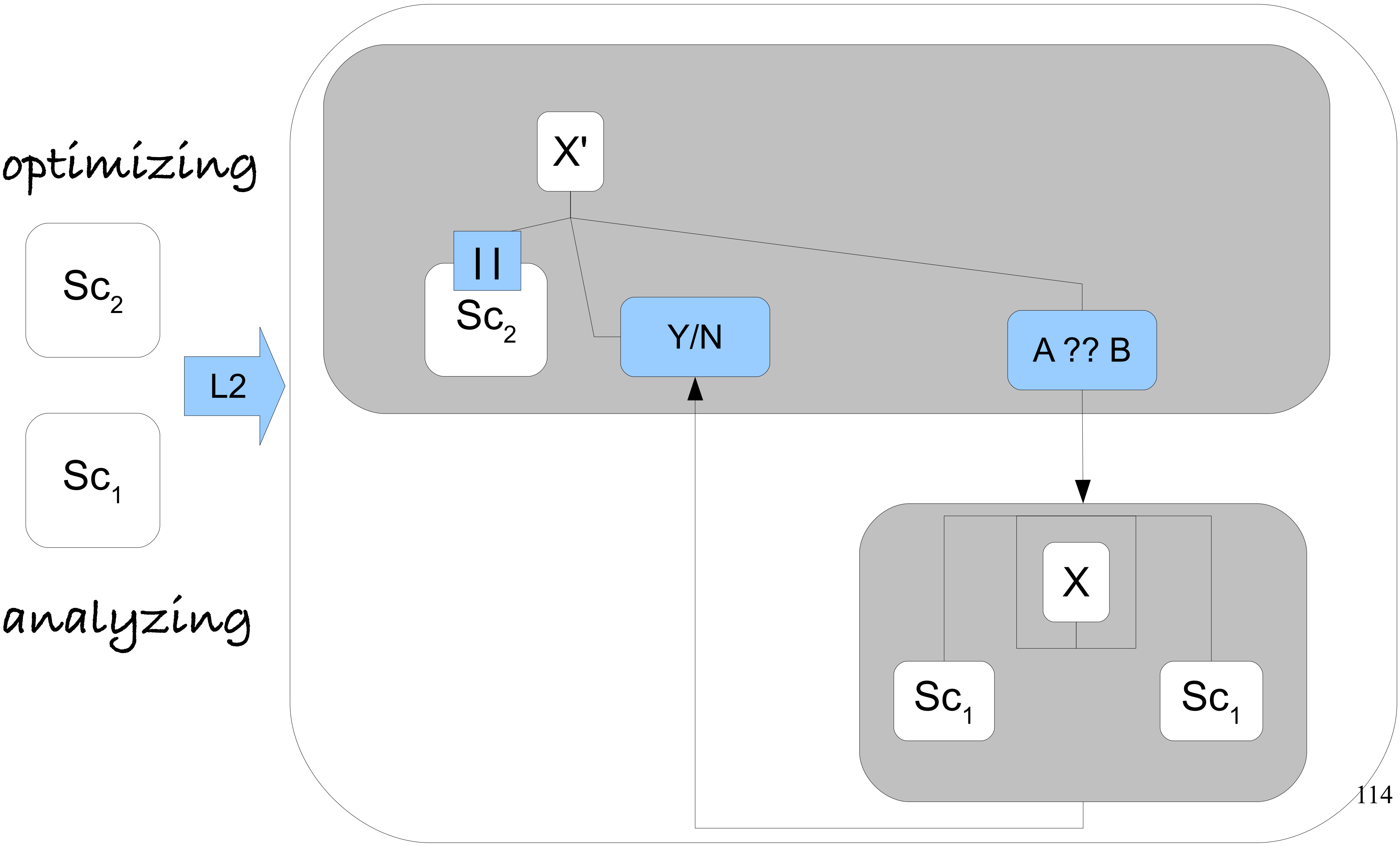


...

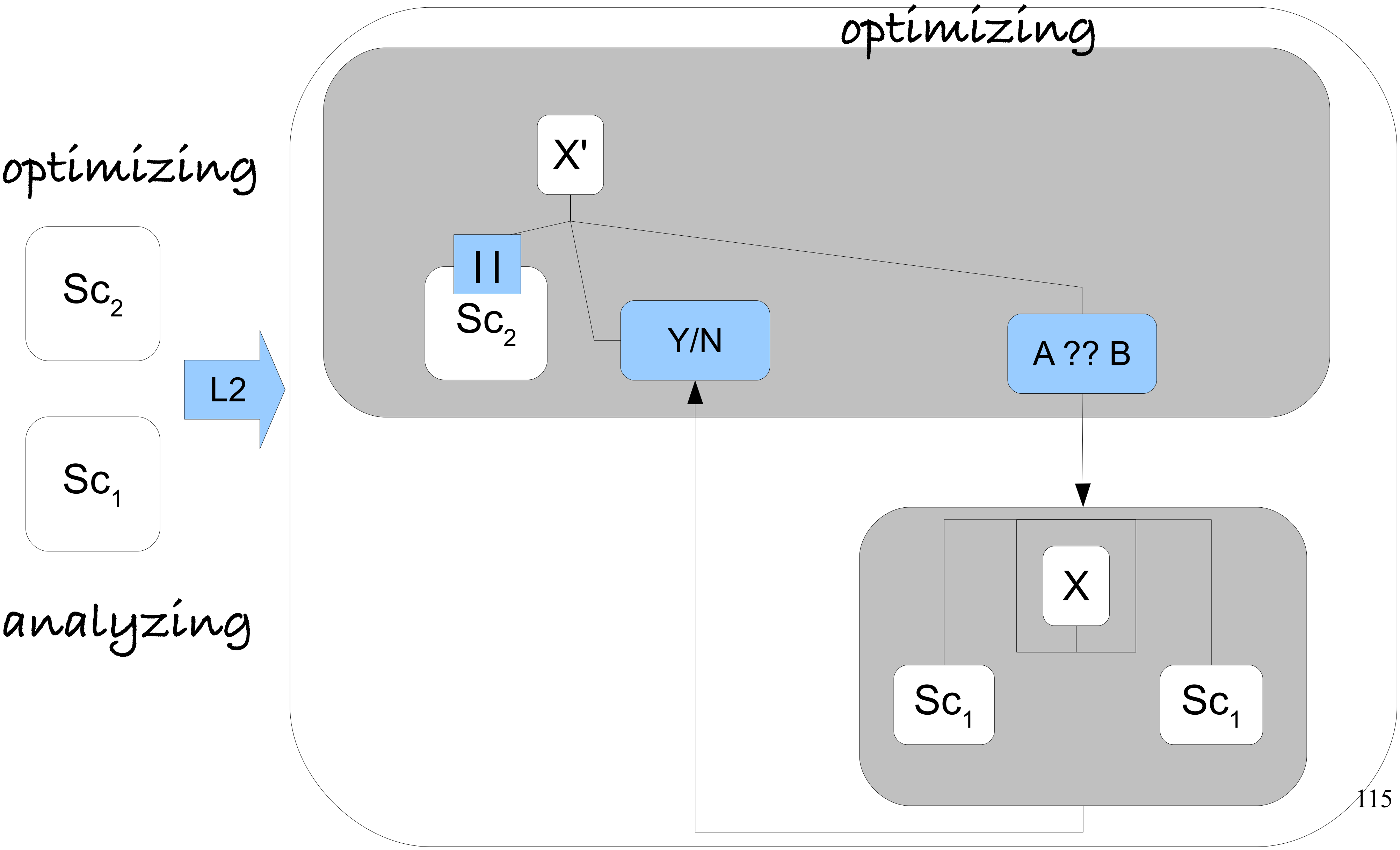
Optimization and Analysis



Optimization and Analysis



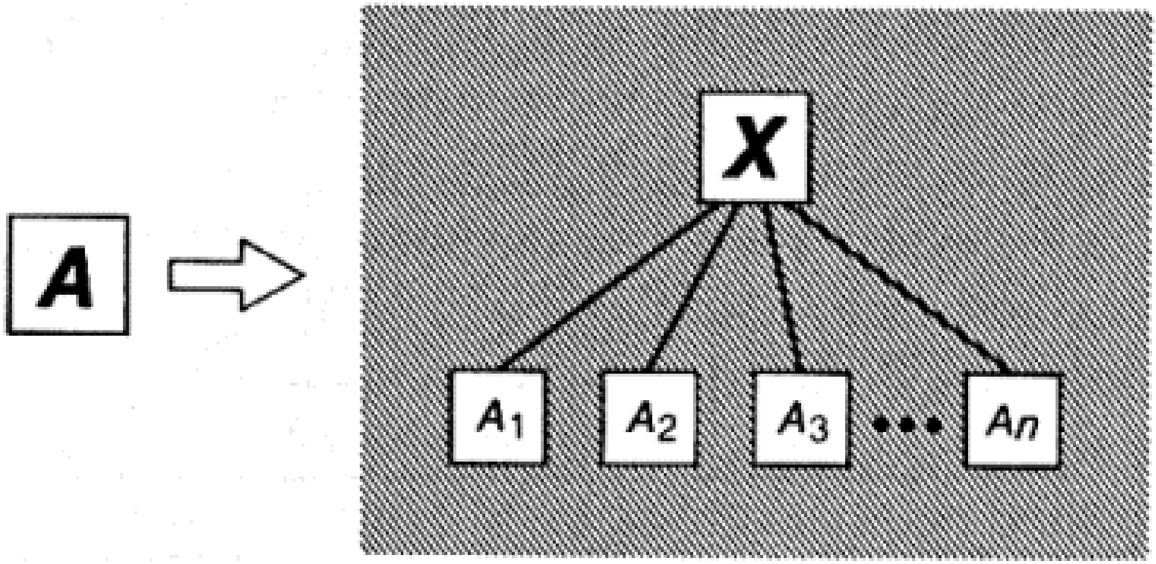
Optimization and Analysis



The Plan

- Supercompilation in a nutshell
- Optimization vs Analysis
- Analyzing supercompilation (HOSC)
- Two-level supercompilation
- Multi-result supercompilation (MRSC)
- Finding a minimal proof by multi-result supercompilation
- On metasystem transitions

Metasystem Transition



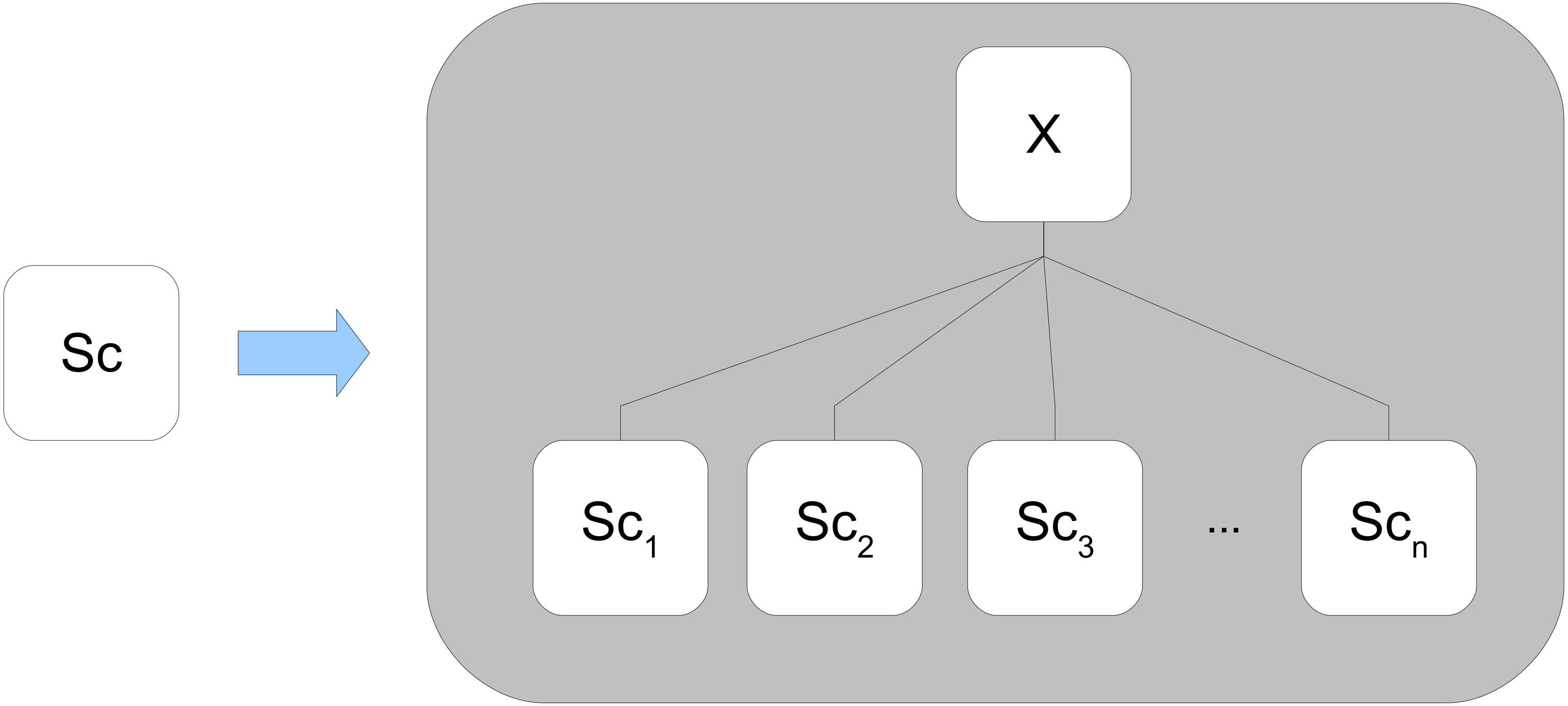
The “Formula” of Metasystem Transition

Control +

Branching Growth ⇒

Metasystem Transition

Supercompilation



Supercompilation is treated as an elementary operation.

The "Formula" of Metasystem Transition

Control +

Branching Growth ⇒

Metasystem Transition

Two-Level Supercompilation +

Multi-Result Supercompilation ⇒

Metasystem Transition

The “Formula” of Metasystem Transition

Control +

Branching Growth ⇒

Metasystem Transition

Two-Level Supercompilation +

Multi-Result Supercompilation ⇒

Metasystem Transition

The projection of the formula onto supercompilation

I.Klyuchnikov and S. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. PSI-2011.

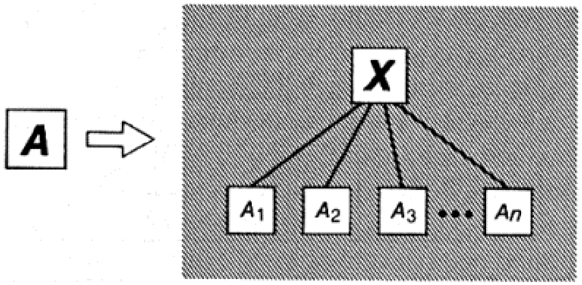
Conslusion

Supercompilation is a unified method for:

- Program optimization by transformation
- Program analysis by transformation

Supercompilation is based on the idea of metasystem transitions.

Control + Branching Growth ⇒
Metasystem Transition



Thanks you for your patience!

QUESTIONS?