Staged multi-result supercompilation
(Filtering by transformation)

Sergei A. Grechanik    Ilya G. Klyuchnikov    Sergei A. Romanenko

Keldysh Institute of Applied Mathematics, Moscow

June 30, 2014 – Pereslavl-Zalessky
1. Problem solving by multi-result supercompilation
2. Staged mrsc: multiple results represented as a residual program
3. Pushing filtering over generation of graphs
4. Pushing filtering over the whistle
5. An executable model of multi-result supercompilation in Agda
6. Conclusions
Outline

1. Problem solving by multi-result supercompilation
2. Staged mrsc: multiple results represented as a residual program
3. Pushing filtering over generation of graphs
4. Pushing filtering over the whistle
5. An executable model of multi-result supercompilation in Agda
6. Conclusions
A popular approach to problem solving is *trial and error*:

- Generate alternatives.
- Evaluate alternatives.
- Select the best alternatives.

Using a multi-result supercompiler `mrsc` and a filter `filter` we get a “problem solver”

$$\text{solver} = \text{filter} \circ \text{mrsc}$$

Thus

- Instead of trying to guess, which variant is ”the best” one, we *produce* a collection of residual graphs: $g_1, g_2, \ldots, g_k$.
- And then *filter* this collection according to some criteria.
What is good and what is bad

Design:

\[ \text{solver} = \text{filter} \circ \text{mrsc} \]

**Good:** this design is modular and gives a clear separation of concerns.

- \text{mrsc} is a general-purpose tool.
- \text{filter} incorporates some knowledge about the problem domain.
- \text{mrsc} knows nothing about the problem domain.
- \text{filter} knows nothing about supercompilation.

**Bad:** the process is time and space consuming.

- \text{mrsc} can produce millions of residual graphs!
Exploiting monotonicity of filters

Monotonicity:

- If some parts of a partially constructed residual graph are "bad", then the completed residual graph is also certain to be a "bad" one.

A solution: fusing filtering and constructing.

\[ \text{solver'} = \text{fuse filter mrsc} \]


Bad:

- Fusion destroys modularity.
- Every time filter is modified, the fusion of mrsc and filter has to be repeated.
1. Problem solving by multi-result supercompilation

2. Staged mrsc: multiple results represented as a residual program

3. Pushing filtering over generation of graphs

4. Pushing filtering over the whistle

5. An executable model of multi-result supercompilation in Agda

6. Conclusions
A “naive” multi-result supercompiler is decomposed into 2 stages:

\[ \text{naive-mrsc} \circ \langle \_ \rangle \circ \text{lazy-mrsc} \]

Or, given an initial configuration \( c \),

\[ \text{naive-mrsc} \ c = \langle \langle \text{lazy-mrsc} \ c \rangle \rangle \]

where \( \text{lazy-mrsc} \) generates a compact representation of a set of graphs, which is then interpreted by \( \langle \langle \_ \rangle \rangle \), to actually produce graphs of configurations.

**Extensional equality**

\[ f \equiv g \] means that \( \forall x \rightarrow f \ x = g \ x \)

**Mixfix notation (Agda)**

\[ \text{if\_then\_else\_} \ p \ x \ y \] is equivalent to \( \text{if} \ p \ \text{then} \ x \ \text{else} \ y \)
The way to staged mrsc

This is achieved by the following steps.

- Replacing the original small-step supercompiler mrsc with a big-step supercompiler naive-mrsc.
- Identifying some operations in naive-mrsc related to the calculation of Cartesian products.
- Rewriting naive-mrsc into lazy-mrsc, which, instead of calculating Cartesian products immediately, outputs requests for \( \langle \_ \rangle \) to calculate them at the second stage.
Staging: big-step $\rightarrow$ small-step

- **Small-step** supercompilation: rewriting the graph of configuration step-by-step.
  \[
  g_0 \rightarrow g_1 \rightarrow \ldots \rightarrow g_n \rightarrow g
  \]
  (This is a generalization of small-step operational semantics.)

- **Big-step** supercompilation: building the subgraphs and then building the graph.
  \[
  g = \text{build}(g_1, g_2, \ldots, g_k)
  \]
  (This is a generalization of big-step, or “natural” operational semantics.)

The MRSC Toolkit implements small-step multi-result supercompilation:

Staging: delaying Cartesian products

At some places, naive-mrsc calculates ”Cartesian products”.

- Suppose, a graph \( g \) is to be constructed from \( k \) subgraphs \( g_1, \ldots, g_k \).
- naive-mrsc computes \( k \) sets of graphs \( gs_1, \ldots, gs_k \).
- And then considers all possible \( g_i \in gs_i \) for \( i = 1, \ldots, k \) and constructs corresponding versions of the graph \( g = \text{build}(g_1, \ldots, g_k) \).

lazy-mrsc generates a ”lazy graph”, which, essentially, is a ”program” to be ”executed” by \( \langle \_ \rangle \).

Unlike naive-mrsc, lazy-mrsc does not calculate Cartesian products immediately: instead, it outputs requests for \( \langle \_ \rangle \) to calculate them at the second stage.

Thus

\[
\text{naive-mrsc} = \langle \_ \rangle \circ \text{lazy-mrsc}
\]
Outline

1. Problem solving by multi-result supercompilation
2. Staged mrsc: multiple results represented as a residual program
3. Pushing filtering over generation of graphs
4. Pushing filtering over the whistle
5. An executable model of multi-result supercompilation in Agda
6. Conclusions
First generating, then filtering

Let $c$ be the initial configuration. We can produce and filter the collection of graphs in two ways.

1. By the direct generation of the collection of graphs, followed by filtering:
   
   $gs = \text{filter (naive-mrsc } c\text{)}$

2. By generating a compact representation of the collection of graphs, followed by the generation of graphs, followed by filtering:
   
   $gs = \text{filter } ⟨⟨ \text{lazy-mrsc } c ⟩⟩$

A problem

In both cases the selection of best solutions is done by generating all the graphs. And there may be, millions…

Conclusion

Compact representation for collections of graphs seems to be of no use.
First filtering, then generating. Is it possible?

What about pushing \texttt{filter} over $\langle\_\rangle$?

**Definition**

A function \texttt{clean} is a \texttt{cleaner} of lazy graphs if for a lazy graph $l$

$$\langle\langle \text{clean } l \rangle\rangle \subseteq \langle\langle l \rangle\rangle$$

Given a filter \texttt{filter}, let \texttt{clean} be a cleaner, such that

$$\text{filter} \circ \langle\_\rangle \equiv \langle\_\rangle \circ \text{clean}$$

Then

$$\text{filter} \circ \text{mrsc} \equiv$$

$$\text{filter} \circ \langle\_\rangle \circ \text{lazy-mrsc} \equiv$$

$$\langle\_\rangle \circ \text{clean} \circ \text{lazy-mrsc}$$

**Conclusion**

Now cleaning is done \texttt{before} the actual generation of graphs.
Some cleaners of practical importance

It is easy to implement cleaners that perform the following tasks.

- Removing subtrees that represent empty sets of graphs.
- Removes subtrees that contain “bad” configurations.
- Selecting subtrees of minimal size.

The above cleaners produce results in linear time.
What are the advantages?

- The construction is **modular**: `lazy-mrsc` and `⟨⟨ _ ⟩⟩` do not have to know anything about filtering, while `clean` does not have to know anything about `lazy-mrsc` and `⟨⟨ _ ⟩⟩`.
- Cleaners are **composable**: we can decompose a sophisticated cleaner into a composition of simpler cleaners.
- In many cases (of practical importance) cleaners can be implemented in such a way that the best graphs can be extracted from a lazy graph **in linear time**.
Outline

1. Problem solving by multi-result supercompilation
2. Staged mrsc: multiple results represented as a residual program
3. Pushing filtering over generation of graphs
4. Pushing filtering over the whistle
5. An executable model of multi-result supercompilation in Agda
6. Conclusions
Decomposing lazy-mrsc

Using codata and corecursion, we can decompose lazy-mrsc:

\[
\text{lazy-mrsc} \equiv \text{prune-cograph} \circ \text{build-cograph}
\]

where

- **build-cograph** constructs a (potentially) infinite tree.
- **prune-cograph** traverses this tree and turns it into a (finite) lazy graph.

**Modularity:**

- **build-cograph** uses driving and rebuilding. Knows nothing about the whistle.
- **prune-cograph** uses the whistle. Knows nothing about driving and rebuilding.
Suppose that
\[
\text{clean} \circ \text{prune-cograph} \equiv \text{prune-cograph} \circ \text{clean}_\infty
\]
where
- \text{clean} is a lazy graph cleaner.
- \text{clean}_\infty a co-graph cleaner.

Then
\[
\text{clean} \circ \text{lazy-mrsc} \equiv \\
\text{clean} \circ \text{prune-cograph} \circ \text{build-cograph} \equiv \\
\text{prune-cograph} \circ \text{clean}_\infty \circ \text{build-cograph}
\]

A good thing
\text{build-cograph} and \text{clean}_\infty work in a lazy way, generating subtrees by demand!
What are the advantages?

Evaluating

\[ \langle\langle \text{prune-cograph (clean}\infty (\text{build-cograph } c)) \rangle\rangle \]

is likely to be less time and space consuming than directly evaluating

\[ \langle\langle \text{clean (lazy-mrsc } c) \rangle\rangle \]

A cograph cleaner working in linear time:

- Removing subtrees that contain “bad” configurations.
Outline

1. Problem solving by multi-result supercompilation
2. Staged mrsc: multiple results represented as a residual program
3. Pushing filtering over generation of graphs
4. Pushing filtering over the whistle
5. An executable model of multi-result supercompilation in Agda
6. Conclusions
An (executable) model of big-step multi-result supercompilation in Agda

The project in Agda

https://github.com/sergei-romanenko/staged-mrsc-agda

What is implemented?

- An abstract model in Agda of big-step multi-result supercompilation. A formal proof is given of the fact that
  \[ \forall (c : \text{Conf}) \rightarrow \text{naive-mrsc } c \equiv \langle \text{lazy-mrsc } c \rangle \]

- The abstract model is instantiated to produce a multi-result supercompiler for counter systems.


- The MRSC Toolkit is a generic framework.
- No means for formulating properties supercompilers and/or proving their correctness.

- The first formally verified supercompiler.
- A specific supercompiler for a specific language.


- This framework is generic.
- It formalizes “traditional” single-result supercompilation.
Graphs of configurations

data Graph (C : Set) : Set where
  back : ∀ (c : C) → Graph C
  forth : ∀ (c : C) (gs : List (Graph C)) → Graph C

- We abstract away from the concrete structure of configurations.
- Arrows in the graph carry no information (if needed, this information can be kept inside “configurations”).
- back c means that c is foldable to (at least one) parent configuration.

Forth-nodes are produced by

- decomposing a configuration into a number of other configurations (e.g. by driving), or
- by rewriting a configuration by another one (e.g. by generalization, or applying a lemma during two-level supercompilation).
record ScWorld : Set₁ where
  field
    Conf : Set
    _⊆_ : (c c' : Conf) → Set
    _⊆ʔ_ : (c c' : Conf) → Dec (c ⊆ c')
    _⇒_ : (c : Conf) → List (List Conf)

whistle : BarWhistle Conf
...

- **Conf** is the type of “configurations”.
- _⊆_ is a “foldability relation”. c ⊆ c' means that c is foldable to c'.
- _⊆ʔ_ is a decision procedure for _⊆_. This procedure is necessary for implementing supercompilation in functional form.
- _⇒_ produces possible decompositions of a configuration. Let cs ∈ (c ⇒). Then c can be decomposed into configurations cs.
- **whistle** is used to ensure termination of functional supercompilation.
record ScWorld : Set₁ where
...

History : Set
History = List Conf

Foldable : ∀ (h : History) (c : Conf) → Set
Foldable h c = Any (_□_ c) h

foldable? : ∀ (h : History) (c : Conf) → Dec (Foldable h c)
foldable? h c = Any.any (_□?_ c) h

- **History** is the list of configurations on the path to the current one.
- **Foldable** \( h \ c \) means that \( c \) is foldable to a configuration in \( h \).
- **foldable?** \( h \ c \) decides whether **Foldable** \( h \ c \).
This means that the graph $g$ can be produced by supercompiling the configuration $c$ with respect to the history $h$.

This means that $\text{length } cs = \text{length } gs$, and each $g \in gs$ can be produced by supercompiling the corresponding $c \in cs$. 

$h \vdash \text{NDSC}_* cs \leftrightarrow gs = \text{Pointwise.Rel } (\vdash \text{NDSC}_-\rightarrow_- h) cs gs$
data \_ \vdash \text{NDSC} \_ \leftrightarrow \_ \text{ where }

\text{ndsc-fold} : \forall \{h : \text{History}\} \{c\}
(f : \text{Foldable } h \ c) \rightarrow
h \vdash \text{NDSC } c \leftrightarrow \text{back } c

\text{ndsc-build} : \forall \{h : \text{History}\} \{c\}
\{cs : \text{List Conf}\} \{gs : \text{List (Graph Conf)}\}
(\neg f : \neg \text{Foldable } h \ c)
(i : cs \in c \Rightarrow) (s : (c :: h) \vdash \text{NDSC* } cs \leftrightarrow gs) \rightarrow
h \vdash \text{NDSC } c \leftrightarrow \text{forth } c \ gs

- \textbf{ndsc-fold}: if c is foldable, let us fold.
- \textbf{ndsc-build}: if c is not foldable, let us build a subtree by selecting a cs \in c \Rightarrow and supercompiling each c \in cs, to produce a list of subgraphs gs.
data _⊢MRSC_⟵_ where

mrsc-fold : ∀ {h : History} {c}
     (f : Foldable h c) →
     h ⊢MRSC c⟵ back c

mrsc-build : ∀ {h : History} {c}
     {cs : List Conf} {gs : List (Graph Conf)}
     (¬f : ¬ Foldable h c) (¬w : ¬ ⊦ h)
     (i : cs ∈ c ⇒) (s : (c :: h) ⊢MRSC* cs⟵ gs) →
     h ⊢MRSC c⟵ forth c gs

• mrsc-fold: if c is foldable, let us fold.
• mrsc-build: if c is not foldable and the history h is not dangerous
  (¬ ⊦ h), let us build a subtree by selecting a cs ∈ c ⇒ and
  supercompiling each c ∈ cs, to produce a list of subgraphs gs.
A relational specification of **small-step single-result** supercompilation was suggested by Klimov.


Klyuchnikov used a supercompilation relation for proving the correctness of a small-step single-result supercompiler for a higher-order functional language.


- We consider a supercompilation relation for a **big-step multi-result** supercompilation.
The supercompiler *naive-mrsc* is a (generic) total function:

\[
\text{naive-mrsc : } (c : \text{Conf}) \rightarrow \text{List} (\text{Graph Conf})
\]

such that

\[
\vdash \text{MRSC} \leftrightarrow \text{naive-mrsc} : \\
\{c : \text{Conf}\} \{g : \text{Graph Conf}\} \rightarrow \\
[] \vdash \text{MRSC} \ c \leftrightarrow g \leftrightarrow g \in \text{naive-mrsc} \ c
\]

- The termination of *naive-mrsc* is guaranteed by a *whistle*.
- In our model of big-step supercompilation whistles are assumed to be “inductive bars”. See

http://www.cairn.info/revue-internationale-de-philosophie-2004-4-page-483.htm
What is a bar?

Bar $D \ h$ means that “$D$ is a bar for the sequence $h$”.

data Bar \{A : Set\} \ (D : \text{List} \ A \rightarrow \text{Set} ) : 
    \( (h : \text{List} \ A ) \rightarrow \text{Set} \) where

now : \{h : \text{List} \ A \} \ (bz : D \ h) \rightarrow \text{Bar} \ D \ h
later : \{h : \text{List} \ A \} 
    \quad (bs : \forall \ c \rightarrow \text{Bar} \ D \ (c :: h)) 
    \rightarrow \text{Bar} \ D \ h

If Bar $D \ h$, then either

- $D \ h$ is valid right now (i.e. $h$ is “dangerous”).
- Or, for all possible $c$ there holds Bar $D \ (c :: h)$ (i.e. any continuation of $h$ eventually becomes “dangerous”).

Bar induction

\( \forall \ D \ h \rightarrow \text{Bar} \ D \ [] \rightarrow \text{Bar} \ D \ h \)
A bar whistle is a record

record BarWhistle (A : Set) : Set₁ where
  field
    ⊤ : (h : List A) → Set
    ⊤:: : (c : A) (h : List A) → ⊤ h → ⊤ (c :: h)
    ⊤? : (h : List A) → Dec (⊤ h)

bar[] : Bar ⊤ []

- ⊤ h means h is dangerous.
- ⊤:: postulates that if h is dangerous, so are all continuations of h.
- ⊤? says that ⊤ is decidable.
- bar[] says that any sequence eventually becomes dangerous. (In Coquand’s terms, Bar ⊤ is required to be “an inductive bar”.)
Computing Cartesian products

The functional specification of big-step multi-result supercompilation is based on the function \texttt{cartesian}:

\[
\text{cartesian} : \forall \{A : \text{Set}\} \\
(xss : \text{List (List } A)) \rightarrow \text{List (List } A)
\]

Namely, suppose that \(xss\) has the form

\[
xs_1 :: xs_2 :: \ldots :: xs_k
\]

Then \texttt{cartesian} returns all possible lists of the form

\[
x_1 :: x_2 :: \ldots :: x_k :: []
\]

where \(x_i \in xs_i\). In Agda this is formulated as follows:

\[
\in*\leftrightarrow\in\text{cartesian} : \\
\forall \{A : \text{Set}\} \{xs : \text{List } A\} \{yss : \text{List (List } A)\} \rightarrow \text{Pointwise.Rel } _\in_ \text{xs yss} \leftrightarrow \text{xs } \in \text{cartesian yss}
\]
naive-mrsc is defined in terms of a more general function naive-mrsc'.

naive-mrsc : (c : Conf) → List (Graph Conf)
naive-mrsc' : ∀ (h : History) (b : Bar ⊩ h) (c : Conf) → List (Graph Conf)

naive-mrsc c = naive-mrsc' [] bar[] c

naive-mrsc' takes 2 additional arguments:

- h is a history.
- b is a proof b of the fact Bar ⊩ h.

naive-mrsc has to supply a proof of the fact Bar ⊩ []. But this proof is supplied by the whistle!
Now comes the definition of \texttt{naive-mrsc}'::

\begin{verbatim}
naive-mrsc' h b c with foldable? h c
... | yes f = [ back c ]
... | no ~f with ? h
... | yes w = []
... | no ~w with b
... | now bz with ~w bz
... | ()
naive-mrsc' h b c | no ~f | no ~w | later bs =
    map (forth c)(concat (map (cartesian o
        map (naive-mrsc' (c :: h) (bs c))) (c \rightarrow)))
\end{verbatim}

- For each $c' \in cs \in (c \rightarrow)$ there is recursively called
  \texttt{naive-mrsc'} (c :: h) (bs c) c' to produce a list of graphs $gs'$.
- \texttt{cartesian} selects a graph $g' \in gs'$ from each $gs'$.
Why \textit{naive-mrsc}' passes the termination check?

The problem with ‘naive-mrsc’ is that in the recursive call

\begin{align*}
\text{naive-mrsc}' (c :: h) (bs \, c) \, c'
\end{align*}

- \( h \) is replaced with \( c :: h \) (which is \textcolor{red}{\textit{bigger}} than \( h \)).
- \( c \) is replaced with \( c' \) (whose size is \textcolor{red}{\textit{unknown}}).

Hence, \( h \) and \( c \) do not become “structurally smaller”.

However,

- \((\text{\textcolor{red}{\textit{later}} bs}) \) becomes \((bs \, c)\).
- Agda’s termination checker considers \( bs \) and \((bs \, c)\) to be “of the same size”.

Therefore

\( (bs \, c) \) is “structurally smaller” than \((\text{\textcolor{red}{\textit{later}} bs})\)!
Big-step supercompilation was studied and implemented by Bolingbroke and Peyton Jones.


- We deal with multi-result supercompilation, rather than single-result supercompilation.
- Our big-step supercompilation constructs graphs of configurations in an explicit way, because the graphs are going to be filtered and/or analyzed at a later stage.
- We consider not only the functional formulation of big-step supercompilation, but also the relational one.
Now we decompose \texttt{naive-mrsc} into two stages

\[
\texttt{naive-mrsc} \equiv \langle \_ \rangle \circ \texttt{lazy-mrsc}
\]

where

- \texttt{lazy-mrsc} produces a “lazy graph”, which is a “residual program”.
- \texttt{\langle \_ \rangle} is an interpreter that executes a lazy graph to actually produce a list of graphs of configurations.

\[
\texttt{lazy-mrsc} : (c : \text{Conf}) \rightarrow \text{LazyGraph Config}
\]

\[
\texttt{\langle \_ \rangle} : \{C : \text{Set}\} (l : \text{LazyGraph C}) \rightarrow \text{List (Graph C)}
\]
A lazy graph is a program whose nodes are commands.

```haskell
data LazyGraph (C : Set) : Set where
  Ø      : LazyGraph C
  stop   : (c : C) → LazyGraph C
  build  : (c : C)
             (lss : List (List (LazyGraph C))) → LazyGraph C
```

- Ø. Generate the empty list of graphs.
- `stop`. Generate a back-node `back c` and stop.
- `build c lss`. Consider all `ls ∈ lss`. Let `ls` has the form `l₁ :: l₂ :: ... :: lₖ :: []`. Execute each `lᵢ` to produce a list of graphs `gss = gs₁ :: gs₂ :: ... :: gsₖ :: []`. By evaluating `cartesian gss`, generate all `gs' = g₁ :: g₂ :: ... :: gₖ :: []`, where `gᵢ ∈ gsᵢ`, and build all `build c gs'`. 
Interpreting lazy graphs

\[ \langle - \rangle : \{ C : \text{Set} \} (l : \text{LazyGraph } C) \rightarrow \text{List } \langle \text{Graph } C \rangle \]

\[ \langle - \rangle^* : \{ C : \text{Set} \} (ls : \text{List } \langle \text{LazyGraph } C \rangle) \rightarrow \text{List } \langle \text{List } \langle \text{Graph } C \rangle \rangle \]

\[ \langle - \rangle \Rightarrow : \{ C : \text{Set} \} (lss : \text{List } \langle \text{List } \langle \text{LazyGraph } C \rangle \rangle) \rightarrow \text{List } \langle \text{List } \langle \text{Graph } C \rangle \rangle \]

\[ \langle [ ] \rangle^* = [ ] \]

\[ \langle l :: ls \rangle^* = \langle l \rangle :: \langle ls \rangle^* \]

\[ \langle [ ] \rangle \Rightarrow = [ ] \]

\[ \langle ls :: lss \rangle \Rightarrow = \text{cartesian } \langle ls \rangle^* ++ \langle lss \rangle \Rightarrow \]

\[ \langle \emptyset \rangle = [ ] \]

\[ \langle \text{stop } c \rangle = [ \text{back } c ] \]

\[ \langle \text{build } c \ lss \rangle = \text{map } \langle \text{forth } c \rangle \langle lss \rangle \Rightarrow \]
Generating lazy graphs 1/2

**lazy-mrsc** is defined in terms of a more general function **lazy-mrsc’**:

\[
\text{lazy-mrsc} : (c : \text{Conf}) \rightarrow \text{LazyGraph Conf} \\
\text{lazy-mrsc’} : \forall (h : \text{History}) (b : \text{Bar} \not\subseteq h) (c : \text{Conf}) \rightarrow \text{LazyGraph Conf}
\]

\[
\text{lazy-mrsc} \ c = \text{lazy-mrsc’} \ [] \ \text{bar[]} \ c
\]

**An idea**

**lazy-mrsc** can be derived from **naive-mrsc** by replacing the call to **cartesian** with the constructor **build**.
Generating lazy graphs 2/2

```plaintext
lazy-mrsc' h b c with foldable? h c
... | yes f = stop c
... | no ¬f with ¥? h
... | yes w = Ø
... | no ¬w with b
... | now bz with ¬w bz
... | ()
lazy-mrsc' h b c | no ¬f | no ¬w | later bs =
  build c (map (map (lazy-mrsc' (c :: h) (bs c))) (c ⊢))
```

Why this is good?

**cartesian** is not called! Thus, there is no combinatorial explosion.

There is a formal proof in Agda of the theorem

```
naive≡lazy : (c : Conf) → naive-mrsc c ≡ ⟨⟨ lazy-mrsc c ⟩⟩
```
The idea to use a compact representation for collections of residual graphs is due to Grechanik.


- As a matter of fact, the data structure **LazyGraph C** formalizes the idea of “overtrees” described in the above paper.
- We show that “lazy graphs” arise in a natural way as a result of staging a big-step multi-result supercompiler and, essentially, are residual “programs” that record graph-building operations delayed at the first stage.
Filtering vs cleaning

What is a filter?
\[ \text{filter } gs \subseteq gs \]

What is a cleaner?
\[ \langle \langle \text{clean } l \rangle \rangle \subseteq \langle \langle l \rangle \rangle \]

Correctness of a cleaner with respect to a filter
\[ \text{filter } \circ \langle \langle \_ \rangle \rangle \equiv \langle \langle \_ \rangle \rangle \circ \text{clean} \]

Pushing filtering over generation
\[ \text{filter } \circ \text{naive-mrsc} \equiv \langle \langle \_ \rangle \rangle \circ \text{clean} \circ \text{lazy-mrsc} \]
Removing graphs with bad configurations (filtering)

A graph is “bad” if it contains a bad configuration.

\[
\begin{align*}
\text{bad-graph} &: \{C : \text{Set}\} (\text{bad} : C \rightarrow \text{Bool}) \\
&(g : \text{Graph } C) \rightarrow \text{Bool} \\
\text{bad-graph*} &: \{C : \text{Set}\} (\text{bad} : C \rightarrow \text{Bool}) \\
&(\text{gs} : \text{List } (\text{Graph } C)) \rightarrow \text{Bool} \\
\text{bad-graph bad (back c)} &= \text{bad c} \\
\text{bad-graph bad (forth c gs)} &= \text{bad c }\lor \text{ bad-graph* bad gs} \\
\text{bad-graph* bad []} &= \text{false} \\
\text{bad-graph* bad (g :: gs)} &= \text{bad-graph bad g }\lor \text{ bad-graph* bad gs} \\
\end{align*}
\]

\[
\begin{align*}
\text{fl-bad-conf} &: \{C : \text{Set}\} (\text{bad} : C \rightarrow \text{Bool}) \\
&(\text{gs} : \text{List } (\text{Graph } C)) \rightarrow \text{List } (\text{Graph } C) \\
\text{fl-bad-conf bad gs} &= \text{filter (not }\circ\text{ bad-graph bad)} \text{ gs}
\end{align*}
\]
Removing graphs with bad configuration (cleaning) 1/2

\[
\text{cl-bad-conf} : \{C : \text{Set}\} (\text{bad} : C \rightarrow \text{Bool})  \\
(\mathit{l} : \text{LazyGraph } C) \rightarrow \text{LazyGraph } C
\]

\[
\text{cl-bad-conf*} : \{C : \text{Set}\} (\text{bad} : C \rightarrow \text{Bool})  \\
(\mathit{ls} : \text{List (LazyGraph } C)) \rightarrow \text{List (LazyGraph } C)
\]

\[
\text{cl-bad-conf} \Rightarrow : \{C : \text{Set}\} (\text{bad} : C \rightarrow \text{Bool})  \\
(\mathit{lss} : \text{List (List (LazyGraph } C))) \rightarrow \text{List (List (LazyGraph } C))
\]

\[
\text{cl-bad-conf* } \text{bad } \text{[]} = \text{[]}
\]

\[
\text{cl-bad-conf* } \text{bad } (\mathit{l} :: \mathit{ls}) =
\]

\[
\text{cl-bad-conf } \text{bad } \mathit{l} :: \text{cl-bad-conf* } \text{bad } \mathit{ls}
\]

\[
\text{cl-bad-conf} \Rightarrow \text{bad } \text{[]} = \text{[]}
\]

\[
\text{cl-bad-conf} \Rightarrow \text{bad } (\mathit{ls} :: \mathit{lss}) =
\]

\[
\text{cl-bad-conf* } \text{bad } \mathit{ls} :: (\text{cl-bad-conf} \Rightarrow \text{bad } \mathit{lss})
\]
Removing graphs with bad configuration (cleaning) 2/2

\[
\begin{align*}
\text{cl-bad-conf bad } \emptyset &= \emptyset \\
\text{cl-bad-conf bad (stop c)} &= \\
&\quad \text{if bad c then } \emptyset \text{ else (stop c)} \\
\text{cl-bad-conf bad (build c lss)} &= \\
&\quad \text{if bad c then } \emptyset \text{ else (build c (cl-bad-conf} \Rightarrow \text{ bad lss))}
\end{align*}
\]

A “metasystem transition”...

Instead of removing bad graphs, we remove parts of the program that would generate bad graphs!

Correctness

\[
\text{cl-bad-conf-correct} : \{C : \text{Set}\} (\text{bad} : C \rightarrow \text{Bool}) \rightarrow
\langle \_ \rangle \circ \text{cl-bad-conf bad} \triangleq \text{fl-bad-conf bad} \circ \langle \_ \rangle
\]
Decomposing lazy-mrsc

By using codata and corecursion, we can decompose lazy-mrsc:

\[ \text{lazy-mrsc} \triangleq \text{prune-cograph} \circ \text{build-cograph} \]

- **build-cograph** constructs a (potentially) infinite tree (a LazyCograph).
- **prune-cograph** traverses this tree and turns it into a (finite) LazyGraph C.

LazyCograph C differs from LazyGraph C only in \( \infty \) in the type of lss.

```haskell
data LazyCograph (C : Set) : Set where
  Ø : LazyCograph C
  stop : (c : C) → LazyCograph C
  build : (c : C) (lss : \(\infty\)(List (List (LazyCograph C)))) → LazyCograph C
```
**Building lazy cographs**

`build-cograph` can be derived from the function `lazy-mrsc` by removing the machinery related to whistles.

```haskell
build-cograph' h c with foldable? h c
... | yes f = stop c
... | no ¬f =
    build c (♯ build-cograph ⇒ h c (c ⇒))
```

```haskell
build-cograph* h [] = []
build-cograph* h (c :: cs) =
    build-cograph' h c :: build-cograph* h cs
```

```haskell
build-cograph ⇒ h c [] = []
build-cograph ⇒ h c (cs :: css) =
    build-cograph* (c :: h) cs :: build-cograph ⇒ h c css
```

```haskell
build-cograph c = build-cograph' [] c
```
Pruning lazy cographs

`prune-cograph` can be derived from `lazy-mrsc` by removing the machinery related to generation of nodes.

```
prune-cograph' h b Ø = Ø
prune-cograph' h b (stop c) = stop c
prune-cograph' h b (build c lss) with ⌊? h
  ... | yes w = Ø
  ... | no ¬w with b
  ... | now bz with ¬w bz
  ... | ()
prune-cograph' h b (build c lss) | no ¬w | later bs =
  build c (map (prune-cograph* (c :: h) (bs c)) (♭ lss))
```

```
prune-cograph* h b [] = []
prune-cograph* h b (l :: ls) =
  prune-cograph' h b l :: (prune-cograph* h b ls)

prune-cograph l = prune-cograph' [] bar[] l
```
Suppose \( \text{clean}^{\infty} \) is a cograph cleaner such that

\[
\text{clean} \circ \text{prune-cograph} \equiv \text{prune-cograph} \circ \text{clean}^{\infty}
\]

then

\[
\text{clean} \circ \text{lazy-mrsc} \equiv
\]

\[
\text{clean} \circ \text{prune-cograph} \circ \text{build-cograph} \equiv
\]

\[
\text{prune-cograph} \circ \text{clean}^{\infty} \circ \text{build-cograph}
\]

**What is good?**

\text{build-cograph} and \( \text{clean}^{\infty} \) work in a lazy way, generating subtrees by demand.
Removing cographs with bad configurations

\[
\text{cl-bad-conf} \infty \text{ bad } \varnothing = \\
\varnothing \\
\text{cl-bad-conf} \infty \text{ bad } (\text{stop } c) = \\
\quad \text{if bad } c \text{ then } \varnothing \text{ else } (\text{stop } c) \\
\text{cl-bad-conf} \infty \text{ bad } (\text{build } c \ lss) \text{ with bad } c \\
\quad \text{| true } = \varnothing \\
\quad \text{| false } = \text{build } c \ (\# (\text{cl-bad-conf} \infty \Rightarrow \text{bad } (\downarrow lss)))
\]

\[
\text{cl-bad-conf} \infty^* \text{ bad } [] = [] \\
\text{cl-bad-conf} \infty^* \text{ bad } (l :: ls) = \\
\quad (\text{cl-bad-conf} \infty \text{ bad } l) :: \text{cl-bad-conf} \infty^* \text{ bad } ls
\]

\[
\text{cl-bad-conf} \infty \Rightarrow \text{bad } [] = [] \\
\text{cl-bad-conf} \infty \Rightarrow \text{bad } (ls :: lss) = \\
\quad \text{cl-bad-conf} \infty^* \text{ bad } ls :: (\text{cl-bad-conf} \infty \Rightarrow \text{bad } lss)
\]
Outline

1. Problem solving by multi-result supercompilation
2. Staged mrsc: multiple results represented as a residual program
3. Pushing filtering over generation of graphs
4. Pushing filtering over the whistle
5. An executable model of multi-result supercompilation in Agda
6. Conclusions
Conclusions

- Big-step multi-result supercompilation can be decomposed into **two stages**. The result of the first stage (a “lazy graph”) is interpreted at the second stage, to produce a collection of residual graphs.

- A lazy graph is a **compact representation** for a collection of residual graphs (and can be regarded as a “program”).

- Filtering a collection of graphs can be replaced with **cleaning a lazy graph**. In some cases of practical importance, cleaning can be performed in linear time.

- By using codata and corecursion, the generator of lazy graphs can be decomposed into two stages: **building** an infinite tree and **pruning** this tree to produce a (finite) lazy graph.

- Some cleaners of lazy graphs can be turned into **cleaners of cographs**, so that cleaning can be pushed over the whistle.