Overgraph Representation for Multi-Result Supercompilation

Sergei Grechanik

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Meta 2012
General Idea of Multi-Resultness

We use **heuristics** to guess the best path.

And get a **single** residual program.

SC(P1)
General Idea of Multi-Resultness

We take (almost) every possible path

We get a set of residual programs

And then we choose the best one (optionally)
A problem

**Millions** of residual programs

A solution

**Overgraph** – a compact representation for sets of graphs
MRSC Toolkit Architecture

Core

Rules
- Whistle
- Generalization Strategy
- Folding Strategy
- Driving Rules
- ...

Rewriting Steps

Graph

Residualization

R
MRSC: Graphs of Configurations

- C1 (Root)
- C2
- C3
- C4
- C5
- C6
- C7
- C8 (Incomplete Nodes)
- C2' (Folding Edge)

Current Node: C8

Folding Edge: C2' to C8
MRSC: Graph Rewriting Steps

- Complete
- Fold
- AddChildNodes
- Rebuild
MRSC: Tree of Graphs
MRSC: Tree of Graphs

Depth-First Traversal of the Tree of Graphs
MRSC: Tree of Graphs

Depth-First Traversal of the Tree of Graphs
MRSC: Tree of Graphs

Depth-First Traversal of the Tree of Graphs
MRSC: Tree of Graphs

Depth-First Traversal of the Tree of Graphs
MRSC: Tree of Graphs

Depth-First Traversal of the Tree of Graphs
MRSC: Tree of Graphs

Depth-First Traversal of the Tree of Graphs
Combinatorial Explosion

Too many graphs

- Use some **heuristics**
- **Share** some parts of graphs

**Spaghetti Stack (MRSC)**
Do Spaghetti Stacks Solve the Problem?

Not entirely

These subtrees are likely to be equal but they won't be shared
Rules : Graph $\rightarrow$ [Step]

Rules transform graphs into rewriting steps

But usually they don't need the whole graph, just a path from the root to the current node.
Rules : Path → [Step]

- Let's try to restrict rules to work on paths

- We would lose an interesting ability to fold with cross edges

- We would need some new representation to make use of this new property
Overtree Representation

Let's combine all configuration trees into one big overtree

An overtree represents a set of trees

```
data Tree = Tree (F Tree)
data OTree = OTree [F OTree]
```
Do Overtrees Solve the Problem?

- They are a bit better, but still...

- We've already lost cross edges

- Are we going to lose folding edges completely?
Overgraph

- Let's just glue together nodes equivalent up to renaming

- Each configuration corresponds to no more than one node
Folding

We don't need special folding edges
Advantages and Problems

- Overgraphs are more compact
- Overgraphs are cleaner
  - One configuration — one node
  - No special folding edges
- Overgraphs contain more information
- Each node can have multiple parents
  - Can we use binary whistles?
  - How can we control generalization?
- How to apply rules?
- How to extract residual programs?
Hyperedges

- We will call **bundles of edges** hyperedges

- Hyperedges represent steps like driving and generalization

- Completion step can be represented as a hyperedge with **zero destination nodes**

- Incomplete nodes have no outgoing hyperedges
Supercompilation with Overgraphs

1) Overgraph **Construction**
   Add nodes and edges while possible

2) Overgraph **Truncation**
   Remove useless nodes and edges

3) **Residualization**
Overgraph Construction

- Rule: Configuration → [Step]
  - Add this node

- Rule: Overgraph → [Hyperedge]

In what order should we apply the rules?

$r$ is monotone if for all graphs $G$ and $H$:

$$G \subseteq H \Rightarrow r(G) \subseteq r(H)$$

If all rules are monotone we can apply them in any order.
Rules

- We can also write rules in this form:

  \[\text{precondition} \rightarrow \text{hyperedges to add}\]

- Examples:

  \[\neg \text{UnaryWhistle}(c) \rightarrow \text{drive}(c)\]
  \[\text{always} \quad \text{c} \rightarrow \text{generalize}(c)\]

  \[\text{min}_\text{depth}(c) < 42 \rightarrow \text{drive}(c)\]

  This precondition is monotone
Binary Whistles

¬ ∃ d ∈ G : BinaryWhistle(c,d)

\[ \text{c} \rightarrow \text{drive}(c) \]

NOT monotone

∃ path p from root to c :

∀ d ∈ p : ¬BinaryWhistle(c,d)

\[ \text{c} \rightarrow \text{drive}(c) \]

OK

This green path won't disappear
Overgraph Truncation

This incomplete node is useless

We should remove all incident hyperedges
Residualization

Building a full set of graphs should be avoided!

We will represent residual programs as trees with back edges (i.e. no subprogram sharing)
Naive Residualization Algorithm

Convert Overgraph into an Overtree and then convert it into a set of trees
Naive Residualization Algorithm

Convert Overgraph into an Overtree and then convert it into a set of trees
Suboptimality

Absolutely identical subtrees

Idea: Cache intermediate results
More Formal Definition

\[ R : \text{Node} \rightarrow [\text{Node}] \rightarrow [\text{Tree}] \]

\[ R \ n \ h \ | \ n \in h = [\text{Fold}(n)] \]
\[ R \ n \ h \ | \ \text{otherwise} = [n \rightarrow (r_1 \ldots r_k) \mid n \rightarrow (d_1 \ldots d_k) \in G, \ ri \in R \ di \ (n:h)] \]

\[ h = [4, 2, 1] \]
\[ n = 2 \]
More Formal Definition

\[
R : \text{Node} \rightarrow \text{[Node]} \rightarrow \text{[Tree]}
\]

\[
R \ n \ h \ | \ n \in h = [\text{Fold}(n)]
\]

\[
R \ n \ h \ | \ \text{otherwise} = [n \rightarrow (r_1 \ldots r_k) | n \rightarrow (d_1 \ldots d_k) \in G, \ ri \in R \ di \ (n:h)]
\]

h = [2, 1]
n = 4

R 2 [4, 2, 1]
History Structure

\[ R : \text{Node} \rightarrow [\text{Node}] \rightarrow [\text{Tree}] \]

- **Predecessors**
  - Can be in a history but cannot be folded against
  - These can influence folding

- **Successors**
  - Won't be in a history

- **Both**

- **History**
Enhanced Residualization

- Removing pure predecessors from history won't change the result
  \[ R \ n \ h = R \ n \ (h \ n \ succs(n)) \]

- Let's rewrite residualization algorithm this way:
  \[
  R \ n \ h \mid n \in h = [Fold(n)] \\
  R \ n \ h \mid \text{otherwise} = \\
  [n \rightarrow (r_1 \ldots r_k) \mid \\
  n \rightarrow (d_1 \ldots d_k) \in G, \\
  r_i \in R \ di \ (n:h \ n \ succs(di))] \\
  \]

- Now we can just apply memoization
Evaluation of Residualization Algorithms

- Caching improves performance

But the algorithms produce trees with back edges. Turned out it is not very useful for most tasks.
Example: Counter Systems

• The task is to find the minimal proof of a counter system's safety
• A proof is a graph, not a tree with back edges
• MRSC uses cross edges to simulate graphs
• But overgraphs may be still useful because they enable truncation
Experiment with Counter Systems

Rules

Core

Branch & Bound

VS

Rules

Overgraph Construction

Truncation

Core

Branch & Bound
Experimental Results

Improvement (times)

- DataRace
- ReaderWriter
- Java
- Xerox
- Firefly
- Berkley
- Illinois
- MOESI
- MESI
- MOSI
- MSI
- Synapse

(in terms of the number of visited nodes)
Why overgraphs were useful?

- We could compute **sets of successors**
- We could **truncate** an overgraph

An overgraph contains a lot of **information** about relations between configurations

This is even more important than its compactness
Further Work

• Experiments with subgraph-producing residualization algorithms
  - need graph-based language
  - tree-producing algorithm seems unsuitable for real-world tasks

• Searching for heuristics (whistles etc) useful for overgraph representation

• Applying overgraphs to higher-level supercompilation
Conclusions

We suggested the Overgraph representation

- An Overgraph is a very compact representation
- Rules, Whistles and Residualization were generalized to Overgraphs
- The implementation has shown its usefulness
  - Caching residualization algorithm
  - Truncation for counter systems
- Overgraph contains a lot of information, so it is possible to analyze multiple graphs at once
Please return to the previous slide
Correctness

• It is possible that not all of the trees extracted from an overgraph represent correct programs

• Usually it is not a problem for single-level supercompilation
Language used in experiments

- The language is essentially based on trees with back edges
  
  \[ Y (\lambda f \rightarrow ...) \]

- Higher order
- Explicit fixed point combinator
- No let-expressions
Overgraph vs E-PEG

• Essentially the same idea applied to different domains
• We work with functional languages, so we have a clear recursion rather than incomprehensible cycles
• We don't have symmetric equalities
• We decided to residualize to trees, they naturally “residualize” to graphs
  − Should we do the same?
There are no more slides