Using Constraint Programming to Reason About Procedural Code

Kathryn Francis and Peter J. Stuckey

University of Melbourne
National ICT Australia

18th May, 2015

Outline

1. Motivation
   - our purpose for reasoning about procedural code
2. Background
   - brief introduction to constraint programming (CP)
3. Standard approaches
   - explanation of how CP + SMT are typically applied to this problem
4. Query-Based approach
   - a better CP encoding
5. (Bounded) Loops
   - Unwinding
   - Untangling
6. Global Constraint for Reaching Definitions
   - our current work
7. Conclusion

Outline

1. Motivation
   - our purpose for reasoning about procedural code
2. Background
   - brief introduction to constraint programming (CP)
3. Standard approaches
   - explanation of how CP + SMT are typically applied to this problem
4. Query-Based approach
   - a better CP encoding
5. (Bounded) Loops
   - Unwinding
   - Untangling
6. Global Constraint for Reaching Definitions
   - our current work
7. Conclusion

Motivation

We have a goal to make it easier for programmers to include optimisation functionality in their applications.

Using existing tools is difficult and time-consuming.
The programmer has to..

- learn how constraint solvers work
- discover available variable and constraint types
- figure out how to represent high-level decisions (and their consequences) using these
- create a model using an unfamiliar language or library methods
- convert returned values back into a usable representation of the solution

Motivation

Our goal is to allow optimisation problems to be defined natively within a general purpose programming language.

The programmer writes code to

- `build` a candidate solution from individual decisions
- `evaluate` a candidate solution

The optimisation library provides a function `buildOptimal` which takes the above two functions as arguments.

Calling `buildOptimal` is equivalent calling `build` using decisions which will give the best return value for `evaluate`.

Travelling Salesman

The programmer writes code (in the application programming language) to:

- Build a list data structure containing a (random) permutation of the stops.
- Compute the travel time for a given ordered list of stops.

The `buildOptimal` method can be used to obtain a list containing the stops in optimal order (with minimal travel time).

Bin Packing

The programmer writes code to

- Build a set data structure containing a (random) subset of the items.
- Compute the total value of a set of items, returning a special ‘invalid’ value if the total weight is too great.

The `buildOptimal` method can be used to obtain a set of items with maximum value for the acceptable weight.

Motivation - Implementing the library

Requirements

Given the two functions written by the programmer, find inputs for `build` which produce a candidate solution with maximum or minimum return value for a subsequent call to `evaluate`.

We achieve this by

1. Translating the code into a constraint model defining the relationship between the inputs and the evaluation result.
2. Using a constraint solver to find an optimal set of inputs.

This same translation could be used to discover other things about the code.

- What input values allow execution to reach this statement?

What we are not going to talk about

Unbounded loops

- We consider the translation of code where loops are bounded
- Good enough for
  - simulation optimization (usually)
  - bounded model checking
  - congolic testing
- Unbounded loops require: abstract interpretation, interpolation, or perhaps lazy constraint addition!

Outline

1. Motivation
   - our purpose for reasoning about procedural code
2. Background
   - brief introduction to constraint programming (CP)
3. Standard approaches
   - explanation of how CP + SMT are typically applied to this problem
4. Query-Based approach
   - a better CP encoding
5. (Bounded) Loops
   - Unwinding
   - Untangling
6. Global Constraint for Reaching Definitions
   - our current work
7. Conclusion

Terminology

- A **model** is a collection of **variables** and **constraints**.
- A **variable** has a **domain** of possible values.
- A **constraint** imposes rules about (dis)allowed combinations of values for variables.
- A **solution** is an assignment of values to variables satisfying all constraints.
- An **optimal solution** is a solution with minimum or maximum value for the **objective variable**.
- Constraints are enforced in the solver using **propagators**. Their purpose is to make deductions about the allowed values for each variable given the current domains of other related variables. This is called **propagation**.

Basic Solver Algorithm

```
void solve() {
  if (some variable has an empty domain) {
    return; // no solutions down here
  }
  if (all variables are fixed) {
    report a solution;
    add constraint obj > current obj;
    return; // the added constraint has to cause an empty domain
  }
  while exists variable v which is not yet fixed {
    for each value x in domain(v) {
      set v = x;
      run propagators; // this will reduce other domains
      solve(); // recursive call
      undo what the propagators did;
} } }
```

stronger propagation $\rightarrow$ smaller domains $\rightarrow$ less search

element constraint

$y = [a, b, c, d][x]$

- remove any index where the element cannot equal y
- remove a value for y if no element at a possible index can take that value
- when x is fixed, the chosen element must equal y

sum constraint (linear)

s = a + b + c + d

- rearrange for each variable: $a = s - (b + c + d)$
- calculate possible values for rhs and use this to filter lhs

global cardinality constraint

Card(*vars*, *values*, *counts*)

- the number of times *values*[*i*] occurs in the array *vars* is given by *counts*[*i*].
- propagation based on matching algorithms
- advantage of CP defining complex constraints

Outline

1. Motivation
   - our purpose for reasoning about procedural code
2. Background
   - brief introduction to constraint programming (CP)
3. Standard approaches
   - explanation of how CP + SMT are typically applied to this problem
4. Query-Based approach
   - a better CP encoding
5. (Bounded) Loops
   - Unwinding
   - Untangling
6. Global Constraint for Reaching Definitions
   - our current work
7. Conclusion

Standard CP Approach

To reason about procedural code:

1. Build a model representing the relationship between inputs and outputs.
2. Search for a solution to this model satisfying some further constraints (in our case, maximizing return value for `evaluate`).

- Note that we do not necessarily search over execution paths directly.
- We may instead search over input values.
- When these are all known so is the execution path.

Standard CP Approach

Building the model:

- A path constraint tells whether each statement will be executed.
- Assignments result in a new version of the assigned variable.
- For field assignments a new version is created for every possibly affected object.
- The new value is constrained to be the assigned one if the path constraint is satisfied, and the previous value otherwise.

State is represented at each point

Standard CP Approach - Example

```
int makegroups(Group mygroup, Group yourgroup,
      bool ibringfamily, bool youbringfamily) {
   mygroup.size++;
   yourgroup.size++;
   if(ibringfamily)
      mygroup.size += 4;
   if(youbringfamily)
      yourgroup.size += 2;
   return mygroup.size;
}
```

Assume:

- mygroup in $\{A, B\}$
- yourgroup = A
- both A and B have initial size 0

Standard CP Approach - Example

```
mygroup.size++;
yourgroup.size++;
if(ibringfamily)
   mygroup.size += 4;
if(youbringfamily)
   yourgroup.size += 2;
```

Assume:

- mygroup in $\{A, B\}$
- yourgroup = A
- both A and B have initial size 0

$s_{A0} = 0$
$s_{B0} = 0$
$s_{A1} = [s_{A0}, s_{A0}+1][\text{mygr}=A]$
$s_{B1} = [s_{B0}, s_{B0}+1][\text{mygr}=B]$
$s_{A2} = s_{A1}+1$
$s_{A3} = [s_{A2}, s_{A2}+4][\text{ibring} \wedge \text{mygr}=A]$
$s_{B2} = [s_{B1}, s_{B1}+4][\text{ibring} \wedge \text{mygr}=B]$
$s_{A4} = [s_{A3}, s_{A3}+2][\text{youbring}]$
$\text{returnval} = [s_{B2}, s_{A4}][\text{mygr}=A]$

We use element for if-then-else to allow disjunctive reasoning.

Standard CP Approach - Problems

Aliasing causes weak propagation:

$s_{A0} = 0$	$\{0\}$
$s_{B0} = 0$	$\{0\}$
$s_{A1} = [s_{A0}, s_{A0}+1][\text{mygr}=A]$	$\{0, 1\}$
$s_{B1} = [s_{B0}, s_{B0}+1][\text{mygr}=B]$	$\{0, 1\}$
$s_{A2} = s_{A1}+1$	$\{1, 2\}$
$s_{A3} = [s_{A2}, s_{A2}+4][\text{ibring} \wedge \text{mygr}=A]$	$\{1, 2, 5, 6\}$
$s_{B2} = [s_{B1}, s_{B1}+4][\text{ibring} \wedge \text{mygr}=B]$	$\{0, 1, 4, 5\}$
$s_{A4} = [s_{A3}, s_{A3}+2][\text{youbring}]$	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
$\text{returnval} = [s_{B2}, s_{A4}][\text{mygr}=\text{A}]$	$\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

We think returnval may be 0 (my group might be empty).

Standard SMT Approach

Building the model:

- Use if-then-else (ITE) constructs to control which paths are executed.
- Represent field assignments using an array to carry the field value for all objects.
- Access field values using "read" (R) on the array
- Update field values using "write" (W) on the array
- Theory of arrays:

$$\begin{aligned} \forall a, i, j, x. i = j &\rightarrow \mathsf{R}(\mathsf{W}(a, i, x), j) = x \\ i \neq j &\rightarrow \mathsf{R}(\mathsf{W}(a, i, x), j) = \mathsf{R}(a, j) \end{aligned}$$

State is represented concisely at each point

Standard SMT Approach - Example

```
mygroup.size++;
yourgroup.size++;
if(ibringfamily)
   mygroup.size += 4;
if(youbringfamily)
   yourgroup.size += 2;
```

Assume:

- mygroup in $\{A, B\}$
- yourgroup = A
- both A and B have initial size 0

$$size_0 = \mathrm{W}(\mathrm{W}(\bot,\mathrm{A},0),\mathrm{B},0)$$
$$size_1 = \mathrm{W}(size_0, \mathrm{mygr}, \mathrm{R}(size_0, \mathrm{mygr})+1)$$
$$size_2 = \mathrm{W}(size_1, \mathrm{A}, \mathrm{R}(size_1, \mathrm{A})+1)$$
$$size_3 = \mathrm{ITE}(\mathrm{ibring}, \mathrm{W}(size_2, \mathrm{mygr}, \mathrm{R}(size_2, \mathrm{mygr})+4), size_2)$$
$$size_4 = \mathrm{ITE}(\mathrm{youbring}, \mathrm{W}(size_3, \mathrm{A}, \mathrm{R}(size_3, \mathrm{A}) + 2), size_3)$$
$$\mathrm{returnval} = \mathrm{R}(size_4, \mathrm{mygr})$$

Concise but
no disjunctive reasoning using
theory of arrays

Outline

1. Motivation
   - our purpose for reasoning about procedural code
2. Background
   - brief introduction to constraint programming (CP)
3. Standard approaches
   - explanation of how CP + SMT are typically applied to this problem
4. Query-Based approach
   - a better CP encoding
5. (Bounded) Loops
   - Unwinding
   - Untangling
6. Global Constraint for Reaching Definitions
   - our current work
7. Conclusion

Query-Based Approach

NICTA THE UNIVERSITY OF MELBOURNE

We realised:

- We don't need to know the state of every object at every program point.
- For correctness, we just need the variable references (queries) to correspond correctly to the assignments.

So we developed a new approach:

- Don't create variables for the state at each program point.
- Instead directly model the relationship between assignments and variable references.
- This supports aliasing without introducing an excessive number of variables.
- It also makes stronger propagation possible.

Query-Based Approach

First convert the code into an ordered list of assignments.

```
mygroup.size++;
yourgroup.size++;
```

```
if(ibringfamily)
   mygroup.size += 4;
if(youbringfamily)
   yourgroup.size += 2;
```

condition	object		field		value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	mygroup.size+ 1
	yourgroup	.	size	=	yourgroup.size+ 1
(ibringfamily)	mygroup	.	size	=	mygroup.size + 4
(youbringfamily)	yourgroup	.	size	=	yourgroup.size + 2

Query-Based Approach

Then identify queries (variable references) of interest.

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$mysize_1$	=	mygroup.size	(after assignment 2)
$yoursize_1$	=	yourgroup.size	(after assignment 3)
$mysize_2$	=	mygroup.size	(after assignment 4)
$yoursize_2$	=	yourgroup.size	(after assignment 5)
returnval	=	mygroup.size	(after assignment 6)

Query-Based Approach

For each query of interest:

- Define a solver variable representing which assignment provides the value.
- This is the assignment which both reaches and matches the query.

Reaches: Is executed before the query and not over-written by an intervening assignment.

Matches: Uses the queried object.

Query-Based Approach

field reference: queryobj.field

assignments:
$$\begin{aligned} &\text{cond1} : \text{obj1.field} := \text{expr1} \\ &\qquad\qquad\qquad\qquad\quad \ldots \\ &\text{condn} : \text{objn.field} := \text{exprn} \end{aligned}$$

$$\Downarrow$$

constraints:
$$\begin{aligned} \text{queryresult} &= [\text{expr1}, \ldots, \text{exprn}][\text{indexvar}] \\ \text{queryobj} &= [\text{obj1}, \ldots, \text{objn}][\text{indexvar}] \\ \text{true} &= [\text{cond1}, \ldots, \text{condn}][\text{indexvar}] \end{aligned}$$

$$\begin{aligned} &(\text{cond2} \wedge \text{queryobj} = \text{obj2}) \rightarrow \text{indexvar} \geq 2 \\ &\qquad\qquad\qquad\qquad\qquad\quad \ldots \\ &(\text{condn} \wedge \text{queryobj} = \text{objn}) \rightarrow \text{indexvar} \geq n \end{aligned}$$

Query-Based Approach - Example

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$mysize_1$	=	mygroup.size	(after assignment 2)
$yoursize_1$	=	yourgroup.size	(after assignment 3)
$mysize_2$	=	mygroup.size	(after assignment 4)
$yoursize_2$	=	yourgroup.size	(after assignment 5)
returnval	=	mygroup.size	(after assignment 6)

Query-Based Approach - Example

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$mysize_1$	**=**	**mygroup.size**	**(after assignment 2)**
$yoursize_1$	=	yourgroup.size	(after assignment 3)
$mysize_2$	=	mygroup.size	(after assignment 4)
$yoursize_2$	=	yourgroup.size	(after assignment 5)
returnval	=	mygroup.size	(after assignment 6)

$mysize_1 = 0$

Query-Based Approach - Example

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$mysize_1$	=	0	
$yoursize_1$	**=**	**yourgroup.size**	**(after assignment 3)**
$mysize_2$	=	mygroup.size	(after assignment 4)
$yoursize_2$	=	yourgroup.size	(after assignment 5)
returnval	=	mygroup.size	(after assignment 6)

$var\ 1..3 : index_1$

$yoursize_1 = [0, 0, mysize_1 + 1][index_1]$

$yourgroup = [A, B, mygroup][index_1]$

$true = [true, true, true][index_1]$

Query-Based Approach - Example

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$mysize_1$	=	0	
$yoursize_1$	**=**	**yourgroup.size**	**(after assignment 3)**
$mysize_2$	=	mygroup.size	(after assignment 4)
$yoursize_2$	=	yourgroup.size	(after assignment 5)
returnval	=	mygroup.size	(after assignment 6)

$var\ 1..2 : index_1$

$yoursize_1 = [0, 1][index_1]$

$yourgroup = [A, mygroup][index_1]$

$(yourgroup = mygroup) \rightarrow index_1 >= 2$

Query-Based Approach - Example

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$mysize_1$	=	0	
$yoursize_1$	**=**	**yourgroup.size**	**(after assignment 3)**
$mysize_2$	=	mygroup.size	(after assignment 4)
$yoursize_2$	=	yourgroup.size	(after assignment 5)
returnval	=	mygroup.size	(after assignment 6)

$$yoursize_1 = [0, 1][yourgroup = mygroup]$$

Query-Based Approach - Example

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$mysize_1$	=	0	
$yoursize_1$	=	$[0, 1][yourgroup = mygroup]$	
$mysize_2$	**=**	**mygroup.size**	**(after assignment 4)**
$yoursize_2$	=	yourgroup.size	(after assignment 5)
$returnval$	=	mygroup.size	(after assignment 6)

$mysize_2 = [0, 0, mysize_1 + 1, yoursize_1 + 1][index_1]$
$mygroup = [A, B, mygroup, yourgroup][index_1]$
$(mygroup = B) \rightarrow index_1 >= 2$
$(mygroup = mygroup) \rightarrow index_1 >= 3$
$(mygroup = yourgroup) \rightarrow index_1 >= 4$

Query-Based Approach - Example

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$mysize_1$	=	0	
$yoursize_1$	=	$[0, 1][yourgroup = mygroup]$	
$mysize_2$	**=**	**mygroup.size**	**(after assignment 4)**
$yoursize_2$	=	yourgroup.size	(after assignment 5)
$returnval$	=	mygroup.size	(after assignment 6)

$$mysize_2 = [mysize_1 + 1, yoursize_1 + 1][mygroup = yourgroup]$$

Query-Based Approach - Example

Condition	Object		Field		Assigned value
	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$$
\begin{array}{rcll}
mysize_1 & = & 0 & \\
yoursize_1 & = & [0, 1][yourgroup = mygroup] & \\
mysize_2 & = & [1, yoursize_1 + 1][mygroup = yourgroup] & \\
\mathbf{yoursize_2} & \mathbf{=} & \mathbf{yourgroup.size} & \textbf{(after assignment 5)} \\
returnval & = & \text{mygroup.size} & \text{(after assignment 6)}
\end{array}
$$

$$yoursize_2 = [yoursize_1 + 1, mysize_2 + 4][ygroup = mgroup \wedge ibring]$$

Query-Based Approach - Example

	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

$$
\begin{aligned}
mysize_1 &= 0 \\
yoursize_1 &= [0, 1][yourgroup = mygroup] \\
mysize_2 &= [1, yoursize_1 + 1][mygroup = yourgroup] \\
yoursize_2 &= [yoursize_1 + 1, mysize_2 + 4][ygroup = mgroup \wedge ibring] \\
\textbf{returnval} &= \textbf{mygroup.size} \quad \textbf{(after assignment 6)}
\end{aligned}
$$

$$
\begin{aligned}
&returnval = [mysize_1 + 1, yoursize_1 + 1, mysize_2 + 4, yoursize_2 + 2][index_1] \\
&mygroup = [mygroup, yourgroup, mygroup, yourgroup][index_1] \\
&true = [true, true, ibring, youbring][index_1] \\
&(yourgroup = mygroup) \rightarrow index_1 \geq 2 \\
&(mygroup = mygroup \wedge ibringfamily) \rightarrow index_1 \geq 3 \\
&(yourgroup = mygroup \wedge youbringfamily) \rightarrow index_1 \geq 4
\end{aligned}
$$

Query-Based Approach - Example

This is our new model:

$$
\begin{array}{rcll}
mysize_1 & = & 0 & \{0\} \\
yoursize_1 & = & [0,1][yourgroup = mygroup] & \{0,1\} \\
mysize_2 & = & [1, yoursize_1{+}1][mygroup = yourgroup] & \{1,2\} \\
yoursize_2 & = & [yoursize_1{+}1, mysize_2{+}4][ygroup = mgroup \wedge ibring] & \{1,2,5,6\} \\
returnval & = & [1, yoursize_1{+}1, mysize_2{+}4, yoursize_2{+}2][index_1] & \{1..8\}
\end{array}
$$

$$
\begin{array}{rcl}
mygroup & = & [mygroup, yourgroup, mygroup, yourgroup][index_1] \\
true & = & [true, true, ibring, youbring][index_1]
\end{array}
$$

$$(yourgroup = mygroup) \rightarrow index_1 \geq 2$$

$$(mygroup = mygroup \wedge ibringfamily) \rightarrow index_1 \geq 3$$

$$(yourgroup = mygroup \wedge youbringfamily) \rightarrow index_1 \geq 4$$

Now we know returnval is not 0 (my group is not empty).

Query-Based Approach - Problem

This is better, but we are still missing propagation.

- Imagine we know $returnval < 5$.
- We should deduce that *ibringfamily* is false.
- With these constraints we cannot do that even if we also know $mygroup = yourgroup$.

$$
\begin{array}{rcll}
mysize_1 & = & 0 & \{0\} \\
yoursize_1 & = & [0,1][yourgroup = mygroup] & \{0,1\} \\
mysize_2 & = & [1, yoursize_1{+}1][mygroup = yourgroup] & \{1,2\} \\
yoursize_2 & = & [yoursize_1{+}1, mysize_2{+}4][ygroup = mgroup \wedge ibring] & \{1,2,5,6\} \\
returnval & = & [1, yoursize_1{+}1, \mathbf{mysize_2{+}4}, yoursize_2{+}2][index_1] & \{1..\mathbf{4}\}
\end{array}
$$

$$
\begin{array}{rcl}
mygroup & = & [mygroup, yourgroup, mygroup, yourgroup][index_1] \\
true & = & [true, true, ibring, youbring][index_1]
\end{array}
$$

$$(yourgroup = mygroup) \rightarrow index_1 \geq 2$$

$$(mygroup = mygroup \wedge ibringfamily) \rightarrow index_1 \geq 3$$

$$(yourgroup = mygroup \wedge youbringfamily) \rightarrow index_1 \geq 4$$

Query-Based Approach - Special Cases

To combat this we looked for easily detected special cases where a better translation could be applied.

The *sum* special case

	A	.	size	=	0
	B	.	size	=	0
	mygroup	.	size	=	$mysize_1 + 1$
	yourgroup	.	size	=	$yoursize_1 + 1$
(ibringfamily)	mygroup	.	size	=	$mysize_2 + 4$
(youbringfamily)	yourgroup	.	size	=	$yoursize_2 + 2$

returnval = mygroup.size (after assignment 6)

$$\begin{aligned}\textbf{returnval} = {} & 0 \times (mygroup = A) + 0 \times (mygroup = B) + \\ & 1 \times (mygroup = mygroup) + 1 \times (yourgroup = mygroup) + \\ & 4 \times (mygroup = mygroup \wedge ibringfamily) + \\ & 2 \times (mygroup = yourgroup \wedge youbringfamily)\end{aligned}$$

Query-Based Approach - Special Cases

To combat this we looked for easily detected special cases where a better translation could be applied.

The *sum* special case

$$
\begin{array}{rrlll}
 & A & . \text{ size} & = & 0 \\
 & B & . \text{ size} & = & 0 \\
 & \text{mygroup} & . \text{ size} & = & \mathit{mysize}_1 + 1 \\
 & \text{yourgroup} & . \text{ size} & = & \mathit{yoursize}_1 + 1 \\
\text{(ibringfamily)} & \text{mygroup} & . \text{ size} & = & \mathit{mysize}_2 + 4 \\
\text{(youbringfamily)} & \text{yourgroup} & . \text{ size} & = & \mathit{yoursize}_2 + 2
\end{array}
$$

returnval = mygroup.size (after assignment 6)

$$\textbf{returnval} = 1 + 1 \times (\mathit{yourgroup} = \mathit{mygroup}) + 4 \times (\mathit{ibringfamily}) + 2 \times (\mathit{mygroup} = \mathit{yourgroup} \wedge \mathit{youbringfamily})$$

Query-Based Approach - Special Cases

This does improve propagation:

- again imagine we know $returnval < 5$
- we can now deduce that *ibringfamily* must be false

$$
\begin{aligned}
\textbf{returnval} &= 1 + 1 \times (yourgroup = mygroup) + 4 \times (ibringfamily) + \\
&\quad 2 \times (mygroup = yourgroup \wedge youbringfamily) \\
4 \times ibringfamily &= returnval - (1 + 1 \times (yourgroup = mygroup) + \\
&\quad 2 \times (mygroup = yourgroup \wedge youbringfamily)) \\
4 \times ibringfamily &\leq 4 - (1 + 1 \times (0) + 2 \times (0)) \\
4 \times ibringfamily &\leq 3 \\
ibringfamily &= 0
\end{aligned}
$$

However this only helps for the specific cases we detect.

Query-Based Approach

Improved propagation at what cost:

- State is size n
- Program trace length is m

Standard CP Encoding is $O(mn)$

SMT Encoding is $O(m)$

Query CP Encoding is $O(m^2)$

We may need to make explicit all state at a program point.

There can be no relevant assignments earlier than that point.

Outline

1. Motivation
   - our purpose for reasoning about procedural code
2. Background
   - brief introduction to constraint programming (CP)
3. Standard approaches
   - explanation of how CP + SMT are typically applied to this problem
4. Query-Based approach
   - a better CP encoding
5. (Bounded) Loops
   - Unwinding
   - Untangling
6. Global Constraint for Reaching Definitions
   - our current work
7. Conclusion

Loop Unwinding

Usually, bounded loops are ‘unwound’.

This is performed as a pre-processing step before the translation to constraints.

```
for(slice : slices) {
    PizzaType pt = slice.pizzatype;
    pt.numslices++;
}
return veg.numSlices;
```

```
pt1 = slice1.pizzatype;
pt1.numslices++;

pt2 = slice2.pizzatype;
pt2.numslices++;

pt3 = slice3.pizzatype;
pt3.numslices++;

return veg.numslices;
```

A list of ordered copies of each loop body

Loop Unwinding Translation Example

The final query is constrained like this.

```
pt1 = slice1.pizzatype;
pt1.numslices++;

pt2 = slice2.pizzatype;
pt2.numslices++;

pt3 = slice3.pizzatype;
pt3.numslices++;

return veg.numslices;
```

var 1..4: i;

// assignment i provides value
retval = [0,q1+1,q2+1,q3+1][i];

// assignment i matches
[veg,pt1,pt2,pt3][i] = veg;

// assignment i reaches
[**true**,**true**,**true**,**true**][i] = **true**;
pt1 = veg $\rightarrow$ i $\geq$ 2
pt2 = veg $\rightarrow$ i $\geq$ 3
pt3 = veg $\rightarrow$ i $\geq$ 4

- Queries q1, q2 and q3 all need their own constraints.

Loop Unwinding Translation Example Revised

Representing reaches differently

```
pt1 = slice1.pizzatype;
pt1.numslices++;

pt2 = slice2.pizzatype;
pt2.numslices++;

pt3 = slice3.pizzatype;
pt3.numslices++;

return veg.numslices;
```

```
var 1..4: i;

// assignment i provides value
retval = [0,q1+1,q2+1,q3+1][i];

// assignment i matches
[veg,pt1,pt2,pt3][i] = veg;

// assignment i reaches
[r0,r1,r2,r3][i] = true;
r0 = (pt1≠veg ∧ pt2≠veg ∧ pt3≠veg)
r1 = (pt2≠pt1 ∧ pt3≠pt1)
r2 = (pt3≠pt2)
r3 = true
```

- Make untangling easier to explain

Loop Unwinding

The translation we just saw has some problems.

- Each iteration contains a lot of uncertainty.
- We have to look up numslices for an unknown object.
- We have to update numslices for an unknown object.
- For the final result all that matters is the number of vegetarian slices (we don't care about q1, q2 and q3).

By untangling the loop instead of unwinding it we can address these problems.

Loop Untangling

Untangling: trade uncertainty in objects for uncertainty in order.

Create and label copies of the loop body based on the possible values of a key query, called the label query.

Create a copy of the loop body for each value which may be taken by the label query. If the same value may occur more than once, create a copy for the first, second, third (etc.) time this value is taken.

- There are more copies of the loop body.
- We don't know which are executed in what order.
- But the constraints for each copy can be much simpler.

The Example Untangled

First we need to choose a label query.

```
for(slice : slices) {
    PizzaType pt = slice.pizzatype;    ← label query
    pt.numslices++;
}
return veg.numSlices;
```

Then we find its possible values in each iteration.

Assuming:

slice1.pizzatype $\in$ {**Veg**, **Marg**}
slice2.pizzatype $\in$ {**Veg**, **Marg**}
slice3.pizzatype $\in$ {**Veg**, **Cap**}

Our label query can be:

Veg	0-3 times
Marg	0-2 times
Cap	0-1 times

The Example Untangled

Now create a copy of the body for each potential value.

1st Veg:	pt = veg; pt.numslices++;	1st Marg:	pt = marg; pt.numslices++;
2nd Veg:	pt = veg; pt.numslices++;	2nd Marg:	pt = marg; pt.numslices++;
3rd Veg:	pt = veg; pt.numslices++;	1st Cap:	pt = cap; pt.numslices++;

return veg.numslices;

- We don't know which iterations happen in which order.
- For those with the same label value we enforce an order.

The Example Untangled

Now create a copy of the body for each potential value.

1st Veg:	veg.numslices++;	1st Marg:	marg.numslices++;
2nd Veg:	veg.numslices++;	2nd Marg:	marg.numslices++;
3rd Veg:	veg.numslices++;	1st Cap:	cap.numslices++;

return veg.numslices;

- We don't know which iterations happen in which order.
- For those with the same label value we enforce an order.

The Example Untangled

Translate using the query-based technique

1st Veg:	veg.numslices++;	1st Marg:	marg.numslices++;
2nd Veg:	veg.numslices++;	2nd Marg:	marg.numslices++;
3rd Veg:	veg.numslices++;	1st Cap:	cap.numslices++;
	return veg.numslices;		

```
var 1..7: i;

// assignment i provides value
retval = [0,q1+1,q2+1,q3+1,q4+1,q5+1,q6+1][i];

// assignment i matches
[veg,veg,veg,veg,marg,marg,cap][i] = veg;

// assignment i reaches
[r0,r1,r2,r3,r4,r5,r6][i] = true;
```

The Example Untangled

Translate using the query-based technique

1st Veg:	veg.numslices++;	1st Marg:	marg.numslices++;
2nd Veg:	veg.numslices++;	2nd Marg:	marg.numslices++;
3rd Veg:	veg.numslices++;	1st Cap:	cap.numslices++;
	return veg.numslices;		

```
var 1..4: i;

// assignment i provides value
retval = [0,q1+1,q2+1,q3+1][i];

// assignment i matches
[veg,veg,veg,veg][i] = veg;

// assignment i reaches
[r0,r1,r2,r3][i] = true;
```

The Example Untangled

Translate using the query-based technique

1st Veg:	veg.numslices++;	1st Marg:	marg.numslices++;
2nd Veg:	veg.numslices++;	2nd Marg:	marg.numslices++;
3rd Veg:	veg.numslices++;	1st Cap:	cap.numslices++;
	return veg.numslices;		

```
var 1..4: i;

// assignment i provides value
retval = [0,q1+1,q2+1,q3+1][i];


// assignment i reaches
[r0,r1,r2,r3][i] = true;
```

The Example Untangled

Translate using the query-based technique

1st Veg:	**veg.numslices**++;	1st Marg:	marg.numslices++;
2nd Veg:	veg.numslices++;	2nd Marg:	marg.numslices++;
3rd Veg:	veg.numslices++;	1st Cap:	cap.numslices++;
	return veg.numslices;		

var 1..4: i;

// assignment i provides value
retval = [0,**q1**+1,q2+1,q3+1][i];

// assignment i reaches
[r0,r1,r2,r3][i] = **true**;

Constraints for q1

var 1..4: j;

// assignment j provides value
q1 = [0,q1+1,q2+1,q3+1][j];

// assignment j matches
// assignment j reaches

The Example Untangled

Translate using the query-based technique

1st Veg:	**veg.numslices**++;	1st Marg:	marg.numslices++;
2nd Veg:	veg.numslices++;	2nd Marg:	marg.numslices++;
3rd Veg:	veg.numslices++;	1st Cap:	cap.numslices++;
	return veg.numslices;		

var 1..4: i;

// assignment i provides value
retval = [0,**q1**+1,q2+1,q3+1][i];

// assignment i reaches
[r0,r1,r2,r3][i] = **true**;

Constraints for q1

q1 = 0;

The Example Untangled

Translate using the query-based technique

1st Veg:	veg.numslices = **0**+1;	1st Marg:	marg.numslices++;
2nd Veg:	veg.numslices++;	2nd Marg:	marg.numslices++;
3rd Veg:	veg.numslices++;	1st Cap:	cap.numslices++;
	return veg.numslices;		

var 1..4: i;

// assignment i provides value
retval = [0, **0**+1,q2+1,q3+1][i];

// assignment i reaches
[r0,r1,r2,r3][i] = **true**;

Constraints for q1

q1 = 0;

The Example Untangled

Translate using the query-based technique

1st Veg:	veg.numslices = 1;	1st Marg:	marg.numslices = 1;
2nd Veg:	veg.numslices = 2;	2nd Marg:	marg.numslices = 2;
3rd Veg:	veg.numslices = 3;	1st Cap:	cap.numslices = 1;
	return veg.numslices;		

```
var 1..4: i;

// assignment i provides value
retval = [0,1,2,3][i];

// assignment i reaches
[r0,r1,r2,r3][i] = true;
```

Comparison: unwound vs. untangled

```
var 1..4: i;

// assignment i provides value
retval = [0,q1+1,q2+1,q3+1][i];

// assignment i matches
[veg,pt1,pt2,pt3][i] = veg;

// assignment i reaches
[r0,r1,r2,r3][i] = true;
```

```
var 1..4: i;

// assignment i provides value
retval = [0,1,2,3][i];

// assignment i reaches
[r0,r1,r2,r3][i] = true;
```

The untangled version is much simpler!

- We know exactly which assignments match.
- We don't need q1, q2 and q3.

But we need a new way of defining *reaches*.

Redefining *reaches*

An assignment reaches a query if it happens before the query and no other matching assignment happens between them.

When loops are unwound

- Some assignments may be skipped, but the relative order of any pair is known.
- So happens before and happens between can be defined just using happens.

When loops are untangled

- The relative order is no longer known.
- We also need to properly constrain which untangled iterations happen.

Current solution

Still unwind parts of the body used to define the label query.

L1 = slice1.pizzatype L2 = slice2.pizzatype L3 = slice3.pizzatype

Link each untangled iteration to an unwound one.

			doesn't happen	**1st**	**2nd**	**3rd**	
UnwoundForVeg1	$\in$	$\{$	0	1	2	3	$\}$
UnwoundForVeg2	$\in$	$\{$	0	1	2	3	$\}$
UnwoundForVeg3	$\in$	$\{$	0	1	2	3	$\}$
UnwoundForMarg1	$\in$	$\{$	0	1	2	3	$\}$
UnwoundForMarg2	$\in$	$\{$	0	1	2	3	$\}$
UnwoundForCap1	$\in$	$\{$	0	1	2	3	$\}$

Use this link to define *happens* and *before*.

Veg1Happens = (UnwoundForVeg1 $>$ 0)

Veg1BeforeVeg2 = Veg1Happens $\wedge$ (UnwoundForVeg1 $<$ UnwoundForVeg2)

Current solution

Add constraints to ensure the correct iterations happen.

Veg2Happens $\rightarrow$ Veg1BeforeVeg2

[veg, L1, L2, L3][UnwoundForVeg1] = veg

Card([UnwoundForVeg1, UnwoundForVeg2, UnwoundForVeg3,
UnwoundForMarg1, UnwoundForMarg2, UnwoundForCap1],
[0, 1, 2, 3], [3, 1, 1, 1])

Then if L1 $\in$ {**`Veg`**, **`Marg`**}, L2 $\in$ {**`Veg`**, **`Marg`**}, L3 $\in$ {**`Veg`**, **`Cap`**}:

			doesn't happen	**1st**	**2nd**	**3rd**	
UnwoundForVeg1	$\in$	{	0	1	2	3	}
UnwoundForVeg2	$\in$	{	0		2	3	}
UnwoundForVeg3	$\in$	{	0			3	}
UnwoundForMarg1	$\in$	{	0	1	2		}
UnwoundForMarg2	$\in$	{	0		2		}
UnwoundForCap1	$\in$	{	0			3	}

Back to the example

For our example *reaches* can be defined using *happens*.

var 1..4: i;

// assignment i provides value
retval = [0,1,2,3][i];

// assignment i reaches
[r0,r1,r2,r3][i] = **true**;

r0 = ¬Veg1Happens
r1 = Veg1Happens ∧ ¬Veg2Happens
r2 = Veg2Happens ∧ ¬Veg3Happens
r3 = Veg3Happens

1st Veg: veg.numslices = 1;

2nd Veg: veg.numslices = 2;

3rd Veg: veg.numslices = 3;

return veg.numslices;

Summary

General loop untangling process:

- Choose a label query.
- Create copies based on that.
- Use the same constraints..
- ..but with new *reaches* expressions.
- Unwind everything underneath and link together.

Outline

1. Motivation
   - our purpose for reasoning about procedural code
2. Background
   - brief introduction to constraint programming (CP)
3. Standard approaches
   - explanation of how CP + SMT are typically applied to this problem
4. Query-Based approach
   - a better CP encoding
5. (Bounded) Loops
   - Unwinding
   - Untangling
6. Global Constraint for Reaching Definitions
   - our current work
7. Conclusion

Global Constraint for Reaching Definitions

Current work: replace reaches constraints with a global

field reference: queryobj.field

assignments:

$$\begin{aligned} &\text{cond}_1 : \text{obj}_1.\text{field} := \text{expr}_1 \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots \\ &\text{cond}_n : \text{obj}_n.\text{field} := \text{expr}_n \end{aligned}$$

$$\Downarrow$$

constraints:

$$\begin{aligned} \text{queryresult} &= [\text{expr}_1, \ldots, \text{expr}_n][\text{indexvar}] \\ \text{queryobj} &= [\text{obj}_1, \ldots, \text{obj}_n][\text{indexvar}] \\ \text{true} &= [\text{reachs}_1, \ldots, \text{reachs}_n][\text{indexvar}] \\ \text{reachs}_1 &= (\text{cond}_1 \wedge (\text{cond}_2 \rightarrow \text{obj}_1 \neq \text{obj}_2) \cdots) \\ &\ldots \\ \text{reachs}_{n-1} &= (\text{cond}_{n-1} \wedge (\text{cond}_n \rightarrow \text{obj}_{n-1} \neq \text{obj}_n) \\ \text{reachs}_n &= \text{cond}_n \end{aligned}$$

Global Constraint for Reaching Definitions

NICTA THE UNIVERSITY OF MELBOURNE

The propagator for the new global constraint will..

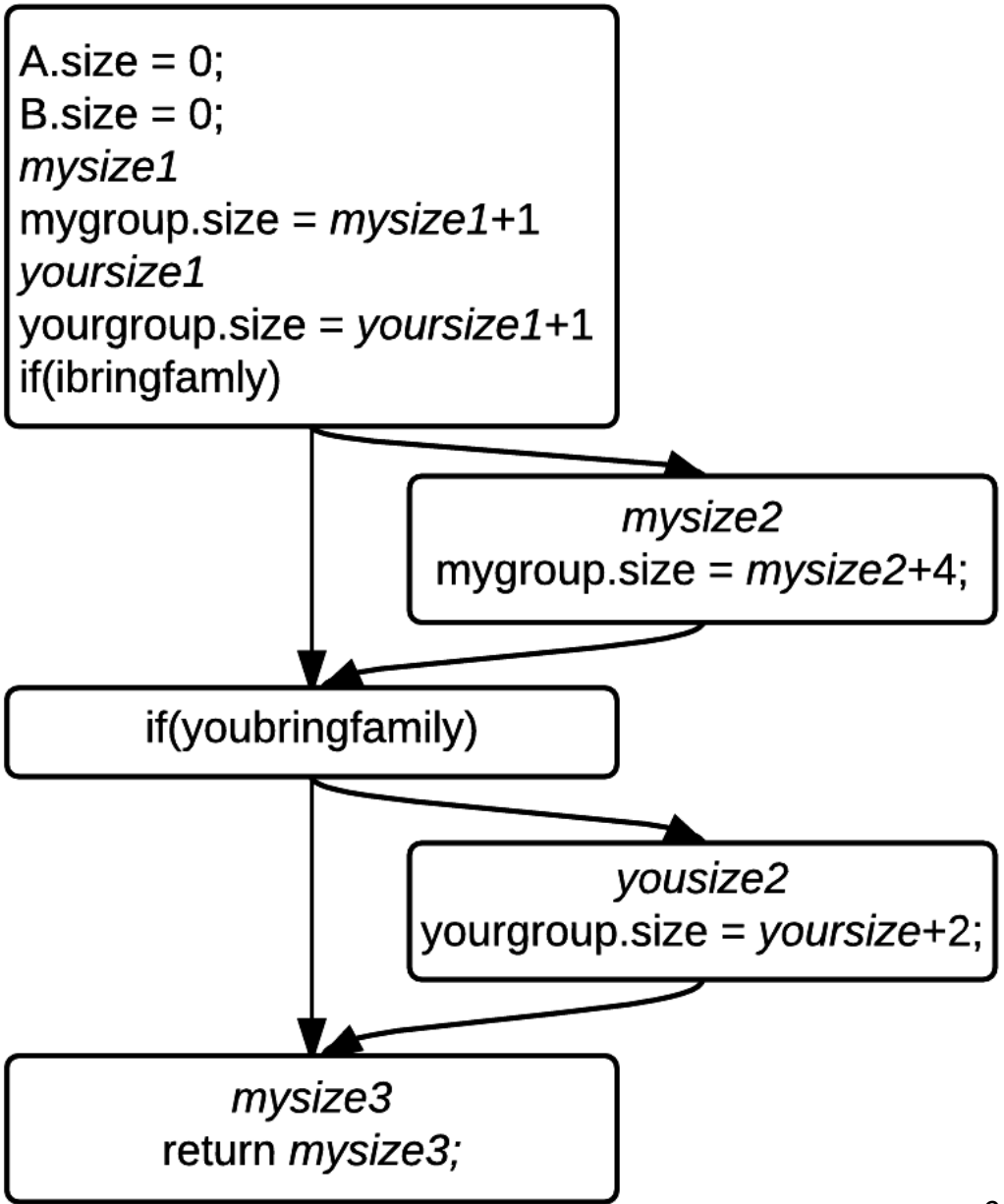
- Remove from *indexvar* the index of changes which cannot reach the query.
- Make deductions about statements which must or must not be reached by execution.
- Achieve some propagation provided by the *sum* constraint.

To do this several different algorithms are used.

- Represent possible execution paths with an explicit graph.
- Existing path propagation algorithms keep this consistent.
- Specialised algorithm uses condensed version of graph to compute which assignments may reach which queries.
- Another algorithm combines the information from both of these to detect nodes which must or must not happen.

Execution Graph - Example

```
mygroup.size++;
yourgroup.size++;
if(ibringfamily)
   mygroup.size += 4;
if(youbringfamily)
   yourgroup.size += 2;
```



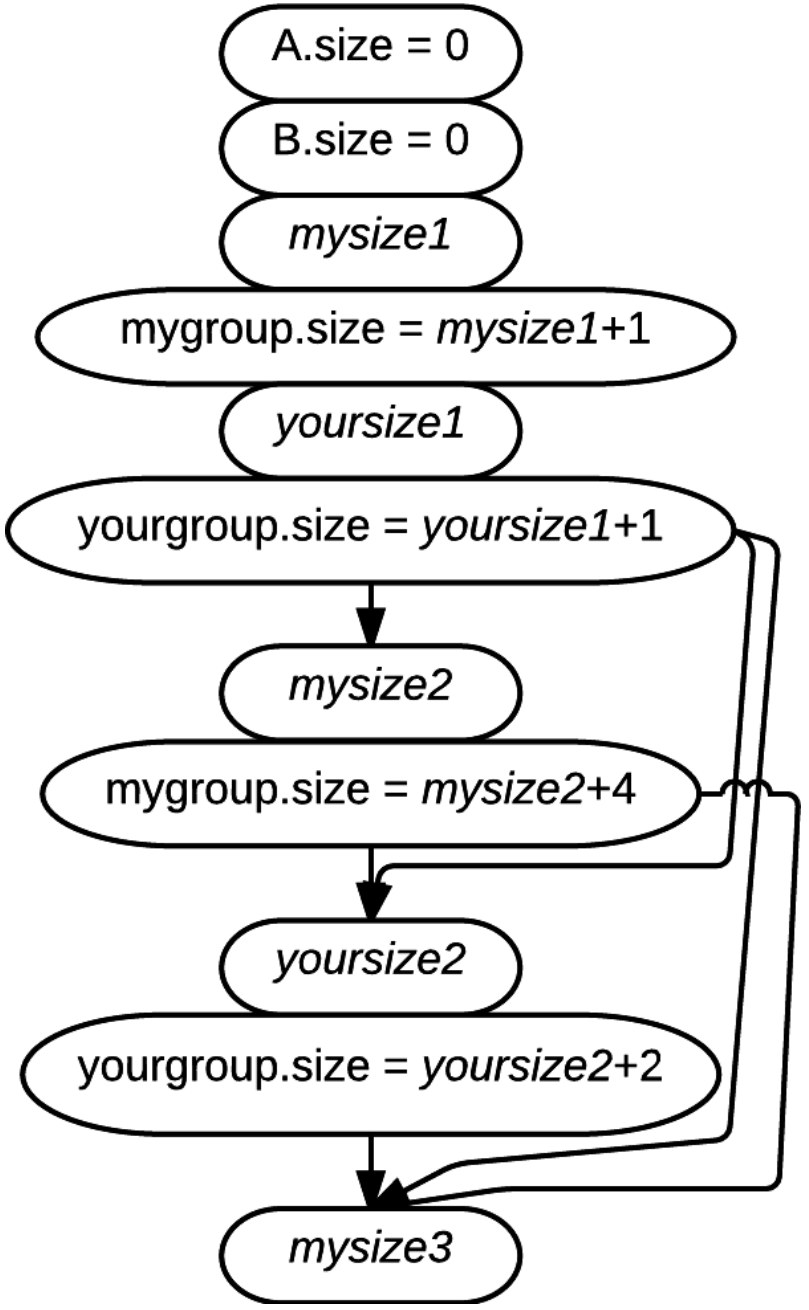
The path propagator keeps this graph consistent with the values of *ibringfamily* and *youbringfamily*.

Condensed Graph - Example

```
mygroup.size++;
yourgroup.size++;
if(ibringfamily)
   mygroup.size += 4;
if(youbringfamily)
   yourgroup.size += 2;
```

We create a condensed graph for each variable/field.

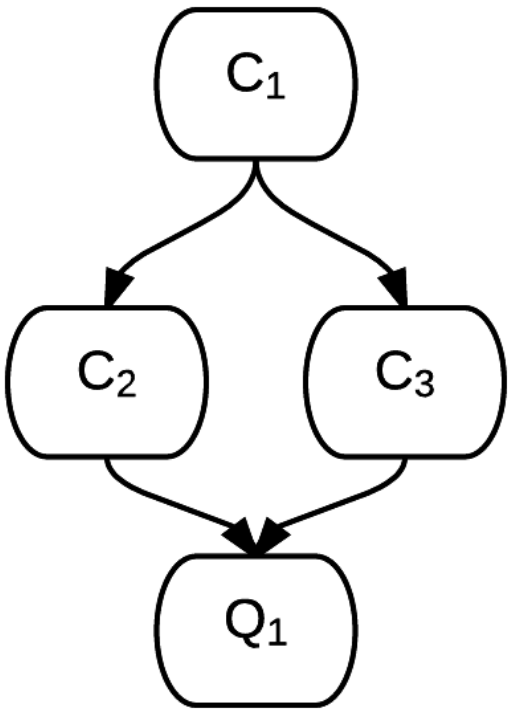
This has an edge $a - b$ if there is a path from a to b without going through another relevant node.



Reaching Definitions Algorithm

Goal:

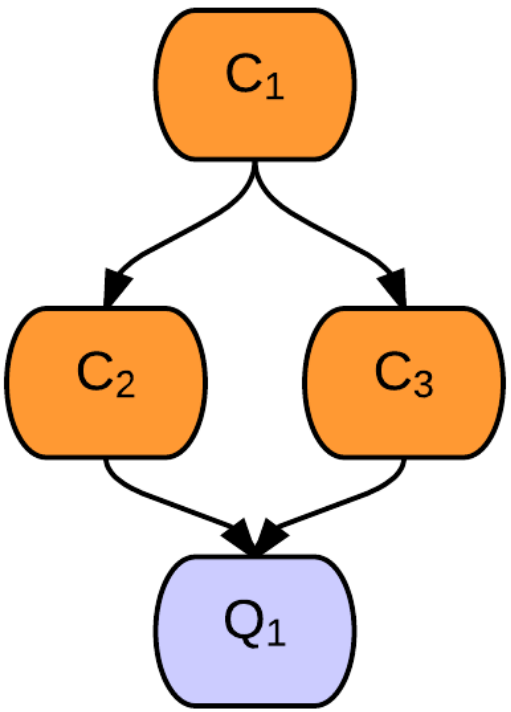
- Find changes which cannot reach the query because they will be overwritten.



Reaching Definitions Algorithm

Goal:

- Find changes which cannot reach the query because they will be overwritten.

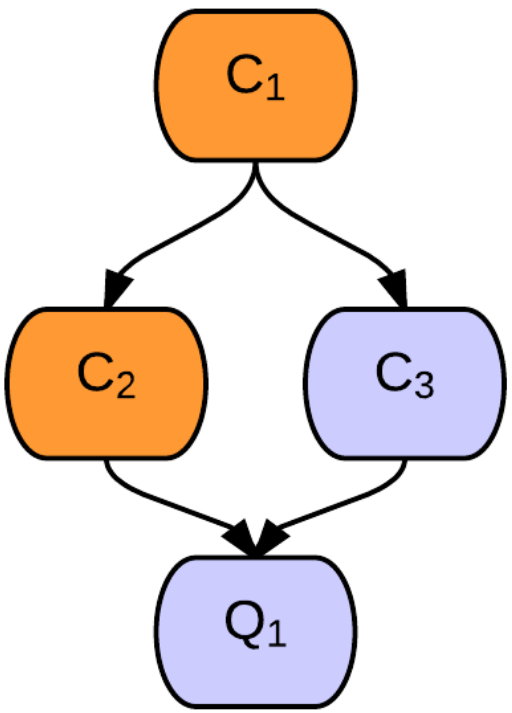


- If C_1 matches C_2 and C_3, then C_1 cannot reach Q_1.

Reaching Definitions Algorithm

Goal:

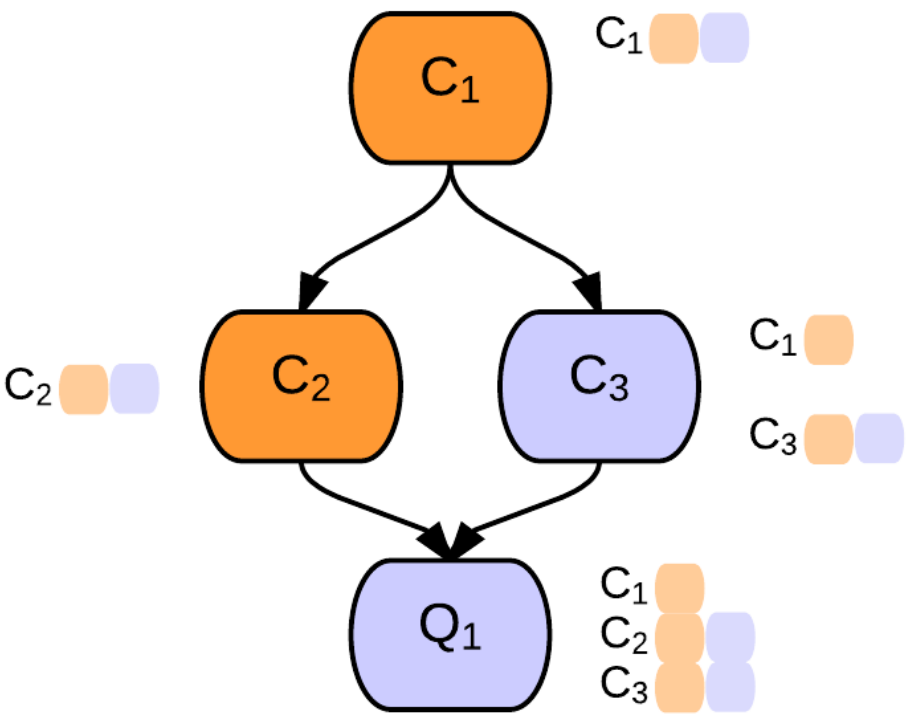
- Find changes which cannot reach the query because they will be overwritten.



- If C_1 matches C_2 and C_3, then C_1 cannot reach Q_1.
- Also, if C_1 matches C_2 and Q_1 matches C_3, then C_1 does not reach Q_1.

Reaching Definitions Algorithm

- Divide nodes into match classes (groups of definitely matching nodes).
- For each node store which changes reach there for which match classes.



Update iteratively:

C reaches here for class *m* if

- this node is *C*

or

- *C* reaches at least one predecessor for *m*
- this is not a change in the same class as *C* or a change of class *m*

Reaching Definitions Algorithm - Problem

This algorithm is effective, but too strict.

- If a query is not on the execution path, then it will not be reached by any change.
- This means the index variable has no possible value.
- In a CP solver this automatically causes failure (stop searching here and backtrack).

The solution is to use option types.

- We declare the index variable to be an optional int.
- This means it is allowed to take a special value $\top$.
- We constrain it to only do so if the query does not happen.
- Option types can be automatically converted to normal variables, but this weakens propagation, so we have implemented native handling of these in the solver.

Detecting Impossible Nodes

By combining path and reaching definitions information..

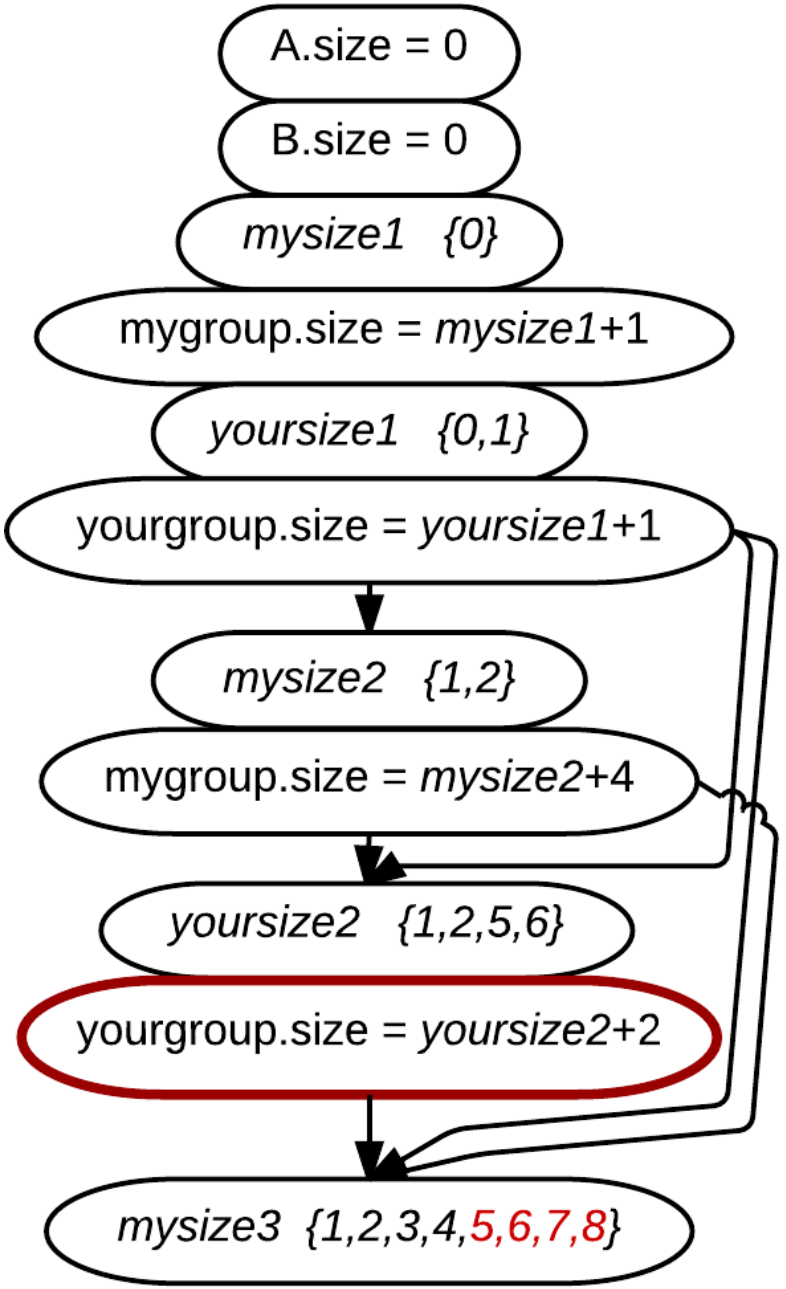
- We can reduce the domains of assigned values.
- We can detect nodes which must not happen.

If every path forward from a change encounters a matching query before an overwriting change, then

- Those query results give the possible assigned values.
- If this leaves no possible assigned value, then the change does not happen.

We look for easy-to-find cases: when all immediate successors in the condensed graph are matching queries.

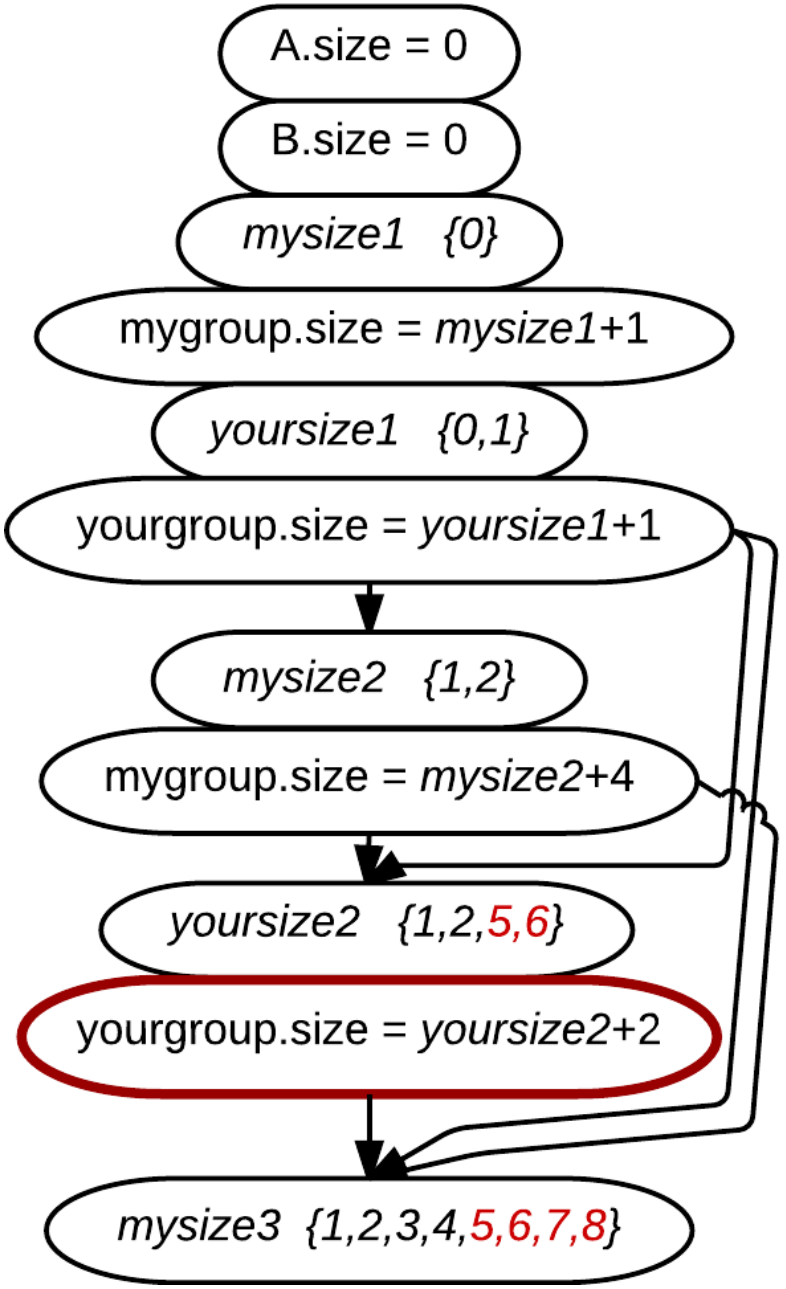
Detecting Impossible Nodes - Example



Imagine as before that we know $mysize3 < 5$ and $mygroup = yourgroup$.

- The only edge forward from the last change goes to the final query, which matches because $mygroup = yourgroup$.

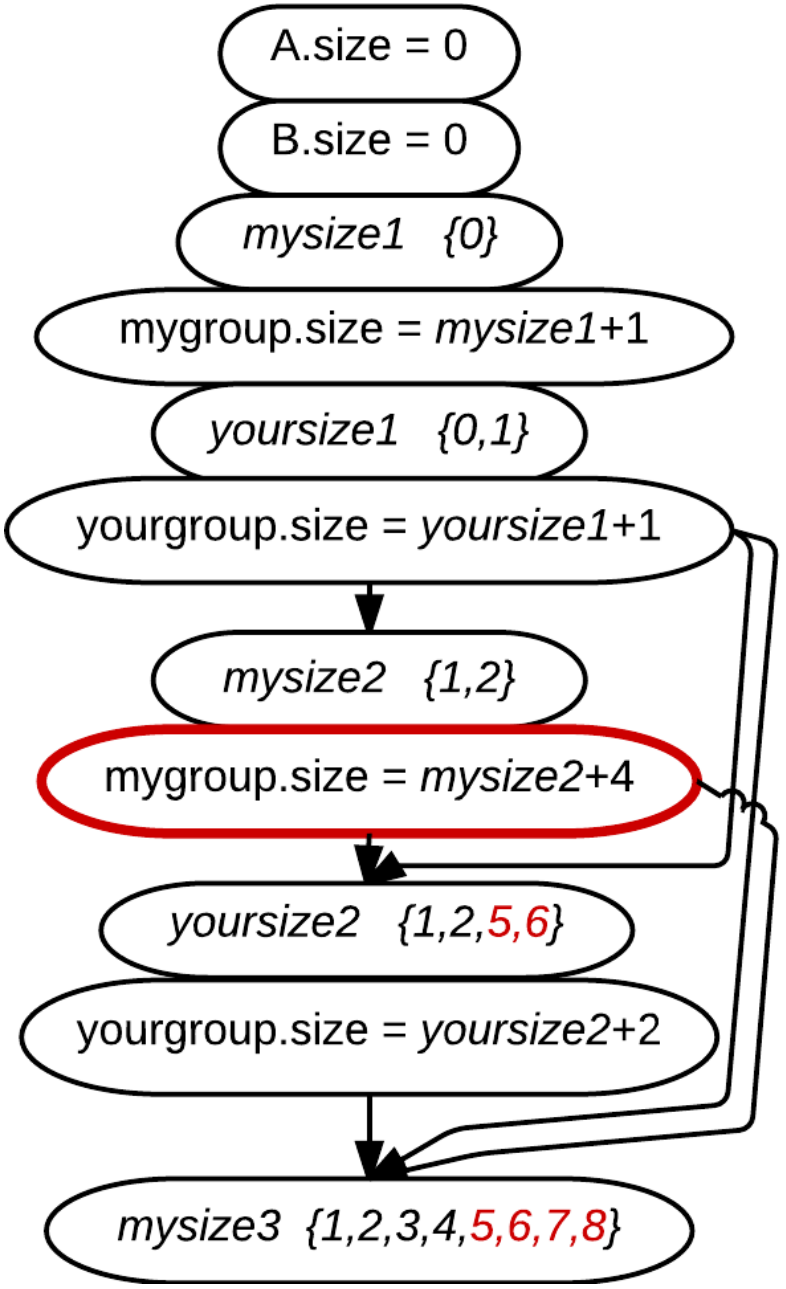
Detecting Impossible Nodes - Example



Imagine as before that we know $mysize3 < 5$ and $mygroup = yourgroup$.

- The only edge forward from the last change goes to the final query, which matches because $mygroup = yourgroup$.
- So $yoursize2 + 2$ must take a value in $\{1, 2, 3, 4\}$.
- Therefore $yoursize2$ cannot be 5 or 6.

Detecting Impossible Nodes - Example



- Both edges forward from the previous change also lead to matching queries.
- So *mysize2* $+ 4$ must be in $\{1, 2\} \cup \{1, 2, 3, 4\}$.
- But *mysize2* $\in \{1, 2\}$.
- Since there is no possible assigned value, this change cannot happen.
- The path propagator will conclude that *ibringfamily* must be false.

Detecting Required Nodes

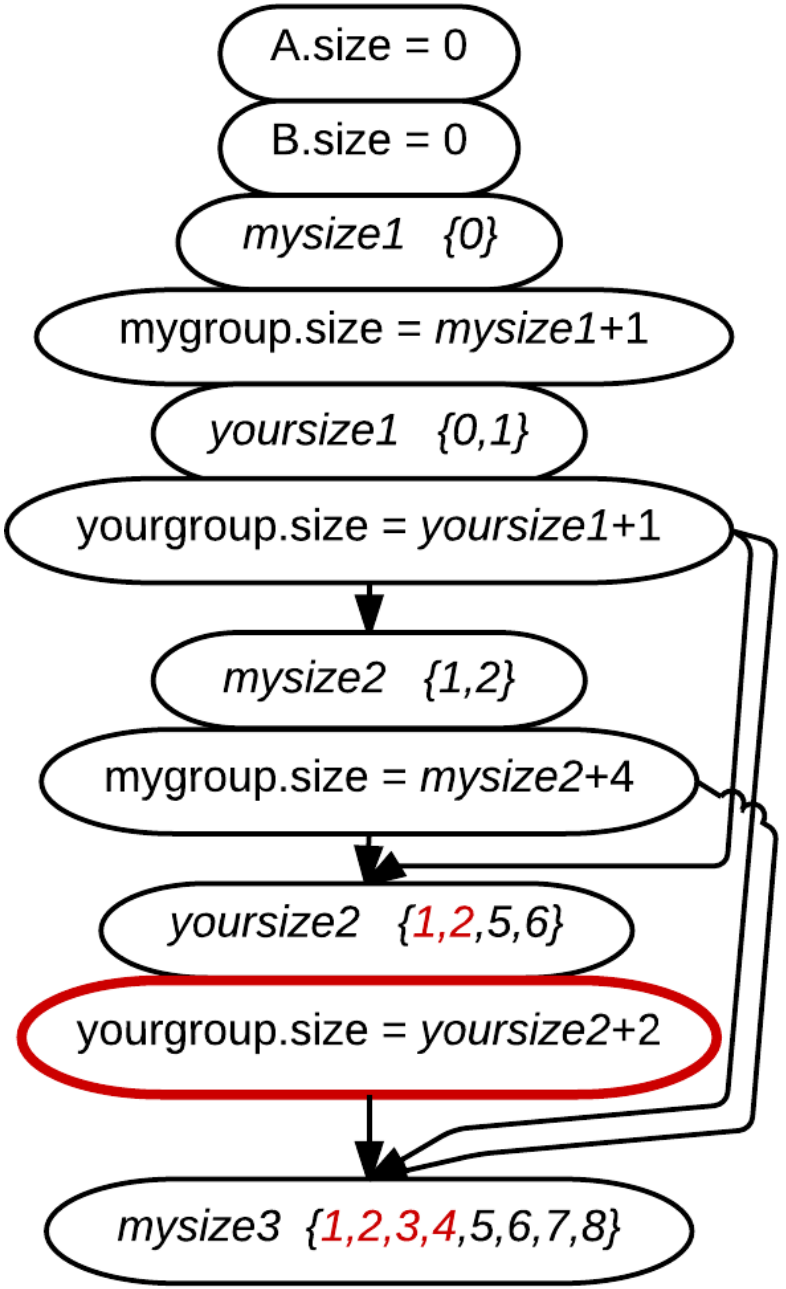
By combining path and reaching definitions information..

- We can discover new dominance relationships.
- We can use these to detect when nodes must happen.

If some change dominates all other possibly reaching changes for a query, then

- The change dominates the query.
- If the query must happen, the change also must happen.

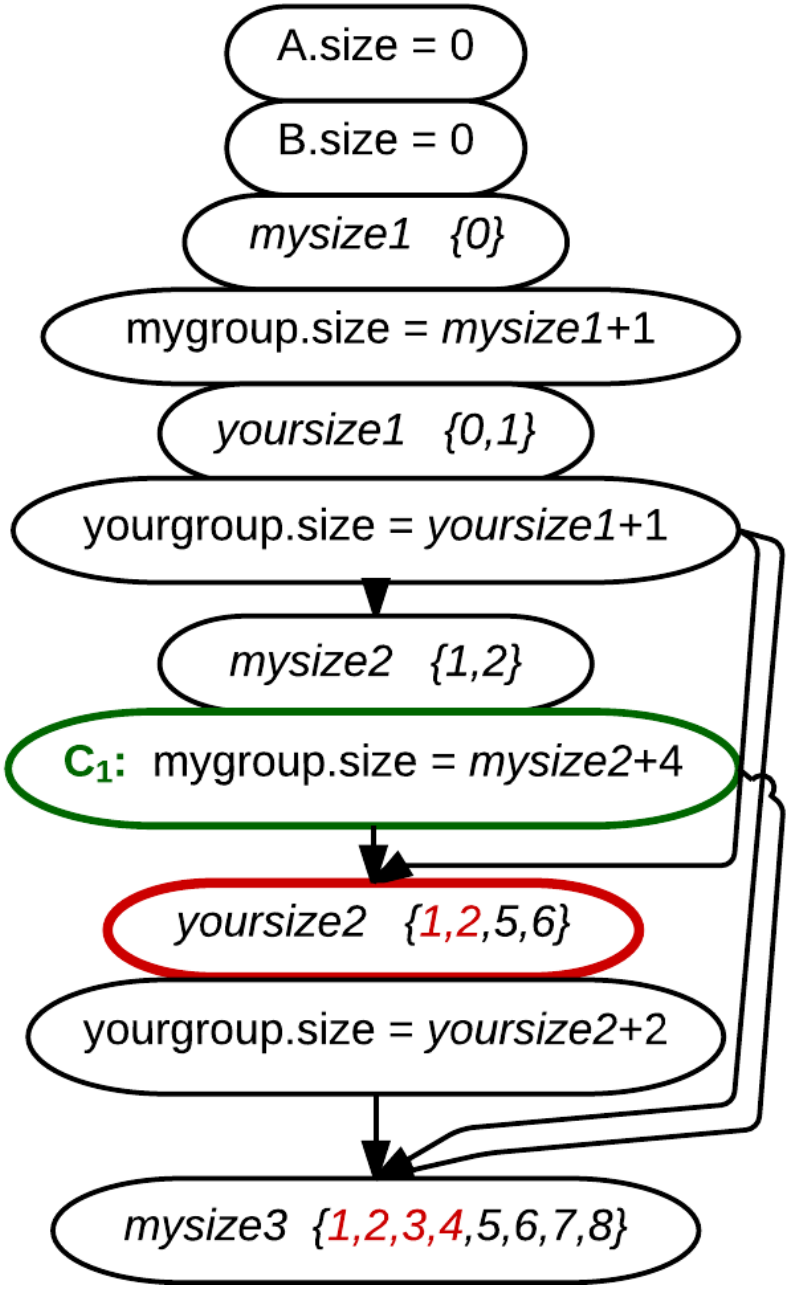
Detecting Required Nodes - Example



Imagine this time that we know $returnval \geq 5$ and $mygroup = yourgroup$.

- As before we can reduce the domain of *yoursize*2.

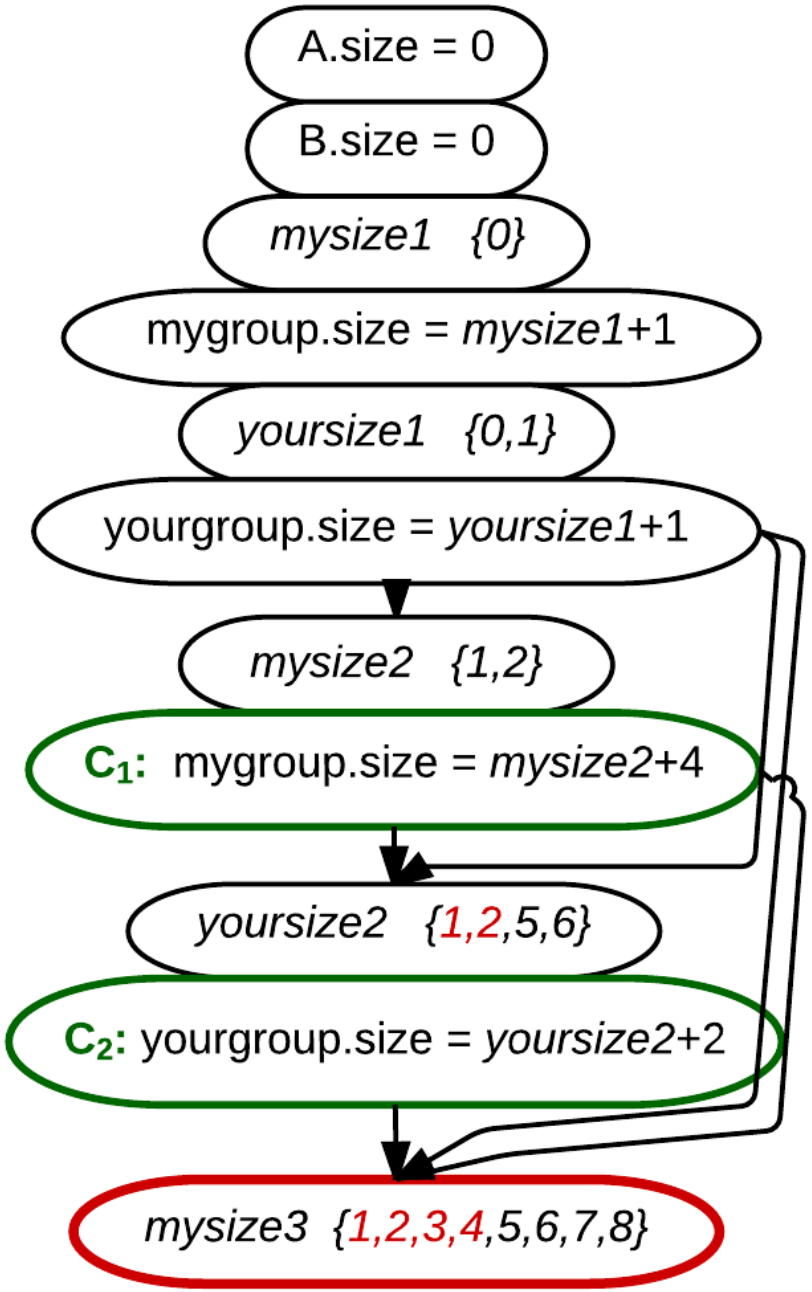
Detecting Required Nodes - Example



Imagine this time that we know *returnval* $\geq$ 5 and *mygroup* = *yourgroup*.

- As before we can reduce the domain of *yoursize*2.
- Then, we see that *yoursize*2 can only be reached by change C_1.
- Therefore C_1 dominates *yoursize*2.

Detecting Required Nodes - Example



- The final query can only be reached by C_1 or C_2.
- C_1 dominates *yoursize2* and therefore C_2.
- Since C_1 dominates all possible reaching changes for *mysize3*, it must dominate *mysize3*.
- *mysize3* must happen, so C_1 must happen.
- The path propagator will conclude that *ibringfamily* must be true.

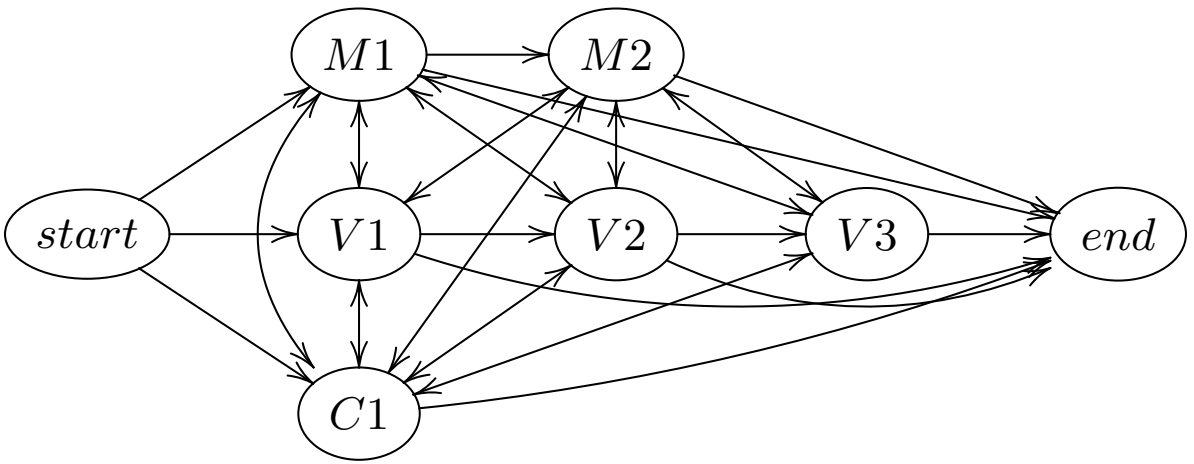
This is another example of propagation provided by *sum*.

Reaching-Definitions Global and Untangling

Global constraint reasons about “dominance” directly!

- Untangling generates dominances
- Untangling propagation is improved by dominances

Tangle Graph: execution paths

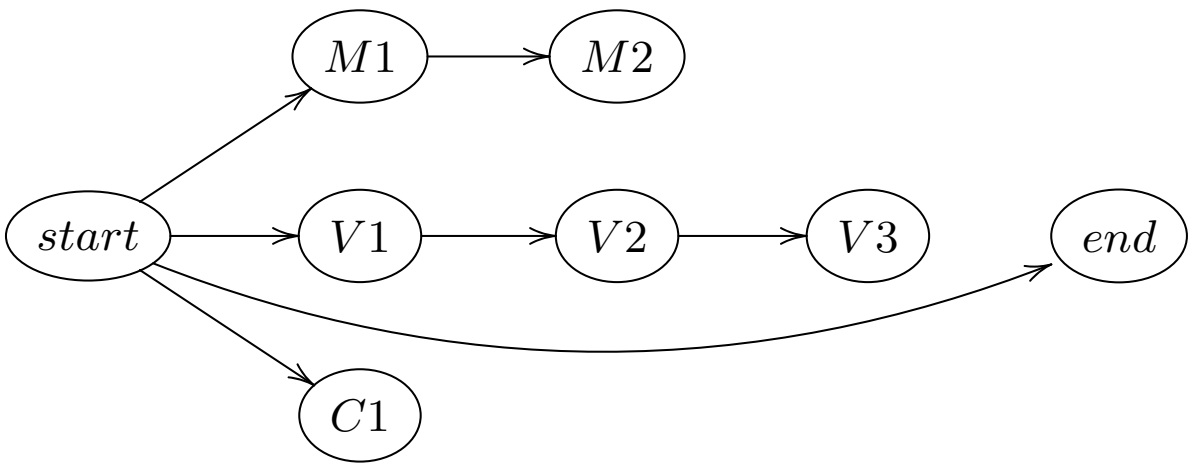


Reaching-Definitions Global and Untangling

Global constraint reasons about “dominance” directly!

- Untangling generates dominances
- Untangling propagation is improved by dominances

Dominance Graph: (transitively reduced) dominance



Outline

1. Motivation
   - our purpose for reasoning about procedural code
2. Background
   - brief introduction to constraint programming (CP)
3. Standard approaches
   - explanation of how CP + SMT are typically applied to this problem
4. Query-Based approach
   - a better CP encoding
5. (Bounded) Loops
   - Unwinding
   - Untangling
6. Global Constraint for Reaching Definitions
   - our current work
7. Conclusion

Summary

- Experimental results have shown that (for our problems):
  - Query-based CP $\gg$ Standard CP $>$ SMT
  - Special cases (e.g. *sum*) mean some benchmarks achieve propagation as strong as a hand written model.
- Global reaching-definitions constraint
  - produces strong propagation of special cases
  - in a general way
  - although it does not cover all cases.
- We have not done time comparisons for the global constraint yet (it's work-in-progress).

Further Work

Efficiency and Trade-offs

- Stronger propagation does not always lead to better overall performance.
- We need to optimise our implementation and investigate how much propagation strength pays off in our case.

Equality Propagation

- When there is a single reaching change we have deduced an equality between two variables.
- We also use equality information to detect matchingness.
- Currently we can only detect that two variables are equal if they are identical or fixed to the same value.
- We would like to improve on this by implementing a union-find based equality propagator.

Questions?

For more details:

Optimisation modelling for software developers.
K. Francis, S. Brand, and P. J. Stuckey. *CP 2012.*

Modelling destructive assignments.
K. Francis, J. Navas, and P. J. Stuckey. *CP 2013.*

Modelling with Option Types in MiniZinc. C. Mears, A. Schutt, P. J. Stuckey, G. Tack, K. Marriott, M. Wallace. *CP-AI-OR 2014.*

Explaining Circuit Propagation. K. Francis and P. J. Stuckey. *Constraints, Jan 2014.*

Loop untangling. K. Francis and P. J. Stuckey. *CP 2014.*

Nothing published about reaching definitions propagator yet so you'll have to talk to me!