Optimizing Program Size Using Multi-result Supercompilation

Dimitur Krustev

IGE+XAO Balkan

IGE+XAO GROUP

27 March 2021 / VPT 2021

Outline

Introduction

- Supercompilation - a very general and powerful program transformation technique, invented by Turchin
- Advantages:
  - fully automatic
  - more powerful than most other similar techniques (partial evaluation, deforestation, . . .)
  - diverse potential applications (program optimization, program analysis and verification, . . .)
- Issues:
  - not powerful enough in certain cases; improvements possible (distillation, higher-level supercompilation, . . .)
  - unpredictable result size - code size explosion possible
  - unpredictable transformation time (related to previous issue)

Approach outline

- Tame unpredictable output program size, relying on:
  - supercompilation itself, in particular multi-result supercompilation
  - a generalization strategy explicitly tailored to avoid code explosion
  - a compact representation for the set of alternative configuration graphs produced by multi-result supercompilation
  - efficient filtering algorithms – based on the compact representation of graph sets – to select “interesting” results w.r.t. program size
- Evaluate approach on a number of small examples

Supercompilation Overview

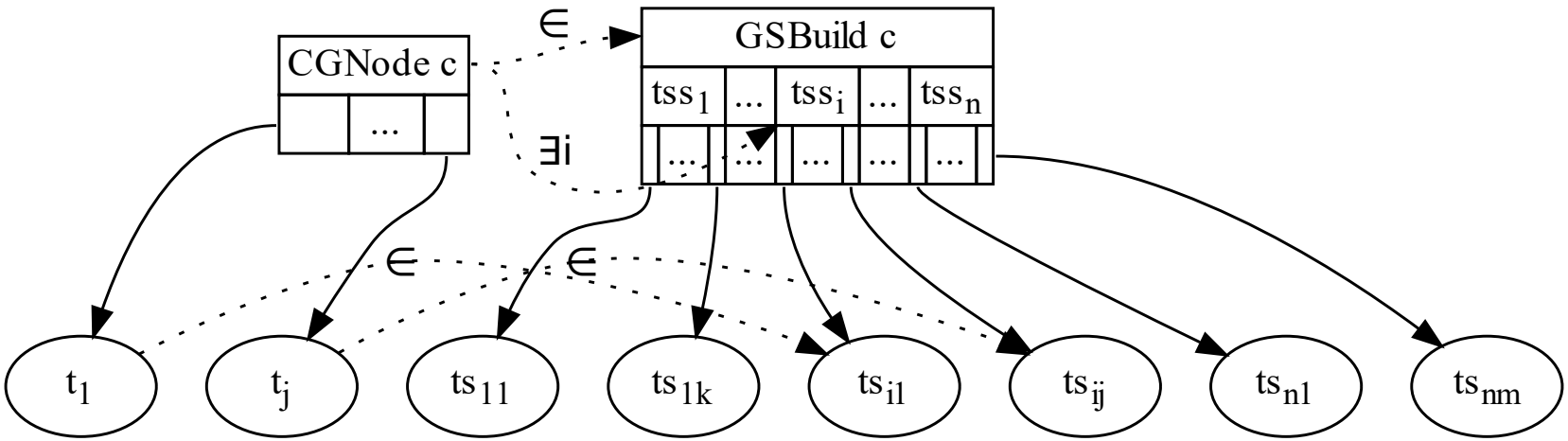
- Supercompilation:
  - transforms (*drives*) *configurations*, which represent sets of possible states of program execution
  - organizes them into *configuration trees* (because transforming a configuration can produce several different new ones, for example due to branching in the input program)
  - performs *folding* to a previously met configuration whenever possible, to turn the potentially infinite configuration tree into a *configuration graph*
- Folding by itself does not guarantee termination of supercompilation. Solution:
  - add dynamic termination checks w.r.t. already explored configurations (*whistle*, based, for example, on the homeomorphic embedding relation)
  - if non-termination risk detected $\Rightarrow$ *generalize*
    - `f(Cons(x, xs), Cons(y, ys))`
      $\Rightarrow$ `let z0 = Cons(x, xs) in f(z0, Cons(y, ys))`

Multi-result Supercompilation (MRSC)

- Classical supercompilation generalizes as late and as little as possible
- In certain situations this can actually lead to worse results
- Key insight of multi-result supercompilation: explore different times and ways to generalize, hoping to find a "better" result (in some sense) among the alternatives
- Example: `f(xs, ys) = fbody` $\in$ program P; transform `f(Cons(x, xs), Cons(y, ys))` to several alternative configurations:
  - `fbody [ xs` $\rightarrow$ `Cons(x, xs), ys` $\rightarrow$ `Cons(y, ys) ]` (*unfolding*, where $e[x \rightarrow e_1, y \rightarrow e_2, \ldots]$ denotes substitution)
  - `let z0 = Cons(x, xs) in f(z0, Cons(y, ys))` (*generalization*)
  - `let z0 = Cons(y, ys) in f(Cons(x, xs), z0)` (a different *generalization*)
  - maybe some other generalizations

Representing Sets of Configuration Graphs

- Conceptually, we should clone the current configuration tree each time we want to explore several alternatives
  $\Rightarrow$ potentially exponential blow-up of the number of configuration trees
- Solution: compact representation, which merges alternative configuration trees/graphs into a single (labeled) graph (with some efficient operations supported: set membership, filtering, . . .)



Using MRSC to Limit Code Size Explosion

- Key idea: for each configuration, explore with priority generalizations, which avoid the risk of code explosion
- In parallel, also explore what a traditional supercompiler would do with the configuration (aggressive unfolding, information propagation, . . .)
- use the efficient filtering operations on the resulting set of configuration graphs, in order to select:
  - the first configuration graph (the one corresponding to maximum generalization at each step)
  - the last one (the one corresponding to what a traditional supercompiler would produce without any generalization, if possible)
  - the smallest and the largest configuration graphs

Input Language

- Tiny first-order functional language with pattern-matching and non-pattern-matching definitions

Expressions $e ::= x$ variable
$\mid a(e_1, \ldots, e_n)$ call

Call kinds $a ::= C$ constructor
$\mid f$ function

Patterns $p ::= C(x_1, \ldots, x_n)$

Function definitions

$d ::= f(x_1, \ldots, x_n) = e$ ordinary function
$\mid g(p_1, y_1, \ldots, y_m) = e_1$ pattern-matching function
$\ldots$
$g(p_n, y_1, \ldots, y_m) = e_n$

Programs $P ::= d_1, \ldots, d_n$

Input Language Examples

- Example: list append, double-append expression

```
append(Nil, ys) = ys;
append(Cons(x, xs), ys) = Cons(x, append(xs, ys));
append(append(xs, ys), zs)
```

- Example: Boolean equality, commutativity

```
not(True)  = False;
not(False) = True;
eqBool(True,  b) = b;
eqBool(False, b) = not(b);
eqBool(eqBool(x, y), eqBool(y, x))
```

Examples (cont.)

- Example: reconstructing Knuth-Morris-Pratt algorithm by supercompilation (program omitted due to size, see paper)

```
...
isSublist(p, s) = match(p, s, p, s);
isSublist(Cons(True, Cons(True, Cons(False,
    Nil))), s)
```

- Example: artificial program demonstrating code explosion in supercompilation (M.H. Sørensen, MSc thesis, 1994)

```
g(Nil,          y) = y;
g(Cons(x, xs), y) = f(g(xs, y));
f(w) = B(w, w);

g(Cons(A, Cons(A, Cons(A, Nil))), z)
```

Driving With Size-limiting Generalization

- Driving and generalization of different expressions shapes (underlining indicates subexpressions to be driven further):
- $x \Rightarrow$
  - x
- $C(e_1, \ldots, e_n) \Rightarrow$
  - $C(\underline{e_1}, \ldots, \underline{e_n})$
- if $f(x_1, \ldots, x_n) = e \in P$ then $f(e_1, \ldots, e_n) \Rightarrow$
  - $\texttt{let}\ y_1 = \underline{e_1}, \ldots, y_n = \underline{e_n}\ \texttt{in}\ \underline{e[x_1 \to y_1, \ldots, x_n \to y_n]}$
    (where $y_1, \ldots, y_n$ – fresh)
  - $\underline{e[x_1 \to e_1, \ldots, x_n \to e_n])]}$

Driving With Size-limiting Generalization (cont.)

- if $g(C(x_1, \ldots, x_m), y_1, \ldots, y_n) = e \in P$ then $g(C(e'_1, \ldots, e'_m), e_1, \ldots, e_n) \Rightarrow$
  - $\texttt{let}\ u_1 = \underline{e'_1}, \ldots, u_m = \underline{e'_m}, z_1 = \underline{e_1}, \ldots, z_n = \underline{e_n}\ \texttt{in}$
    $\underline{e[x_1 \to u_1, \ldots, x_m \to u_m, y_1 \to z_1, \ldots, y_n \to z_n]}$
    (where $u_1, \ldots, u_m, z_1, \ldots, z_n$ – fresh)
  - $\underline{e[x_1 \to e'_1, \ldots, x_m \to e'_m, y_1 \to e_1, \ldots, y_n \to e_n]}$
- if $g(p_1, y_1, \ldots, y_n) = e'_1, \ldots, g(p_m, y_1, \ldots, y_n) = e'_m \in P$ then $g(x, e_1, \ldots, e_n) \Rightarrow$
  - $\texttt{case}\ x\ \texttt{of}\ \{\ p_1 \to \underline{\texttt{propagate}(x, p_1, (e_1, \ldots, e_n), (y_1, \ldots, y_n), e'_1)};$
    $\ldots; p_m \to \underline{\texttt{propagate}(x, p_m, (e_1, \ldots, e_n), (y_1, \ldots, y_n), e'_m)};\ \}$

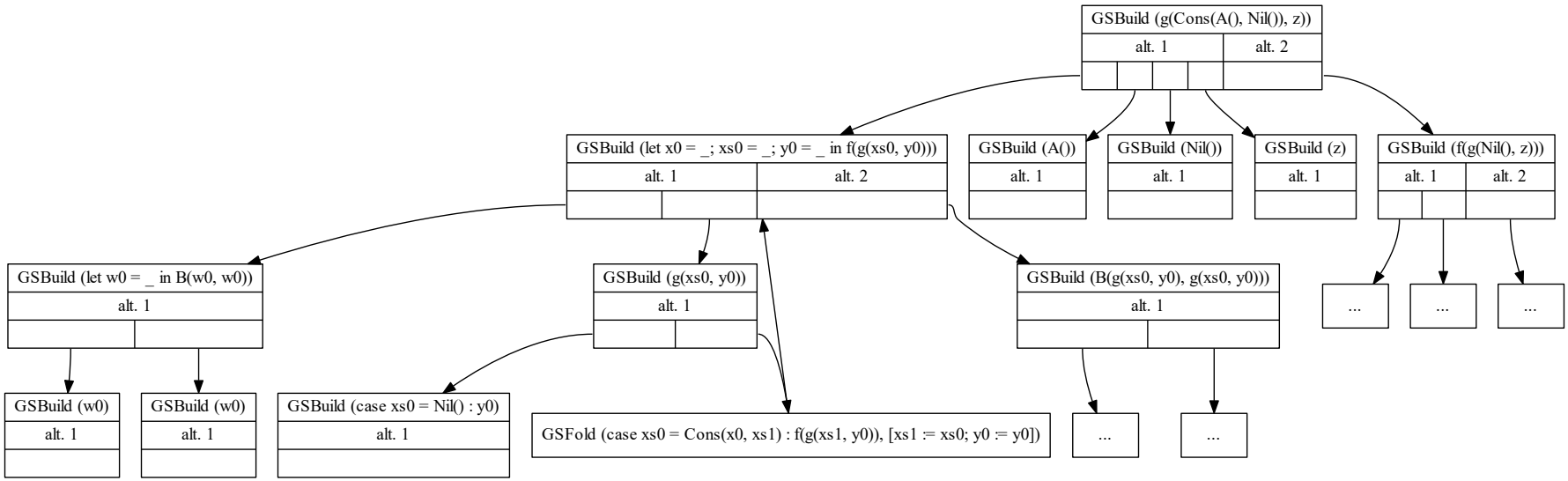
Driving With Size-limiting Generalization (cont.)

- $g(f(e'_1, \ldots, e'_m), e_1, \ldots, e_n) \Rightarrow$
  - $\texttt{let}\ x_0 = f(e'_1, \ldots, e'_m), x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ \underline{g(x_0, \ldots, x_n)}$

    (where $x_0, \ldots, x_n$ – fresh)
  - $\underline{g(\underline{f(e'_1, \ldots, e'_m)}, e_1, \ldots, e_n)}$

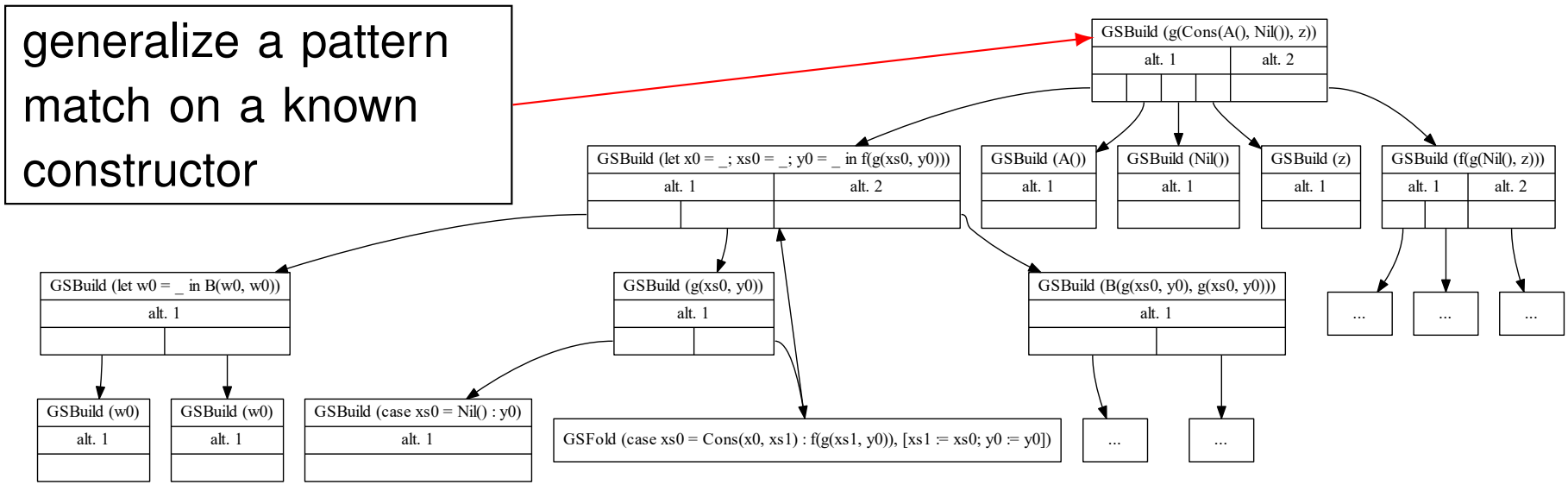
MRSC Example

- Configuration graph of `g(Cons(A, Nil), z)` (using the code-explosion program from earlier slides)
  - only the leftmost branches of the graph are fully shown, the remaining parts are omitted



MRSC Example

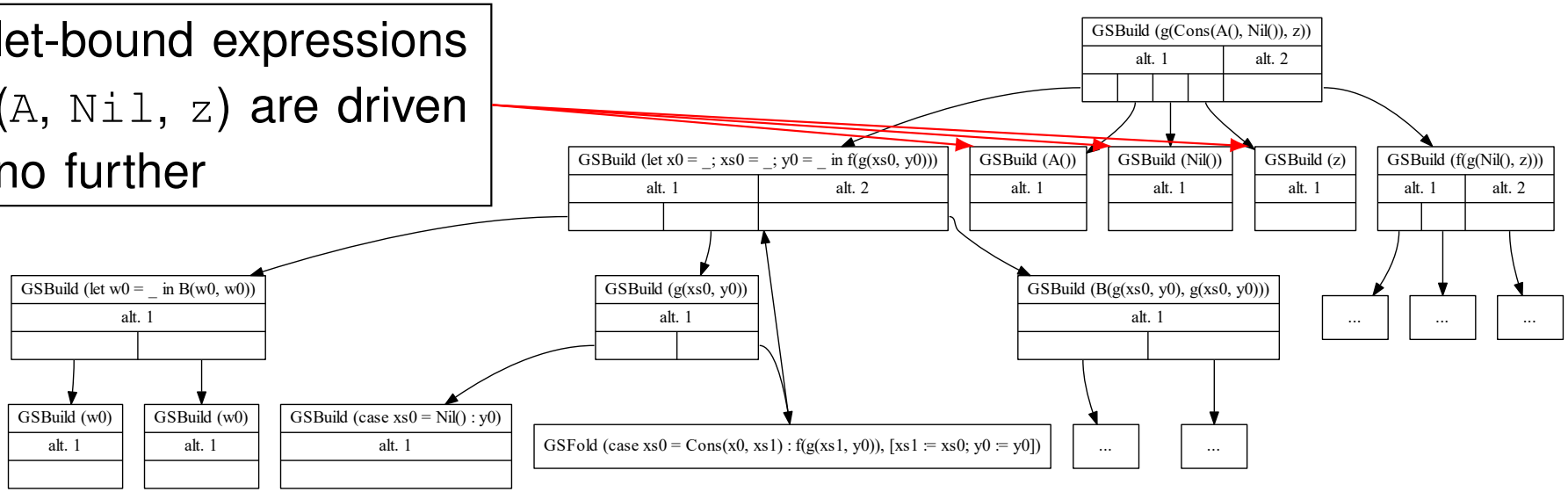
- Configuration graph of `g(Cons(A, Nil), z)` (using the code-explosion program from earlier slides)
  - only the leftmost branches of the graph are fully shown, the remaining parts are omitted



MRSC Example

- Configuration graph of `g(Cons(A, Nil), z)` (using the code-explosion program from earlier slides)
  - only the leftmost branches of the graph are fully shown, the remaining parts are omitted

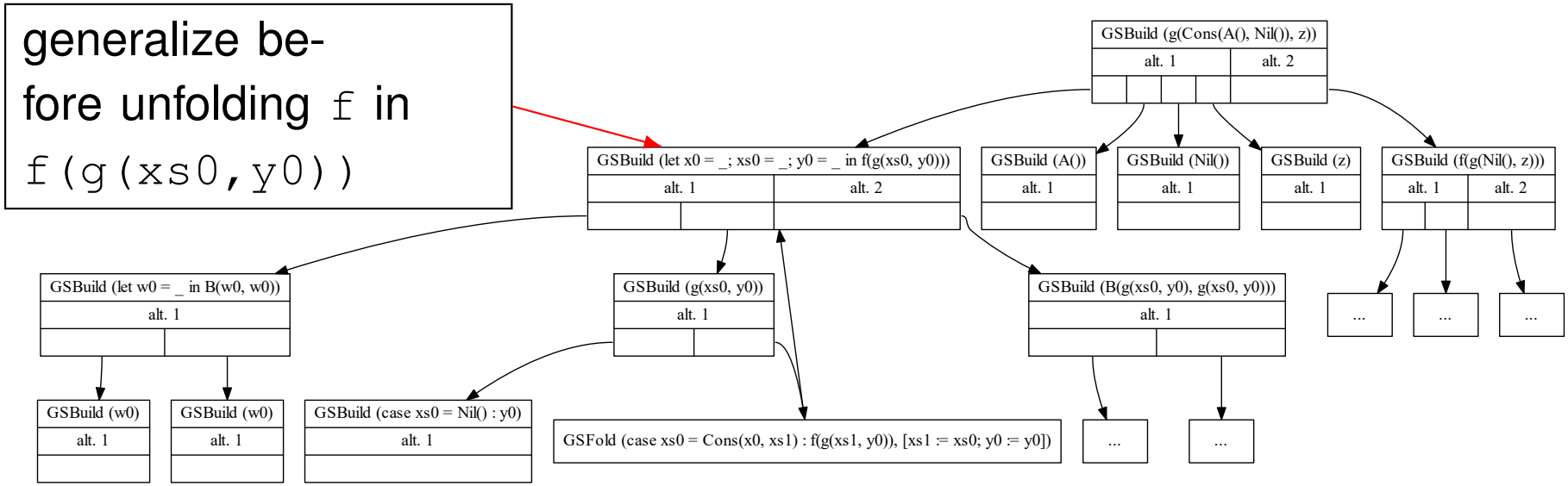
let-bound expressions (`A`, `Nil`, `z`) are driven no further



MRSC Example

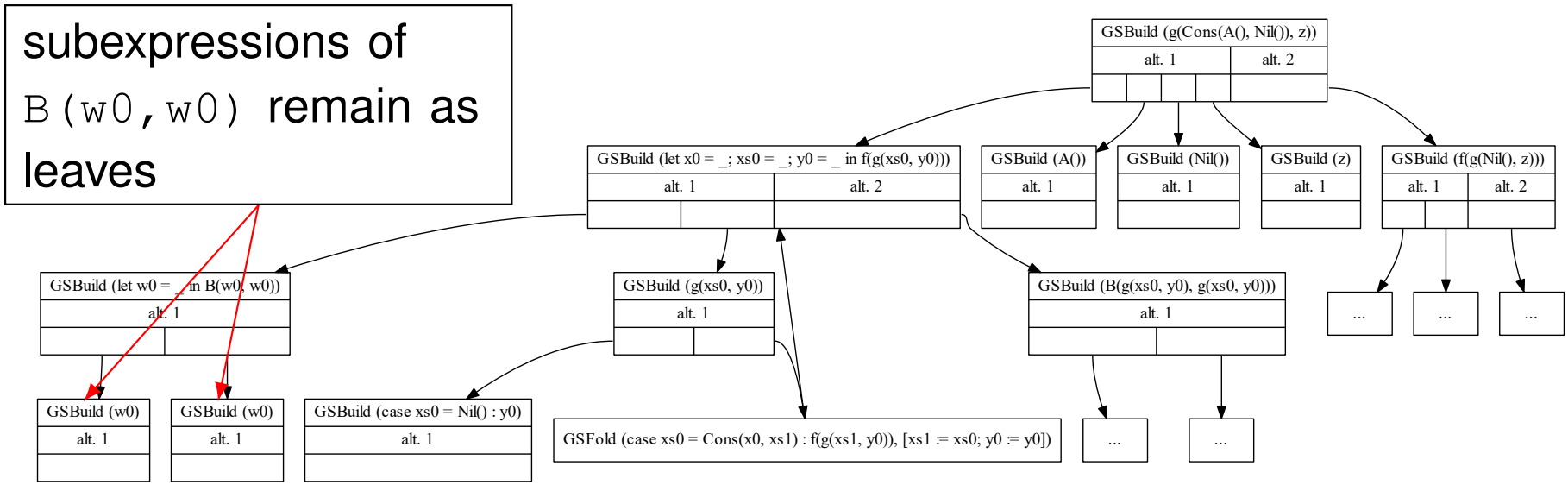
- Configuration graph of `g(Cons(A, Nil), z)` (using the code-explosion program from earlier slides)
  - only the leftmost branches of the graph are fully shown, the remaining parts are omitted

generalize before unfolding `f` in `f(g(xs0,y0))`



MRSC Example

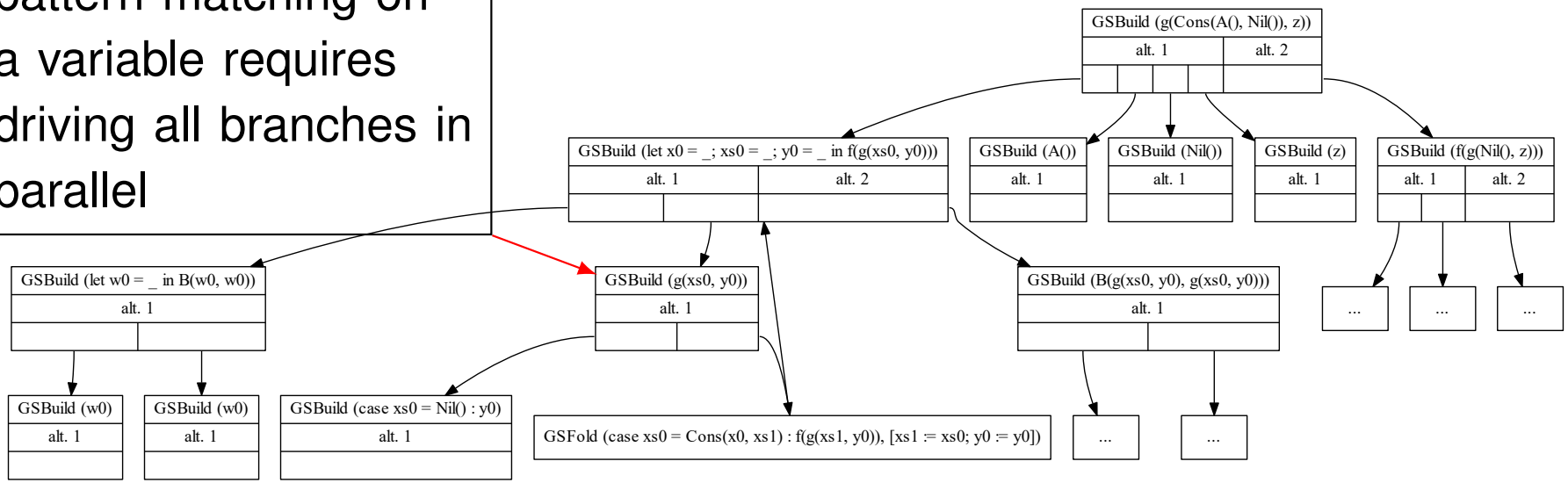
- Configuration graph of `g(Cons(A, Nil), z)` (using the code-explosion program from earlier slides)
  - only the leftmost branches of the graph are fully shown, the remaining parts are omitted



MRSC Example

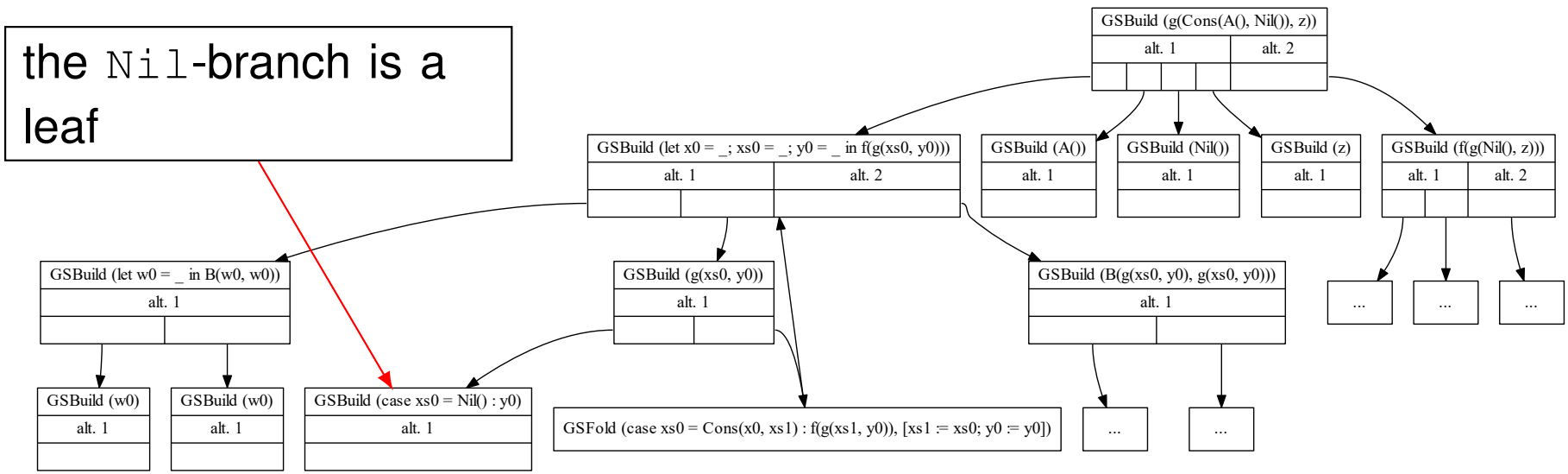
- Configuration graph of `g(Cons(A, Nil), z)` (using the code-explosion program from earlier slides)
  - only the leftmost branches of the graph are fully shown, the remaining parts are omitted

pattern matching on a variable requires driving all branches in parallel



MRSC Example

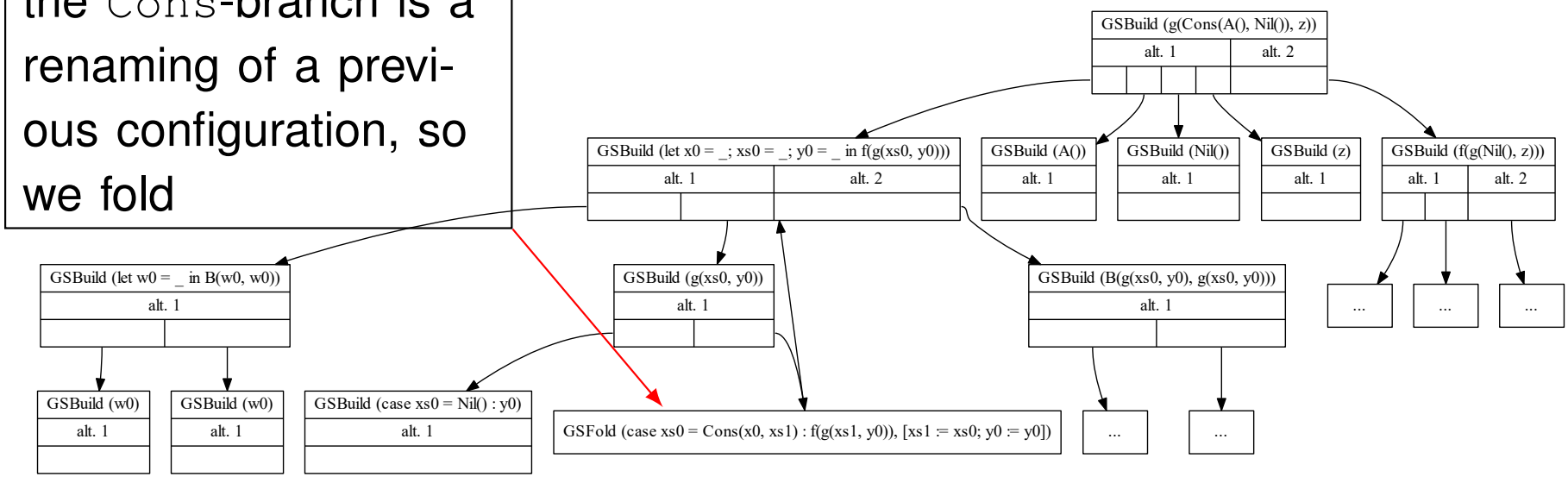
- Configuration graph of `g(Cons(A, Nil), z)` (using the code-explosion program from earlier slides)
  - only the leftmost branches of the graph are fully shown, the remaining parts are omitted



MRSC Example

- Configuration graph of `g(Cons(A, Nil), z)` (using the code-explosion program from earlier slides)
  - only the leftmost branches of the graph are fully shown, the remaining parts are omitted

the `Cons`-branch is a renaming of a previous configuration, so we fold



Empirical Evaluation – Example Statistics

- Statistics about 4 examples from previous slides

Example	First	Last	Min. size	Max. size
double append	12	10	10	19
KMP test	203	39	38	1055
`eqBool` symmetry	16	17	16	30
exp growth	15	37	15	57

- Minimum-size result for “exp growth” example:

```
main_let1(w0) = B(w0, w0);
expression: main_let1(main_let1(B(z, z)))
```

- Key take: exploring different combinations of (sub-)configuration generalizations + a mechanism for quickly finding suitable graphs among the whole set of alternative graphs results in a reliable way to obtain an optimal program without code explosion

Empirical Evaluation – Examples (cont.)

- Statistics on several different examples (from long version in arXiv):

Example	First	Last	Min. size	Max. size
Even-or-odd	14	18	14	21
`idNat` Idempotent	9	6	6	12
take-length	13	8	8	19
length-intersperse	36	27	27	187

- Minimum-size result for `take(length(xs), xs)`

```
f_(xs) = f__case0(xs);
f__case0(Nil(), ) = Nil();
f__case0(Cons(x00, xs00), ) = Cons(x00, f_(xs00));
expression: f_(xs)
```

Conclusions

Achievements:

- An approach for systematically exploring different combinations of configuration generalizations, which
  - keeps the benefits of the aggressive optimizations performed by traditional supercompilers
  - while reducing the risk of code size explosion in the resulting program
- The approach re-uses existing ideas about efficient implementation of MRSC, coupled with a generalization strategy specifically aimed at reducing risks of code size explosion
- Empirical evaluation based on several small examples (typically used for benchmarks of supercompilers and similar program transformers) shows encouraging results

Future Work

- Evaluate the approach on more and larger examples
- Refine approach to generalization
  - for example, no need to generalize a variable
  - not useful to generalize a function argument, which occurs only once in the body
- Study theoretical properties, especially potential bounds on result size

Future Work (cont.)

Idea for bounds on result program size (WIP after paper published):

- Use a further restricted input language (still Turing-complete though)
  - only pattern-matching function definitions
  - function calls in "A-Normal Form": $g(x_1, \ldots, x_n)$
  - only direct recursion $\Rightarrow$ we can topologically sort definitions:
    - $P = g_1(\ldots) = e_1, \ldots, g_n(\ldots) = e_n$
    - s.t. if $g_j(x_1, \ldots, x_m) \in e_i$ then $j \geq i$
- Current conjecture on size bound:
  - $\forall i \exists C.graphSize(g_i) <= C + \sum_{g_j(x_1,\ldots,x_m) \in e_i \wedge j > i} graphSize(g_j)$