A Simple Supercompiler Formally Verified in Coq

Dimitur Krustev

IGE+XAO Balkan

4 July 2010 / META 2010
Outline

1. Introduction
   - Questions on the Title
   - Decomposition of Supercompilation
   - Coq Features Used

2. Supercompiler Organization and Correctness Proof
   - Expression Language and Simple Normalization
   - Propagation of Test Outcomes in Branches
   - Full Language, Loop Unrolling

3. Possible Extensions and Applications
   - Test Generation, Extensional Equivalence
   - More Realistic Language
   - Use Information Propagation in Isolation

Dimitur Krustev
A Simple Supercompiler Formally Verified in Coq
Outline

1. Introduction
   - Questions on the Title
   - Decomposition of Supercompilation
   - Coq Features Used

2. Supercompiler Organization and Correctness Proof
   - Expression Language and Simple Normalization
   - Propagation of Test Outcomes in Branches
   - Full Language, Loop Unrolling

3. Possible Extensions and Applications
   - Test Generation, Extensional Equivalence
   - More Realistic Language
   - Use Information Propagation in Isolation
Questions on the Title.

- **Supercompiler?**
- **Formal verification?**
  - Important for non-experimental supercompilers
  - Fresh look over supercompilation process
- **In Coq?**
  - A matter of taste
  - Non-critical (very few Coq-specific features used)
- **Simple?**
  - Toy language ...
  - ... over a toy data domain (simple binary trees).
  - Cut supercompilation into smaller pieces ...
  - ... with modular proofs of correctness.
  - But: less powerful supercompiler
Questions on the Title.

- Supercompiler?
- Formal verification?
  - Important for non-experimental supercompilers
  - Fresh look over supercompilation process
- In Coq?
  - A matter of taste
  - Non-critical (very few Coq-specific features used)
- Simple?
  - Toy language ...
  - ... over a toy data domain (simple binary trees).
  - Cut supercompilation into smaller pieces ...
  - ... with modular proofs of correctness.
  - But: less powerful supercompiler
Questions on the Title.

- Supercompiler?
- Formal verification?
  - Important for non-experimental supercompilers
  - Fresh look over supercompilation process
- In Coq?
  - A matter of taste
  - Non-critical (very few Coq-specific features used)
- Simple?
  - Toy language ...
  - ... over a toy data domain (simple binary trees).
  - Cut supercompilation into smaller pieces ...
  - ... with modular proofs of correctness.
  - But: less powerful supercompiler
Questions on the Title.

- Supercompiler?
- Formal verification?
  - Important for non-experimental supercompilers
  - Fresh look over supercompilation process
- In Coq?
  - A matter of taste
  - Non-critical (very few Coq-specific features used)
- Simple?
  - Toy language ...
  - ... over a toy data domain (simple binary trees).
  - Cut supercompilation into smaller pieces ...
  - ... with modular proofs of correctness.
  - But: less powerful supercompiler
Questions on the Title.

- Supercompiler?
- Formal verification?
  - Important for non-experimental supercompilers
  - Fresh look over supercompilation process
- In Coq?
  - A matter of taste
  - Non-critical (very few Coq-specific features used)
- Simple?
  - Toy language ...
  - ... over a toy data domain (simple binary trees).
  - Cut supercompilation into smaller pieces ...
  - ... with modular proofs of correctness.
  - But: less powerful supercompiler
Outline

1. Introduction
   - Questions on the Title
   - Decomposition of Supercompilation
   - Coq Features Used

2. Supercompiler Organization and Correctness Proof
   - Expression Language and Simple Normalization
   - Propagation of Test Outcomes in Branches
   - Full Language, Loop Unrolling

3. Possible Extensions and Applications
   - Test Generation, Extensional Equivalence
   - More Realistic Language
   - Use Information Propagation in Isolation
Classical Organization of Supercompilation

- Apply driving step (simplifications, information propagation, call unfolding, ...)
- Whistle?
  - Yes: Fold
  - No: Postprocess
- Fold?
  - Yes: Fold
  - No: Generalize
- N/A: Postprocess
- Finish
Decomposition of Supercompilation (this work).

- **Simple normalization** (≈ deforestation – unfolding) - \text{normConv}

  Example term := IfNil Id Id (Tl # Hd).
  Eval compute in (ntrm2trm (normConv (term $ term))).
  = IfNil Id (IfNil Id Id (Tl # Hd)) (Hd # Tl) : Trm

  Theorem normConvPreservesEval: forall (t: Trm) (v: Val),
  evalNT (normConv t) v = evalT t v.

- **Propagation of test outcomes inside if-branches** - \text{norm}

  Eval compute in (ntrm2trm (norm (term $ term))).
  = IfNil Id Nil (Hd # Tl) : Trm

  Theorem normPreservesEval: forall t v,
  evalNT (norm t) v = evalT t v.

- **Single-step loop unrolling** - unrollToInit

- **Ensuring termination** - firstEmbedded

- **Multi-step loop unrolling** - sscp
Decomposition of Supercompilation (this work).

- **Simple normalization** (≈ deforestation – unfolding) - normConv
  
  Example term := IfNil Id Id (Tl # Hd).
  Eval compute in (ntrm2trm (normConv (term $ term))).
  = IfNil Id (IfNil Id Id (Tl # Hd)) (Hd # Tl) : Trm
  
  Theorem normConvPreservesEval: forall (t: Trm) (v: Val),
  evalNT (normConv t) v = evalT t v.

- **Propagation of test outcomes inside if-branches** - norm
  
  Eval compute in (ntrm2trm (norm (term $ term))).
  = IfNil Id Nil (Hd # Tl) : Trm
  
  Theorem normPreservesEval: forall t v,
  evalNT (norm t) v = evalT t v.

- **Single-step loop unrolling** - unrollToInit
- **Ensuring termination** - firstEmbedded
- **Multi-step loop unrolling** - sscp
Decomposition of Supercompilation (this work).

- **Simple normalization (≈ deforestation – unfolding)** - normConv
  Example term := IfNil Id Id (Tl # Hd).
  Eval compute in (ntrm2trm (normConv (term $ term))).
  = IfNil Id (IfNil Id Id (Tl # Hd)) (Hd # Tl) : Trm
  Theorem normConvPreservesEval: forall (t: Trm) (v: Val),
  evalNT (normConv t) v = evalT t v.

- **Propagation of test outcomes inside if-branches** - norm
  Eval compute in (ntrm2trm (norm (term $ term))).
  = IfNil Id Nil (Hd # Tl) : Trm
  Theorem normPreservesEval: forall t v,
  evalNT (norm t) v = evalT t v.

- **Single-step loop unrolling** - unrollToInit
- **Ensuring termination** - firstEmbedded
- **Multi-step loop unrolling** - sscp
Outline

1. Introduction
   - Questions on the Title
   - Decomposition of Supercompilation
   - Coq Features Used

2. Supercompiler Organization and Correctness Proof
   - Expression Language and Simple Normalization
   - Propagation of Test Outcomes in Branches
   - Full Language, Loop Unrolling

3. Possible Extensions and Applications
   - Test Generation, Extensional Equivalence
   - More Realistic Language
   - Use Information Propagation in Isolation

Dimitur Krustev
A Simple Supercompiler Formally Verified in Coq
Coq Features Used.

- Coq: Proof assistant based on CoC + inductive types
- Simplified point of view:
  - Total(!) functional programming language
  - Inductive datatypes (Inductive ... := ... | ... .)
  - Pattern matching
    (match ... with ... => ... | ... end)
  - Structural recursion (top-level - Fixpoint, local - fix)
  - Lambda-functions (fun ... => ...)
- Interactive proofs in intuitionistic logic
  - The usual logical quantifiers/connectives
    forall, exists, ->, \/, \/, ~, <->
  - Interactive tactics for proofs by induction, rewriting, etc.
- Not used: dependent types(!), co-induction, classical logic
Coq Examples.

- Inductive datatypes
  

- Definitions by structural recursion
  
  Fixpoint power (n m : nat) {struct m} : nat :=
  
  match m with
  | 0 => 1 | S m1 => n * power n m1
  
  end.

  Eval compute in (power 2 5).
  
  = 32 : nat

- Partial evaluation
  
  Eval cbv beta iota delta -[mult] in
  
  (fun n => power n 3).

  = fun n : nat => n * (n * (n * 1)) : nat -> nat
Expression Sublanguage – Syntax.

- **Data domain**: simple binary trees (S-expressions with 1 atom)
  
  \[
  \text{Inductive Val: Set := | VNil: Val} \\
  \quad | \text{VCons: Val -> Val -> Val | VBottom: Val.}
  \]

- **Expression language**: tree constructors and selectors, identity, function composition, if-expressions

  \[
  \begin{align*}
  \text{Inductive Selector: Set := | HD | TL.} \\
  \text{Inductive Trm: Set := | Nil: Trm} \\
  \quad | \text{Cons: Trm -> Trm -> Trm | Sel: Selector -> Trm} \\
  \quad | \text{Id: Trm | Cmp: Trm -> Trm -> Trm} \\
  \quad | \text{IfNil: Trm -> Trm -> Trm -> Trm | Bottom.}
  \end{align*}
  \]

  Infix "$" := Cmp (at level 60, right associativity).

  Notation Hd := (Sel HD). Notation Tl := (Sel TL).

  Infix "#" := Cons (at level 62, right associativity).
Definition evalSel (sel: Selector) (v: Val) : Val :=
  match v with | VCons v1 v2 =>
    match sel with | HD => v1 | TL => v2 end
  | _ => VBottom
end.

Fixpoint evalT (t: Trm) (v: Val) {struct t} : Val :=
  match t with
  | Nil => VNil | Bottom => VBottom
  | Cons t1 t2 => VCons (evalT t1 v) (evalT t2 v)
  | Sel sel => evalSel sel v | Id => v
  | Cmp t1 t2 => evalT t1 (evalT t2 v)
  | IfNil t1 t2 t3 => match evalT t1 v with
    | VNil => evalT t2 v | VCons _ _ => evalT t3 v
    | VBottom => VBottom
  end
end.
Simple Normalization.

- Based on simplifications like:
  - \( \text{Cmp} \ \text{Hd} \ (\text{Cons} \ x \ y) \approx x \)
  - \( \text{IfNil} \ (\text{IfNil} \ x \ y \ z) \ u \ v \approx \text{IfNil} \ x \ \text{IfNil} \ y \ u \ v \ (\text{IfNil} \ z \ u \ v) \)

- Produces terms in normal form:
  \[
  \text{Inductive} \ \text{NTrm}: \ \text{Set} := \\
  | \ \text{NNil}: \ \text{NTrm} | \ \text{NCons}: \ \text{NTrm} \rightarrow \ \text{NTrm} \rightarrow \ \text{NTrm} \\
  | \ \text{NSelCmp}: \ \text{list} \ \text{Selector} \rightarrow \ \text{NTrm} \\
  | \ \text{NIfNil}: \ \text{list} \ \text{Selector} \rightarrow \ \text{NTrm} \rightarrow \ \text{NTrm} \rightarrow \ \text{NTrm} \\
  | \ \text{NBottom}: \ \text{NTrm}.
  \]

- ... which can injected back into full-blown terms:
  \[
  \text{Fixpoint} \ \text{ntrm2trm} \ (\text{nt}: \ \text{NTrm}) : \ \text{Trm} := \ldots \\
  \text{Definition} \ \text{evalNT} \ \text{nt} \ \text{v} := \ \text{evalT} \ (\text{ntrm2trm} \ \text{nt}) \ \text{v}.
  \]

- Structurally-recursive implementation:
  \[
  \text{Fixpoint} \ \text{normConv} \ (\text{t}: \ \text{Trm}) : \ \text{NTrm} := \ldots
  \]
Simple Normalization – Correctness.

- Tricky point - no full function composition in normal forms, yet we can still compose them:

  Definition normNCmp : NTrm -> NTrm -> NTrm := ...
  Lemma normNCmpPreservesEval: forall nt1 nt2 v, evalNT (normNCmp nt1 nt2) v = evalNT nt1 (evalNT nt2 v).

- With the help of some other (simpler) lemmas like:

  Lemma normSelsNCmpPreservesEvalT: forall sels nt v, evalT (ntrm2trm (normSelsNCmp sels nt)) v = evalSels sels (evalT (ntrm2trm nt) v).
  Lemma normNCmpIfIf: forall sels1 sels2 nt1_1 nt1_2 nt2_1 nt2_2, let nt1 := NIfNil sels1 nt1_1 nt1_2 in normNCmp nt1 (NIfNil sels2 nt2_1 nt2_2) = NIfNil sels2 (normNCmp nt1 nt2_1) (normNCmp nt1 nt2_2).

- ... we can establish correctness of simple normalization:

  Theorem normConvPreservesEval: forall t v, evalNT (normConv t) v = evalT t v.
Poor-man Explicit Substitutions.

- Primitives for pairing and function composition give us:
  - Variable-free programming
  - Simple form of explicit substitutions

- Example: `IfNil x1 x2 x3` has 3 free variables.
  - Pack them into an input tree: `x1 # x2 # x3`
  - Replace the original expression with:
    \[
    IfNil \text{Hd} \ ((\text{Hd} \ $ \ Tl) \ (Tl \ $ \ Tl))
    \]

- Computing object-level representations of substitutions:
  \[
  \text{replaceAt} \ (\text{pos}: \ \text{list} \ \text{Selector}) \ (t \ tr: \ NTrm): \ NTrm
  \]

- Now, we can represent and apply the substitution of `Nil` for `x2` in the above expression:
  \[
  \text{let } nt1 := \text{normConv} \ (\text{IfNil} \ \text{Hd} \ ((\text{Hd} \ $ \ Tl) \ (Tl \ $ \ Tl))) \ \text{in}
  \text{let } nt2 := \text{normConv} \ \text{Nil} \ \text{in}
  \text{let } x2p := \text{TL}::\text{HD}::\text{nil} \ \text{in}
  \text{ntrm2trm} \ (\text{normNCmp} \ nt1 \ (\text{replaceAt} \ x2p \ (\text{normConv} \ \text{Id}) \ nt2)))
  \]
  \[
  = \text{IfNil} \ \text{Hd} \ \text{Nil} \ (Tl \ $ \ Tl)
  \]

Dimitur Krustev

A Simple Supercompiler Formally Verified in Coq
Internalize propagation of if-condition outcome:

```coq
Definition setNilAt (sels: list Selector): NTrm :=
  replaceAt sels (NSelCmp nil) NNil.

Definition setConsAt (sels: list Selector): NTrm :=
  replaceAt sels (NSelCmp nil)
    (NCons (NSelCmp (sels ++ HD::nil))
    (NSelCmp (sels ++ TL::nil))).

Fixpoint propagateIfCond (nt: NTrm) {struct nt} : NTrm :=
  ...
  | NIfNil sels nt1 nt2 =>
    let nt1a := propagateIfCond nt1 in
    let nt2a := propagateIfCond nt2 in
    let nt1b := normNCmp nt1a (setNilAt sels) in
    let nt2b := normNCmp nt2a (setConsAt sels) in
    NIfNil sels nt1b nt2b
  ...
```
Propagation of Test Outcomes – Correctness.

- Test outcome propagation on top of simple normalization
  - Easier to give structurally recursive (total) definition
  - Easier to prove correctness on top of normalization correctness proof

Theorem propagateIfCondPreservesEval: \(\forall nt \; v,\) \(\text{evalNT} (\text{propagateIfCond} nt) \; v = \text{evalNT} nt \; v.\)

Definition norm \((t: \text{Trm})\) :=
\(\text{propagateIfCond} (\text{normConv} \; t).\)

Theorem normPreservesEval: \(\forall t \; v,\) \(\text{evalNT} (\text{norm} \; t) \; v = \text{evalT} t \; v.\)
Outline

1. Introduction
   - Questions on the Title
   - Decomposition of Supercompilation
   - Coq Features Used

2. Supercompiler Organization and Correctness Proof
   - Expression Language and Simple Normalization
   - Propagation of Test Outcomes in Branches
     - Full Language, Loop Unrolling

3. Possible Extensions and Applications
   - Test Generation, Extensional Equivalence
   - More Realistic Language
   - Use Information Propagation in Isolation

Dimitur Krustev
A Simple Supercompiler Formally Verified in Coq
SWhile Language.

- Expression language - not Turing-complete
- Embed in simple imperative language (“SWhile”) with:
  - while-loops
  - single (implicit) variable

Inductive SWhileStmt: Set :=
  | Assign: Trm -> SWhileStmt
  | Seq: SWhileStmt -> SWhileStmt -> SWhileStmt
  | While: Trm -> SWhileStmt -> SWhileStmt.

Infix ";;" := Seq (at level 65, right associativity).
Notation "'VAR' '←' e" := (Assign e) (at level 64).
Notation "'WHILE' cond 'DO' body 'DONE'" := (While cond body)

- Further simplification - single while-loop (analog to Kleene normal forms in recursion theory)

VAR <- initExp knf;
WHILE condExp knf DO VAR <- bodyExp knf DONE;
VAR <- finalExp knf
SWhile Language – Semantics.

“SWhile” semantics in Coq?

- inductive relations (elegant, non-executable)
- or, a “folk” trick:
  - replace a partial function \( f : X \to Y \)
  - with a total function \( f' : \text{nat} \to X \to \text{option } Y \),
    where
    - \( f' \ d \ x = \text{Some } y \to f \ x = y \) (\( f \ x \) is defined)
    - \( f' \ d \ x = \text{None} \) means \( f \ x \) cannot be computed in
      “stack depth” \( d \)
    - \( f' \) is structurally recursive on \( d \)

- Total “quasi-interpreter” for single-loop programs:

  Definition evalKNF (d: nat) (knf: KNFProg) (v: Val)
  : option Val := ...

Dimitur Krustev
A Simple Supercompiler Formally Verified in Coq
Loop Unrolling

- Analog to call unfolding in “SWhile”: loop unrolling
- Only a simple form of (single-step) unrolling considered – replace:

```
VAR  <- initExp knf;
WHILE condExp knf DO VAR  <- bodyExp knf DONE;
VAR  <- finalExp knf
```

with:

```
VAR  <- ntrm2trm (norm
    (IfNil (condExp knf) Id (bodyExp knf) $ initExp knf));
WHILE condExp knf DO VAR  <- bodyExp knf DONE;
VAR  <- finalExp knf
```

- Process tree replaced by a stream of repeated unrollings
Final Supercompiler, Correctness.

• “Whistle” – the usual one: homeomorphic embedding

• No need for folding, generalization in this (over-)simplified setting

• Final supercompiler

Definition sscp ... (n : nat) (knf : KNFProg) : option KNFProg := ...

• Correctness: a) Totality (using Kruskal's Tree Theorem as an axiom)

  Theorem sscp_total: forall b knf, exists n, exists knf1, sscp b n knf = Some knf1.

... b) Preservation of semantics

  Theorem sscp_correct: forall b knf knf1 n v1 v2, strictTrm (condExp knf) -> sscp b n knf = Some knf1 ->
  ((exists d1, evalKNF d1 knf v1 = Some v2) <->
  (exists d2, evalKNF d2 knf1 v1 = Some v2)).

Dimitur Krustev
Consider the usual Lisp-like encoding of lists and booleans as S-expressions (\texttt{WFalse := Nil, WTrue := Nil \# Nil}, etc.)

A program to check if the input list contains \texttt{WFalse}:
\begin{verbatim}
VAR <- Id \# WFalse; \{VAR = input \# output\}
WHILE Hd DO
  VAR <- IfNil (Hd $ Hd) (Nil \# WTrue) (Tl $ Hd \# Tl)
DONE;
VAR <- Tl \{VAR = output\}
\end{verbatim}

Its specialized version – non-empty input list prepended with its negated head:
\begin{verbatim}
Definition listHasWFalse_knf_negdupHd :=
  let negate x := IfNil x WTrue WFalse in
  modifyKNFinput listHasWFalse_knf
  (IfNil Id Id (negate Hd \# Id)).
\end{verbatim}
Example of Supercompilation (cont.)

- Result of supercompiling the specialized version:

  ```plaintext
  VAR <- IfNil Id (Nil # WFalse)
  (IfNil Hd (Nil # WTrue) (Nil # WTrue));
  WHILE Hd DO 
  VAR  <- IfNil (Hd $ Hd) (Nil # WTrue) (Tl $ Hd # Tl)
  DONE; VAR  <- Tl
  ```

- ... and with superfluous `IfNil` removed further by hand:

  ```plaintext
  VAR <- IfNil Id (Nil # WFalse) (Nil # WTrue);
  WHILE Hd DO 
  VAR  <- IfNil (Hd $ Hd) (Nil # WTrue) (Tl $ Hd # Tl)
  DONE; VAR  <- Tl
  ```

- Loop still here but a simple static post-processing could remove it
We can define a downsized version of the supercompiler, without information propagation: $sscp'$

Its result on the example:

\[
\begin{align*}
\text{VAR} & \leftarrow \text{IfNil } \text{Id} \\
& \left(\text{IfNil } \text{Id} \left(\text{IfNil } \text{Id} \left(\text{IfNil } \text{Hd} \left(\text{Nil } \# \text{ Nil} \right) \text{ Nil } \# \text{ Id} \right) \# \text{ Nil} \right) \\
& \left(\text{IfNil } \text{Id} \left(\text{IfNil } \text{Hd} \left(\text{Nil } \# \text{ Nil } \# \text{ Nil} \right) \left(\text{IfNil } \text{Id} \text{ Tl } \text{Id} \# \text{ Nil} \right) \right) \left(\text{IfNil } \text{Hd} \left(\text{IfNil } \text{Id} \text{ Tl } \text{Id} \# \text{ Nil} \right) \right) \right) \\
& \left(\text{IfNil } \text{Hd} \left(\text{IfNil } \text{Id} \text{ Tl } \text{Id} \# \text{ Nil} \right) \right) \left(\text{Nil } \# \text{ Nil } \# \text{ Nil} \right) \left(\text{Nil } \# \text{ Nil } \# \text{ Nil} \right) \right) \\
& \left(\text{Nil } \# \text{ Nil } \# \text{ Nil} \right) \\
\text{WHILE } \text{Hd} \\
\text{DO } \text{VAR} & \leftarrow \text{IfNil } \left(\text{Hd } \$ \text{ Hd} \right) \left(\text{Nil } \# \text{ Nil } \# \text{ Nil} \right) \left(\text{Tl } \$ \text{ Hd } \# \text{ Tl} \right) \text{ DONE;} \\
\text{VAR} & \leftarrow \text{Tl}
\end{align*}
\]
Outline

1. Introduction
   - Questions on the Title
   - Decomposition of Supercompilation
   - Coq Features Used

2. Supercompiler Organization and Correctness Proof
   - Expression Language and Simple Normalization
   - Propagation of Test Outcomes in Branches
   - Full Language, Loop Unrolling

3. Possible Extensions and Applications
   - Test Generation, Extensional Equivalence
   - More Realistic Language
   - Use Information Propagation in Isolation

Dimitur Krustev
A Simple Supercompiler Formally Verified in Coq
Normalization can simplify test generation

Inductive NTrm : Set :=
    | NNil : NTrm |
    | NCons : NTrm -> NTrm -> NTrm |
    | NSelCmp : list Selector -> NTrm |
    | NIfNil : list Selector -> NTrm -> NTrm -> NTrm |
    | NBottom : NTrm.

Idea: expressions can extract information from input tree only through selector compositions

- max. length of selector compositions = N
- ⇒ Expression cannot look deeper than N inside input tree
- Trees of depth ≤ N should suffice as tests

Finite tests sets ⇒ extensional equivalence decidable
Outline

1. Introduction
   - Questions on the Title
   - Decomposition of Supercompilation
   - Coq Features Used

2. Supercompiler Organization and Correctness Proof
   - Expression Language and Simple Normalization
   - Propagation of Test Outcomes in Branches
   - Full Language, Loop Unrolling

3. Possible Extensions and Applications
   - Test Generation, Extensional Equivalence
   - More Realistic Language
   - Use Information Propagation in Isolation
More Realistic Language

- More powerful forms of loop unrolling?
- Add function calls to expression language

\[
\text{Inductive Trm: Set :=}
\]

... |
\[
| \text{Ref: FunRef -> Trm.}
\]

- It becomes Turing-complete
- Still possible to:
  - Isolate simple normalization, and information propagation
  - Implement them by structural recursion

- Complications:
  - How to specify semantics in Coq?
  - Normal forms - slightly more complicated
  - We need folding and generalization now
  - Termination proof of full supercompiler with generalization –
    more complicated(?)
Outline

1. Introduction
   - Questions on the Title
   - Decomposition of Supercompilation
   - Coq Features Used

2. Supercompiler Organization and Correctness Proof
   - Expression Language and Simple Normalization
   - Propagation of Test Outcomes in Branches
   - Full Language, Loop Unrolling

3. Possible Extensions and Applications
   - Test Generation, Extensional Equivalence
   - More Realistic Language
   - Use Information Propagation in Isolation
Apply (positive) information propagation in cases where we do need the full power of supercompilation (with its complications, like “whistle”, etc.)

In systems like Coq itself; example:

Fixpoint listHasFalse (l: list bool) : bool :=
  match l with |
  | nil => false |
  | false::_ => true |
  | true::l1 => listHasFalse l1 |
  end.

Goal forall b l, listHasFalse (b::negb b::l) = true.
  compute. fold listHasFalse.

...forall (b : bool) (l : list bool), (if b then
  if if b then false else true then listHasFalse l
  else true else true) = true

Strengthen stream fusion?

...
Summary

- First formal verification of a supercompiler.
- Helped by a more fine-grained decomposition of the supercompilation process.
  - Structurally recursive deforestation and information propagation, with separate proofs.
  - Simple form of explicit substitutions also helpful.

Outlook

- Extend to more realistic languages, more powerful transformations.
- Applications to test generation, compiler optimizations.
- Some day: self-verifiable supercompiler