Metacomputation
Gentle Introduction to Advanced Topics

Ilya Klyuchnikov

Keldysh Institute of Applied Mathematics
(Russian Academy of Sciences)
Metacomputation: The Big Picture

Future Works

New Applications

New Methods

Basics

Target Language, Evaluation

Driving SRT, PPT

Methods

metaprograms

URANAN

Applications

Analysis

Verification

Optimization

Specialization

Transformation

SCP

L2MRSC

Inverse Computation

Neighborhood Testing

Non-Standard Semantics
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 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
Execution

...
N. Jones. The Essence of Program Transformation by Partial Evaluation and Driving / 1999
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

$S_0$ 

$S_1$

$S_2$

$S_3$

... 

$S_n$ 

cond' 

cond''
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)

“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)

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

inefficient, elegant program

efficient, inelegant program

transformer, e.g., part of an optimising compiler
Optimization vs Analysis
Optimization vs Analysis

- inefficient, elegant program
- efficient, inelegant program
- transformer, e.g., part of an optimising compiler
Optimization vs Analysis

- Reducing execution time
- Reducing code size

- Simplifying the structure
- Revealing hidden properties
**Optimization vs Analysis**

<table>
<thead>
<tr>
<th>Reducing execution time</th>
<th>Simplifying the structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reducing code size</td>
<td>Revealing hidden properties</td>
</tr>
</tbody>
</table>

The “best” output program.  
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.

Example #1. Church numbers

\[ 0 = \lambda s \; z \rightarrow z \]
\[ 1 = \lambda s \; z \rightarrow s \; z \]
\[ 2 = \lambda s \; z \rightarrow s \; (s \; z) \]
\[ 3 = \lambda s \; z \rightarrow s \; (s \; (s \; z)) \]
\[ \ldots \]
\[ n = \lambda s \; z \rightarrow s^n \; z \]

\[ f^{m+n} \; z \rightarrow s^m \; (s^n \; z) \]
\[ f^{m\cdot n} \; z \rightarrow (s^n)^m \; z \]
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);

\[ \text{mult} \ x \ y \equiv \text{unchurch} \ (\text{churchMult} \ (\text{church} \ x) \ (\text{church} \ y)) \]

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

\[ P_1 \quad P_2 \]
Inferring the equivalence of programs
Inferring the equivalence of programs

\[ P_1 \xrightarrow{SC} P_1' \equiv P_2 \xrightarrow{SC} P_2' \]
Inferring the equivalence of programs
Example #2. Abstract machines

Example #2. Abstract machines

Example #2. Abstract machines

Danvy, Millikin (by hand):

\[ \text{am}_1 \quad \text{am}_2 \]

Example #2. Abstract machines

Danvy, Millikin (by hand):

Example #2. Abstract machines

Danvy, Millikin (by hand):

\[ \text{am}_1 \rightarrow \text{am}' \rightarrow \text{am}'' \rightarrow \text{am}_2 \]

HOSC (automatically):

\[ \text{am}_1 \rightarrow \text{am}_2 \]

Example #2. Abstract machines

Danvy, Millikin (by hand):


HOSC (automatically):

Example #2. Abstract machines

Danvy, Millikin (by hand):

\[ \text{am}_1 \rightarrow \text{am}' \rightarrow \text{am}'' \rightarrow \text{am}_2 \]

HOSC (automatically):

\[ \text{am}_1 \rightarrow \text{am} \rightarrow \text{am}_2 \]

Example #2. Abstract machines

Danvy, Millikin (by hand):

\[ \text{am}_1 \rightarrow \text{am}' \rightarrow \text{am}'' \rightarrow \text{am}_2 \]

HOSC (automatically):

\[ \text{am}_1 \rightarrow \text{am} \rightarrow \text{am}_2 \]


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
  • \( \text{sc}(\text{int}, \text{prog}) = \text{prog}' \)
  • \( \text{sc}(\text{sc}, \text{int}) = \text{compiler} \)
  • \( \text{sc}(\text{sc}, \text{sc}) = \text{compiler generator} \)

The most popular (old) idea
Approaches to self-application

- Futamura projections
  - $\text{sc}(\text{int}, \text{prog}) = \text{prog}'$
  - $\text{sc}(\text{sc}, \text{int}) = \text{compiler}$
  - $\text{sc}(\text{sc}, \text{sc}) = \text{compiler generator}$

- Distillation
- Two-level supercompilation
- ...

The most popular (old) idea
Approaches to self-application

- Futamura projections
  - $sc(\text{int, prog}) = \text{prog}'$
  - $sc(sc, \text{int}) = \text{compiler}$
  - $sc(sc, sc) = \text{compiler generator}$

- Distillation

- Two-level supercompilation

- ...

The most popular (old) idea

Rather new approaches
Approaches to self-application

- Futamura projections
  - \( \text{sc}(\text{int}, \text{prog}) = \text{prog}' \)
  - \( \text{sc}(\text{sc}, \text{int}) = \text{compiler} \)
  - \( \text{sc}(\text{sc}, \text{sc}) = \text{compiler generator} \)

- Distillation
  - Rather new approaches

- Two-level supercompilation
  - The most popular (old) idea

...
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
Escape from Whistle

- $S_0$ -> $S_1$
- $S_1$ -> $S_2$
- $S_1$ -> $S_3$
- $S_1$ -> $...$
- $S_1$ -> $S_n$

- Cond' $ightarrow S_2$
- Cond'' $ightarrow S_3$
- Cond' $ightarrow ...$
- Cond' $ightarrow S_n$

Diagram shows a sequence of states ($S_0, S_1, S_2, S_3, ... S_n$) connected by conditional transitions (Cond', Cond'').
Escape from Whistle

Diagram:

- $S_0$ connected to $S_1$ by $\text{cond}'$
- $S_1$ connected to $S_2$ by $\text{cond}'$
- $S_1$ connected to $S_3$ by $\text{cond}''$
- $S_3$ connected to $\ldots$
- $S_n$ connected to $S'_n$
Escape from Whistle

\[
\begin{align*}
S_0 & \quad \text{cond}' \quad S_1 \quad \text{cond}'' \quad S_2 \\
S_1 & \quad \text{cond}'' \quad S_3 \\
S_3 & \quad \ldots \\
S_n & \quad \equiv_{SC} \quad S'_n
\end{align*}
\]
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; };

or (even m) (odd m)

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));}

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;};
   S x -> f x;};

in f m

61
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).

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

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

\[
\text{doubleA} = \epsilon \mid a \text{ doubleA } a \\
\text{doubleA} = \epsilon \mid a \ a \ a \ \text{doubleA}
\]
It Is Worth to Do!

The task: grammar transformation

\[
\text{doubleA} = \epsilon \mid a \; \text{doubleA} \; a \\
\text{doubleA} = \epsilon \mid a \; a \; \text{doubleA}
\]

\[
L2(SC_{---}, SC_{---}), L2(SC_{+++}, SC_{+++}), L2(SC_{++-}, SC_{++-}), \\
L2(SC_{--+}, SC_{--+}), \ldots \quad \text{FAILURE}
\]
The task: grammar transformation

\[
\text{doubleA} = \epsilon \mid a \ \text{doubleA} \ a
\]
\[
\text{doubleA} = \epsilon \mid a \ a \ \text{doubleA}
\]

\[
L2(\text{SC}_{--}, \text{SC}_{--}), \ L2(\text{SC}_{+-}, \text{SC}_{+-}), \ L2(\text{SC}_{-+}, \text{SC}_{-+}), \\
L2(\text{SC}_{--}, \text{SC}_{-+}), \ldots \quad \text{FAILURE}
\]
\[
L2(\text{SC}_{++}, \text{SC}_{--}), \ldots \quad \text{SUCCESS}
\]
It Is Worth to Do!

The task: grammar transformation

doubleA = \epsilon \mid a \ \text{doubleA} \ a

doubleA = \epsilon \mid a \ a \ \text{doubleA}

L2(Sc_{___}, Sc_{___}), L2(Sc_{+++}, Sc_{+++}), L2(Sc_{+-+}, Sc_{+-+}),
L2(Sc_{-+-}, Sc_{-+-}), \ldots \quad \text{FAILURE}

L2(Sc_{++-}, Sc_{++-}), \ldots \quad \text{SUCCESS}

Interesting Pattern:
L2(Sc_2, Sc_1) – Sc_2 should be a bit smarter than Sc_1 (A managing person should be \textbf{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—
Step #1: 2 Instances of a Supercompiler
Step #2: 3 Instances of a Supercompiler
Step #3: Combining 2 Supercompilers
Step #4: Combining Many Supercompilers
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

![Diagram](image-url)
Let’s Treat Many Supercompilers as One Multi-Result Supercompiler

MSc

P

\[ P_1', \quad P_2', \quad P_3' \]
Checking for the equivalence

\[
\begin{align*}
P_a' & \quad P_{a1}' & \quad P_{a2}' & \quad P_{a3}' \\
P_a & \quad \text{MSc} & \\
P_b' & \quad P_{b1}' & \quad P_{b2}' & \quad P_{b3}' \\
P_b & \quad \text{MSc}
\end{align*}
\]
Checking for the equivalence

\[ P_{a1}' \quad P_{a2}' \quad P_{a3}' \quad \text{MSc} \]

\[ P_{b1}' \quad P_{b2}' \quad P_{b3}' \quad \text{MSc} \]
Checking for the equivalence

$P_{a} \cap P_{b}$
Checking for the equivalence of:

\[ P_a \cap P_{b1} \]

\[ P_a \cap P_{b2} \]

\[ P_a \cap P_{b3} \]
Checking for the equivalence

\[ P_{a_1}' \cap P_{a_2} \cap P_{a_3}' = P_{b_1}' \cap P_{b_2} \cap P_{b_3}' \]
Supercompiler Combinators

\[\text{type \ SC \ prog} = \text{prog} \rightarrow \text{prog}\]

\[\text{type \ MSC \ prog} = \text{prog} \rightarrow [\text{prog}]\]

\[\text{MERGE} :: [\text{SC \ prog}] \rightarrow \text{MSC \ prog}\]

\[\text{L2} :: \text{SC \ prog} \rightarrow \text{MSC \ prog} \rightarrow \text{SC \ prog}\]
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.
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:
  - \( \forall \text{events: safe (go init events) == true} \)
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
- 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
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:
    - $\forall$ events: safe (go init events) == true
- 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
- I. Klyuchnikov (2011)
  - Finding a minimal proof by multi-result supercompilation
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
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
}
trait CountersPreSyntax extends PreSyntax[OmegaConf] {
  val instance = OmegaConfInstanceOrdering
  def rebuilding(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

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

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

22 steps

8 steps
Optimization and Analysis

inefficient, elegant program

efficient, inelegant program

transformer, e.g., part of an optimising compiler
Optimization and Analysis

\[ \text{Sc}_2 \quad \text{II} \quad \text{Sc}_2 \quad \text{Y/N} \quad \text{A ?? B} \]

\[ \text{L2} \quad \text{Sc}_1 \]

\[ \text{X'} \quad | \quad \text{X} \quad \text{Sc}_1 \quad \text{Sc}_1 \]
Optimization and Analysis

optimizing

Sc₂

L2

analyzing

Sc₁

X'
Optimization and Analysis

Optimizing

$\text{Sc}_2$

analyzing

$\text{Sc}_1$

$L2$

$\text{Sc}_2$

$X'$

$\text{Y/N}$

$A \text{ ?? } B$

$\text{X}$

$\text{Sc}_1$

$\text{Sc}_1$
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 \Rightarrow

Metasystem Transition
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 $\Rightarrow$
Metasystem Transition

Two-Level Supercompilation +
Multi-Result Supercompilation $\Rightarrow$
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.
Conclusion

Supercompilation is a unified method for:

- Program optimization by transformation
- Program analysis by transformation

Supercompilation is based on the idea of metasystem transitions.

\[ \text{Control} + \text{Branching Growth} \Rightarrow \text{Metasystem Transition} \]
Thanks you for your patience!

QUESTIONS?