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

Outline

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

Problem solving by multi-result supercompilation

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"

$$solver = filter \circ mrsc$$

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:

$$solver = filter \circ mrsc$$

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

- `mrsc` is a general-purpose tool.
- `filter` incorporates some knowledge about the problem domain.
- `mrsc` knows nothing about the problem domain.
- `filter` knows nothing about supercompilation.

Bad: the process is time and space consuming.

- `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.

$solver' = fuse\ filter\ mrsc$

Andrei V. Klimov, Ilya G. Klyuchnikov, Sergei A. Romanenko. **Automatic Verification of Counter Systems via Domain-Specific Multi-Result Supercompilation.** In Third International Valentin Turchin Workshop on Metacomputation (Proceedings of the Third International Valentin Turchin Workshop on Metacomputation. Pereslavl-Zalessky, Russia, July 5-9, 2012). A.V. Klimov and S.A. Romanenko, Ed. - Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2012, 260 p. ISBN 978-5-901795-28-6, pages 112-141.

Bad:

- Fusion destroys modularity.
- Every time filter is modified, the fusion of mrsc and filter has to be repeated.

Outline

Staged mrsc: multiple results represented as a residual program

A "naive" multi-result supercompiler is decomposed into 2 stages:

$$\texttt{naive-mrsc} \doteq \langle\!\langle \_ \rangle\!\rangle \circ \texttt{lazy-mrsc}$$

Or, given an initial configuration c,

$$\texttt{naive-mrsc c} = \langle\!\langle \texttt{ lazy-mrsc c } \rangle\!\rangle$$

where `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

$\texttt{f} \doteq \texttt{g}$ means that $\forall\ \texttt{x} \rightarrow \texttt{f x = g x}$

Mixfix notation (Agda)

`if_then_else_ p x y` is equivalent to `if p then x 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 $\langle\langle \_ \rangle\rangle$ 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:

Ilya G. Klyuchnikov, Sergei A. Romanenko. **Formalizing and Implementing Multi-Result Supercompilation.** In Third International Valentin Turchin Workshop on Metacomputation (Proceedings of the Third International Valentin Turchin Workshop on Metacomputation. Pereslavl-Zalessky, Russia, July 5-9, 2012). A.V. Klimov and S.A. Romanenko, Ed. - Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2012, 260 p. ISBN 978-5-901795-28-6, pages 142-164.

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 $g = \mathrm{build}(g_1, \ldots, g_k)$.

`lazy-mrsc` generates a "lazy graph", which, essentially, is a "program" to be "executed" by $\langle\!\langle \_ \rangle\!\rangle$.

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

Thus

$$\texttt{naive-mrsc} \doteq \langle\!\langle \_ \rangle\!\rangle \circ \texttt{lazy-mrsc}$$

Outline

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 = filter (naive-mrsc c)`

2. By generating a compact representation of the collection of graphs, followed by the generation of graphs, followed by filtering:

   `gs = filter ⟪ 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 filter over $\langle\!\langle \_ \rangle\!\rangle$?

Definition

A function clean is a cleaner of lazy graphs if for a lazy graph l

$$\langle\!\langle \texttt{clean l} \rangle\!\rangle \subseteq \langle\!\langle \texttt{l} \rangle\!\rangle$$

Given a filter filter, let clean be a cleaner, such that

$$\texttt{filter} \circ \langle\!\langle \_ \rangle\!\rangle \doteq \langle\!\langle \_ \rangle\!\rangle \circ \texttt{clean}$$

Then

$$\begin{aligned} &\texttt{filter} \circ \texttt{mrsc} \doteq \\ &\quad \texttt{filter} \circ \langle\!\langle \_ \rangle\!\rangle \circ \texttt{lazy-mrsc} \doteq \\ &\quad\quad \langle\!\langle \_ \rangle\!\rangle \circ \texttt{clean} \circ \texttt{lazy-mrsc} \end{aligned}$$

Conclusion

Now cleaning is done 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 $\langle\langle \_ \rangle\rangle$ do not have to know anything about filtering, while `clean` does not have to know anything about `lazy-mrsc` and $\langle\langle \_ \rangle\rangle$.
- 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

Decomposing *lazy-mrsc*

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

$$\texttt{lazy-mrsc} \mathrel{\stackrel{\circ}{=}} \texttt{prune-cograph} \circ \texttt{build-cograph}$$

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.

Cleaning before whistling

Suppose that

$$\texttt{clean} \circ \texttt{prune-cograph} \doteq \texttt{prune-cograph} \circ \texttt{clean}\infty$$

where

- `clean` is a lazy graph cleaner.
- `clean∞` a cograph cleaner.

Then

$$\begin{aligned}&\texttt{clean} \circ \texttt{lazy-mrsc} \doteq \\ &\quad \texttt{clean} \circ \texttt{prune-cograph} \circ \texttt{build-cograph} \doteq \\ &\quad \texttt{prune-cograph} \circ \texttt{clean}\infty \circ \texttt{build-cograph}\end{aligned}$$

A good thing

`build-cograph` and `clean∞` work in a lazy way, generating subtrees by demand!

What are the advantages?

Evaluating

$\langle\langle$ `prune-cograph (clean∞ (build-cograph c))` $\rangle\rangle$

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

$\langle\langle$ `clean (lazy-mrsc c)` $\rangle\rangle$

A cograph cleaner working in linear time:

- Removing subtrees that contain "bad" configurations.

Outline

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

  $$\forall\ (\mathtt{c} : \mathtt{Conf}) \to \mathtt{naive\text{-}mrsc}\ \mathtt{c} \equiv \langle\!\langle\ \mathtt{lazy\text{-}mrsc}\ \mathtt{c}\ \rangle\!\rangle$$

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

Related works 1/2

Ilya G. Klyuchnikov and Sergei A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute of Applied Mathematics, 2011. URL: http://library.keldysh.ru/preprint.asp?lg=e&id=2011-77.

Ilya G. Klyuchnikov, Sergei A. Romanenko. Formalizing and Implementing Multi-Result Supercompilation. In *Third International Valentin Turchin Workshop on Metacomputation (Proceedings of the Third International Valentin Turchin Workshop on Metacomputation. Pereslavl-Zalessky, Russia, July 5-9, 2012)*. A.V. Klimov and S.A. Romanenko, Ed. - Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2012, 260 p. ISBN 978-5-901795-28-6, pages 142-164.

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

Related works 2/2

Dimitur N. Krustev. **A simple supercompiler formally verified in Coq.** In *A. P. Nemytykh, editor, Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia, Pereslavl-Zalessky, Russia, July 1–5, 2010*, pages 102–127. Ailamazyan University of Pereslavl, Pereslavl-Zalessky, 2010.

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

Dimitur N. Krustev. *Towards a Framework for Building Formally Verified Supercompilers in Coq.* In *Proceedings of the 13th International Symposium on Trends in Functional Programming (TFP 2012), St Andrews, UK, June 12-14, 2012.* Lecture Notes in Computer Science Volume 7829, 2013, pp 133–148.

- 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).

“Worlds” of supercompilation 1/2

```
record ScWorld : Set₁ where
  field
    Conf : Set
    _⊑_ : (c c′ : Conf) → Set
    _⊑?_ : (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′`.
- `_⊑?_` 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.

“Worlds” of supercompilation 2/2

```
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`.

Relational big-step non-deterministic supercompilation 1/2

```
h ⊢NDSC c ↪ g
```

This means that the graph g can be produced by supercompiling the configuration c with respect to the history h.

```
data _⊢NDSC_↪_ : ∀ (h : History) (c : Conf)
  (g : Graph Conf) → Set
```

```
h ⊢NDSC* cs ↪ gs
```

This means that length cs = length gs, and each g ∈ gs can be produced by supercompiling the corresponding c ∈ cs.

```
_⊢NDSC*_↪_ : ∀ (h : History) (cs : List Conf)
  (gs : List (Graph Conf)) → Set

h ⊢NDSC* cs ↪ gs = Pointwise.Rel (_⊢NDSC_↪_ h) cs gs
```

Relational big-step non-deterministic supercompilation 2/2

```
data _⊢NDSC_↪_ where

  ndsc-fold  : ∀ {h : History} {c}
    (f : Foldable h c) →
    h ⊢NDSC c ↪ back c

  ndsc-build : ∀ {h : History} {c}
    {cs : List Conf} {gs : List (Graph Conf)}
    (¬f : ¬ Foldable h c)
    (i : cs ∈ c ⇉) (s : (c :: h) ⊢NDSC* cs ↪ gs) →
    h ⊢NDSC c ↪ forth c gs
```

- `ndsc-fold`: if `c` is foldable, let us fold.
- `ndsc-build`: if `c` is not foldable, let us build a subtree by selecting a `cs ∈ c ⇉` and supercompiling each `c ∈ cs`, to produce a list of subgraphs `gs`.

Relational big-step multi-result supercompilation 2/2

```
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`.

Related works

A relational specification of small-step single-result supercompilation was suggested by Klimov.

Andrei V. Klimov. **A program specialization relation based on supercompilation and its properties.** In *First International Workshop on Metacomputation in Russia (Proceedings of the first International Workshop on Metacomputation in Russia. Pereslavl-Zalessky, Russia, July 2–5, 2008)*. A. P. Nemytykh, Ed. - Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008, 108 p. ISBN 978-5-901795-12-5, pages 54–77.

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

Ilya G. Klyuchnikov. **Supercompiler HOSC: proof of correctness.** Preprint 31. Keldysh Institute of Applied Mathematics, Moscow. 2010. URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2010-31`

- We consider a supercompilation relation for a big-step multi-result supercompilation.

Supercompilers as total functions

The supercompiler `naive-mrsc` is a (generic) total function:

```
naive-mrsc : (c : Conf) → List (Graph Conf)
```

such that

```
  ⊢MRSC↪⇔naive-mrsc :
    {c : Conf} {g : Graph Conf} →
     [] ⊢MRSC c ↪ g ⇔ g ∈ 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

Thierry Coquand. **About Brouwer's fan theorem.** September 23, 2003. Revue internationale de philosophie, 2004/4 n° 230, p. 483-489.
http://www.cairn.info/revue-internationale-de-philosophie-2004-4-page-483.htm
http://www.cairn.info/load_pdf.php?ID_ARTICLE=RIP_230_0483

What is a bar?

`Bar D h` means that "`D` is a bar for the sequence `h`".

```
data Bar {A : Set} (D : List A → Set) :
         (h : List A) → Set where
  now   : {h : List A} (bz : D h) → Bar D h
  later : {h : List A}
          (bs : ∀ c → Bar D (c :: h)) → 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

`∀ D h → Bar D [] → Bar D h`

Bar whistles

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 `cartesian`:

```
cartesian : ∀ {A : Set}
  (xss : List (List A)) → List (List A)
```

Namely, suppose that *xss* has the form

$$xs_1 :: xs_2 :: \ldots :: xs_k$$

Then `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:

```
∈*↔∈cartesian :
  ∀ {A : Set} {xs : List A} {yss : List (List A)} →
    Pointwise.Rel _∈_ xs yss ↔ xs ∈ cartesian yss
```

Functional big-step multi-result supercompilation 1/2

`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!

Functional big-step multi-result supercompilation 2/2

Now comes the definition of naive-mrsc′:

```
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 ∘
        map (naive-mrsc′ (c :: h) (bs c))) (c ⇉)))
```

- For each c′ ∈ cs ∈ (c ⇉) there is recursively called naive-mrsc′ (c :: h) (bs c) c′ to produce a list of graphs gs′.
- cartesian selects a graph g′ ∈ gs′ from each gs′.

Why *naive-mrsc′* passes the termination check?

The problem with 'naive-mrsc′' is that in the recursive call

```
naive-mrsc′ (c :: h) (bs c) c′
```

- `h` is replaced with `c :: h` (which is bigger than `h`).
- `c` is replaced with `c′` (whose size is unknown).

Hence, `h` and `c` do not become "structurally smaller".

However,

- `(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 `(later bs)`!

Related works

Big-step supercompilation was studied and implemented by Bolingbroke and Peyton Jones.

Maximilian C. Bolingbroke and Simon L. Peyton Jones. **Supercompilation by evaluation.** In *Proceedings of the third ACM Haskell symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 2010, pages 135–146. `http://doi.acm.org/10.1145/1863523.1863540`

- 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 naive-mrsc into two stages

```
naive-mrsc ≗ ⟪_⟫ ∘ lazy-mrsc
```

where

- lazy-mrsc produces a "lazy graph", which is a "residual program".
- ⟪_⟫ is an interpreter that executes a lazy graph to actually produce a list of graphs of configurations.

```
lazy-mrsc : (c : Conf) → LazyGraph Conf
⟪_⟫ : {C : Set} (l : LazyGraph C) → List (Graph C)
```

Lazy graphs of configuration

A *lazy graph* is a program whose nodes are commands.

```
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` $\in$ `lss`. Let `ls` has the form $l_1 :: l_2 :: \ldots :: l_k :: []$. Execute each l_i to produce a list of graphs $gss = gs_1 :: gs_2 :: \ldots :: gs_k :: []$. By evaluating `cartesian gss`, generate all $gs' = g_1 :: g_2 :: \ldots :: g_k :: []$, where $g_i \in gs_i$, and build all `build c gs'`.

Interpreting lazy graphs

```
⟪_⟫ : {C : Set} (l : LazyGraph C) → List (Graph C)
⟪_⟫* : {C : Set} (ls : List (LazyGraph C)) →
    List (List (Graph C))
⟪_⟫⇉ : {C : Set} (lss : List (List (LazyGraph C))) →
    List (List (Graph C))
```

```
⟪ [] ⟫* = []
⟪ l :: ls ⟫* = ⟪ l ⟫ :: ⟪ ls ⟫*

⟪ [] ⟫⇉ = []
⟪ ls :: lss ⟫⇉ = cartesian ⟪ ls ⟫* ++ ⟪ lss ⟫⇉

⟪ Ø ⟫ = []
⟪ stop c ⟫ = [ back c ]
⟪ build c lss ⟫ = map (forth c) ⟪ lss ⟫⇉
```

Generating lazy graphs 1/2

`lazy-mrsc` is defined in terms of a more general function `lazy-mrsc′`:

```
lazy-mrsc : (c : Conf) → LazyGraph Conf
lazy-mrsc′ : ∀ (h : History) (b : Bar ↯ h) (c : Conf) →
             LazyGraph Conf

lazy-mrsc c = lazy-mrsc′ [] 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

```
lazy-mrsc′ h b c with foldable? h c
... | yes f = stop c
... | no ¬f with ↯? h
... | yes w = ∅
... | 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 combinatory explosion.

There is a formal proof in Agda of the theorem

```
naive≡lazy : (c : Conf) → naive-mrsc c ≡ ⟪ lazy-mrsc c ⟫
```

Related works

The idea to use a compact representation for collections of residual graphs is due to Grechanik.

Sergei A. Grechanik. **Overgraph Representation for Multi-Result Supercompilation.** In *Third International Valentin Turchin Workshop on Metacomputation (Proceedings of the Third International Valentin Turchin Workshop on Metacomputation.* Pereslavl-Zalessky, Russia, July 5–9, 2012). A.V. Klimov and S.A. Romanenko, Ed. – Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2012, 260 p. ISBN 978-5-901795-28-6, pages 48–65.

- 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?

$\texttt{filter gs} \subseteq \texttt{gs}$

What is a cleaner?

$\langle\!\langle\ \texttt{clean l}\ \rangle\!\rangle \subseteq \langle\!\langle\ \texttt{l}\ \rangle\!\rangle$

Correctness of a cleaner with respect to a filter

$\texttt{filter} \circ \langle\!\langle\_\rangle\!\rangle \doteq \langle\!\langle\_\rangle\!\rangle \circ \texttt{clean}$

Pushing filtering over generation

$\texttt{filter} \circ \texttt{naive-mrsc} \doteq \langle\!\langle\_\rangle\!\rangle \circ \texttt{clean} \circ \texttt{lazy-mrsc}$

Removing graphs with bad configurations (filtering)

A graph is "bad" if it contains a bad configuration.

```
bad-graph : {C : Set} (bad : C → Bool)
    (g : Graph C) → Bool
bad-graph* : {C : Set} (bad : C → Bool)
    (gs : List (Graph C)) → Bool

bad-graph bad (back c) = bad c
bad-graph bad (forth c gs) = bad c ∨ bad-graph* bad gs

bad-graph* bad [] = false
bad-graph* bad (g :: gs) =
    bad-graph bad g ∨ bad-graph* bad gs
```

```
fl-bad-conf : {C : Set} (bad : C → Bool)
    (gs : List (Graph C)) → List (Graph C)

fl-bad-conf bad gs = filter (not ∘ bad-graph bad) gs
```

Removing graphs with bad configuration (cleaning) 1/2

```
cl-bad-conf : {C : Set} (bad : C → Bool)
  (l : LazyGraph C) → LazyGraph C
cl-bad-conf* : {C : Set} (bad : C → Bool)
  (ls : List (LazyGraph C)) → List (LazyGraph C)
cl-bad-conf⇉ : {C : Set} (bad : C → Bool)
  (lss : List (List (LazyGraph C))) →
    List (List (LazyGraph C))
```

```
cl-bad-conf* bad [] = []
cl-bad-conf* bad (l :: ls) =
  cl-bad-conf bad l :: cl-bad-conf* bad ls
```

```
cl-bad-conf⇉ bad [] = []
cl-bad-conf⇉ bad (ls :: lss) =
  cl-bad-conf* bad ls :: (cl-bad-conf⇉ bad lss)
```

Removing graphs with bad configuration (cleaning) 2/2

```
cl-bad-conf bad ∅ = ∅
cl-bad-conf bad (stop c) =
  if bad c then ∅ else (stop c)
cl-bad-conf bad (build c lss) =
  if bad c then ∅ else (build c (cl-bad-conf⇉ bad lss))
```

A "metasystem transition"...

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

Correctness

```
cl-bad-conf-correct : {C : Set} (bad : C → Bool) →
    ⟪_⟫ ∘ cl-bad-conf bad ≗ fl-bad-conf bad ∘ ⟪_⟫
```

Decomposing lazy-mrsc

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

$$\texttt{lazy-mrsc} \doteq \texttt{prune-cograph} \circ \texttt{build-cograph}$$

- `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 ∞ in the type of `lss`.

```
data LazyCograph (C : Set) : Set where
  Ø     : LazyCograph C
  stop  : (c : C) → LazyCograph C
  build : (c : C) (lss : ∞(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.

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

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

build-cograph⇉ h c [] = []
build-cograph⇉ h c (cs :: css) =
  build-cograph* (c :: h) cs :: build-cograph⇉ h c css

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 ∅ = ∅
prune-cograph′ h b (stop c) = stop c
prune-cograph′ h b (build c lss) with  ↯? h
... | yes w = ∅
... | 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
```

Promoting some cleaners over the whistle

Suppose $\texttt{clean}\infty$ is a cograph cleaner such that

$$\texttt{clean} \circ \texttt{prune-cograph} \doteq \texttt{prune-cograph} \circ \texttt{clean}\infty$$

then

$$\begin{aligned}&\texttt{clean} \circ \texttt{lazy-mrsc} \doteq \\ &\quad \texttt{clean} \circ \texttt{prune-cograph} \circ \texttt{build-cograph} \doteq \\ &\quad \texttt{prune-cograph} \circ \texttt{clean}\infty \circ \texttt{build-cograph}\end{aligned}$$

What is good?

$\texttt{build-cograph}$ and $\texttt{clean}\infty$ work in a lazy way, generating subtrees by demand.

Removing cographs with bad configurations

```
cl-bad-conf∞ bad ∅ =
  ∅
cl-bad-conf∞ bad (stop c) =
  if bad c then ∅ else (stop c)
cl-bad-conf∞ bad (build c lss) with bad c
... | true = ∅
... | false = build c (♯ (cl-bad-conf∞⇉ bad (♭ lss)))
```

```
cl-bad-conf∞* bad [] = []
cl-bad-conf∞* bad (l ∷ ls) =
  (cl-bad-conf∞ bad l) ∷ cl-bad-conf∞* bad ls

cl-bad-conf∞⇉ bad [] = []
cl-bad-conf∞⇉ bad (ls ∷ lss) =
  cl-bad-conf∞* bad ls ∷ (cl-bad-conf∞⇉ bad lss)
```

Outline

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.