Metacomputation
A Gentle Introduction to Advanced Topics

Ilya Klyuchnikov

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

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \ldots \rightarrow S_n \]

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
Optimization vs Analysis

- Inefficient, elegant program
- Efficient, inelegant program
- Transformer, e.g., part of an optimising compiler
Optimization vs Analysis
## Optimization vs Analysis

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

<table>
<thead>
<tr>
<th>Optimization Goals</th>
<th>Analysis Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reducing execution time</td>
<td>Simplifying the structure</td>
</tr>
<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

\[ 0 = \lambda s \, z \rightarrow z \]

\[ 1 = \lambda s \, z \rightarrow s \, z \]

\[ 2 = \lambda s \, z \rightarrow s \, (s \, z) \]

\[ 3 = \lambda s \, z \rightarrow s \, (s \, (s \, z)) \]

\[ \vdots \]

\[ n = \lambda s \, z \rightarrow s^n \, z \]

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

\[ f^{m\times n} \, z \rightarrow ((s^n)^m) \, z \]
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);

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

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);

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

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

\[ \text{P}_1 \xrightarrow{\text{SC}} \text{P}_1' \]

\[ \text{P}_2 \xrightarrow{\text{SC}} \text{P}_2' \]
Inferring the equivalence of programs
Inferring the equivalence of programs

\[ P_1 \xrightarrow{\text{SC}} P_1' \quad \cong \quad P_2 \xleftarrow{\text{SC}} P_2' \]
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):

1. $am_1$ → $am'$ → $am''$ → $am_2$

Example #2. Abstract machines

Danvy, Millikin (by hand):

\[ am_1 \rightarrow am' \rightarrow am'' \rightarrow am_2 \]

HOSC (automatically):

\[ am_1 \rightarrow am_2 \]

Example #2. Abstract machines

Danvy, Millikin (by hand):

HOSC (automatically):

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):

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

31 loc \quad 13 loc \quad 51 loc

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

- 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

- **Two-level supercompilation**

- ...
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

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \ldots \rightarrow S_n \]

cond'

cond''
Escape from Whistle

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \ldots \rightarrow S_n \equiv SC \rightarrow S'_n \]
Two-Level Supercompilation
Two-Level Supercompilation

There is a shortcut!
Shortcut Two-Level supercompilation

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n \]

cond'

cond''
Shortcut Two-Level supercompilation

I. Klyuchnikov. Towards effective two-level supercompilation. KIAM Preprint #81. 2010
Example #3. Even or odd

```haskell
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;}

... SC

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

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;};
  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));}
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: \( SC_{--}, SC_{+-}, SC_{-+}, \ldots \) - 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

\[
\text{doubleA} = \epsilon \mid a \text{ doubleA } a \\
\text{doubleA} = \epsilon \mid a \ a \ a \ \text{doubleA}
\]
The task: grammar transformation

doubleA = $\epsilon$ | a doubleA a
doubleA = $\epsilon$ | a a doubleA

L2($SC_{---},SC_{---}$), L2($SC_{---},SC_{---}$), L2($SC_{---},SC_{---}$), L2($SC_{---},SC_{---}$), ...

FAILURE
The task: grammar transformation

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

L2(\text{SC--}, \text{SC--}), L2(\text{SC++}, \text{SC--}), L2(\text{SC--}, \text{SC++}),
L2(\text{SC---}, \text{SC---}), ... \quad \text{FAILURE}
L2(\text{SC++}, \text{SC---}), ... \quad \text{SUCCESS}

I. Klyuchnikov. Towards effective two-level supercompilation. KIAM Preprint #81. 2010
The task: grammar transformation

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

\[ L_2(\text{Sc}_{--}, \text{Sc}_{--}), L_2(\text{Sc}_{+-}, \text{Sc}_{--}), L_2(\text{Sc}_{--}, \text{Sc}_{++}), \]
\[ L_2(\text{Sc}_{--}, \text{Sc}_{++}), \ldots \quad \text{FAILURE} \]
\[ L_2(\text{Sc}_{++}, \text{Sc}_{--}), \ldots \quad \text{SUCCESS} \]

Interesting Pattern:
\[ L_2(\text{Sc}_2, \text{Sc}_1) \rightarrow \text{Sc}_2 \text{ should be a bit smarter than Sc}_1 \text{ (A managing person should be a bit clever than a person being managed)} \]
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
Step #5?
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

\[ P 
\rightarrow \]

\[ Sc_2 \rightarrow \]

\[ P_1' \]

\[ Sc_2 \rightarrow \]

\[ P_2' \]

\[ Sc_3 \rightarrow \]

\[ P_3' \]
Let’s Treat Many Supercompilers as One Multi-Result Supercompiler

Diagram:

- **P**
  - **MSc**
    - **P₁'**
    - **P₂'**
    - **P₃'**
Checking for the equivalence

\[ P_{a1}' \rightarrow P_{a2}' \rightarrow P_{a3}' \rightarrow MSc \]

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

\[ P_{a1}' \cap P_{a2}' \cap P_{a3}' \cap P_{b1}' \cap P_{b2}' \cap P_{b3}' \]
Checking for the equivalence

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

\[ P_{a2}' \]

\[ P_a \]

\[ P_b \]
Checking for the equivalence

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

\[ \bigcap \]

\[ P_{a2}' \]

\[ P_{a1}' \]

\[ P_{a2} \]

\[ P_{a3}' \]

\[ P_{b1}' \]

\[ P_{b2}' \]

\[ P_{b3}' \]
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.
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.
class MultiSC(val ordering: PartialOrdering[SLLEExpr])
  extends GenericMultiMachine[Expr, DriveInfo[SLLEExpr], Extra]
  with SLLSyntax
  with SLLSemantics
  with SimpleDriving[SLLEexpr]
  with Folding[SLLEexpr]
  with PartialOrderingTermination[SLLEexpr]
  with InAdvanceAllGens[SLLEexpr]
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:
  \[ \forall \text{events: safe (go init events)} \Rightarrow \text{true} \]
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:
    - \( \forall \text{events}: \text{safe (go init events)} = \text{true} \)
- 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
    }
}
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 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
  }
}
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

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

Fig. 3. Graph of the automatic proof.
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

22 steps

8 steps
Optimization and Analysis

inefficient, elegant program

efficient, inelegant program

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

optimizing

Sc₂
Sc₁

analyzing

X'
Y/N
A ?? B

X
Sc₁
Sc₁
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 \rightarrow
Metasystem Transition

Two-Level Supercompilation +
Multi-Result Supercompilation \rightarrow
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?