Status of the New Backend

Karim Abdelhak, Philip Hannebohm, Bernhard Bachmann

University of Applied Sciences Bielefeld
Bielefeld, Germany

FH Bielefeld
University of
Applied Sciences

Proper Hybrid Models for Smarter Vehicles

https://phymos.de

The presented work is part of the PHyMoS project, supported by the German Federal Ministry for Economic Affairs and Climate Action.

Supported by:

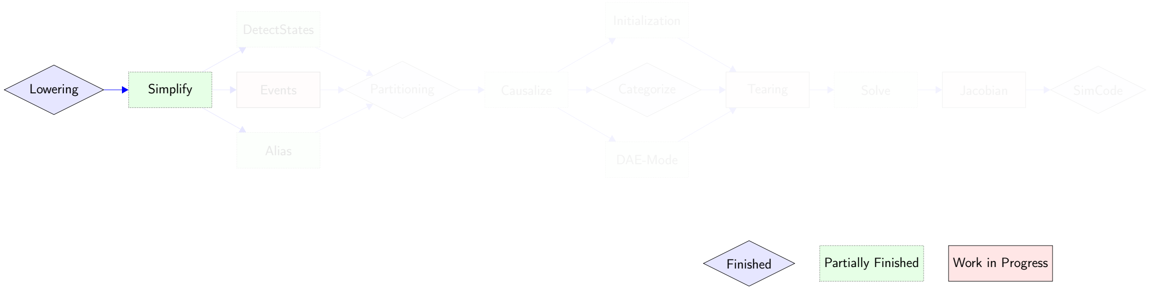
Federal Ministry
for Economic Affairs
and Climate Action

on the basis of a decision
by the German Bundestag

1. Overview

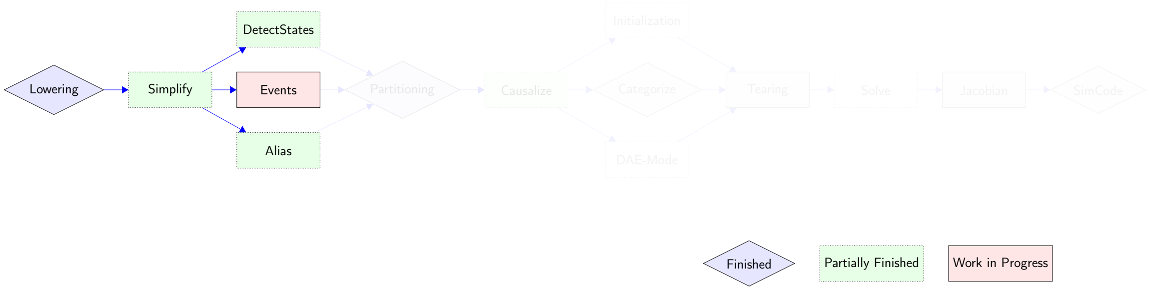
Backend Modules

Status on Array-Handling



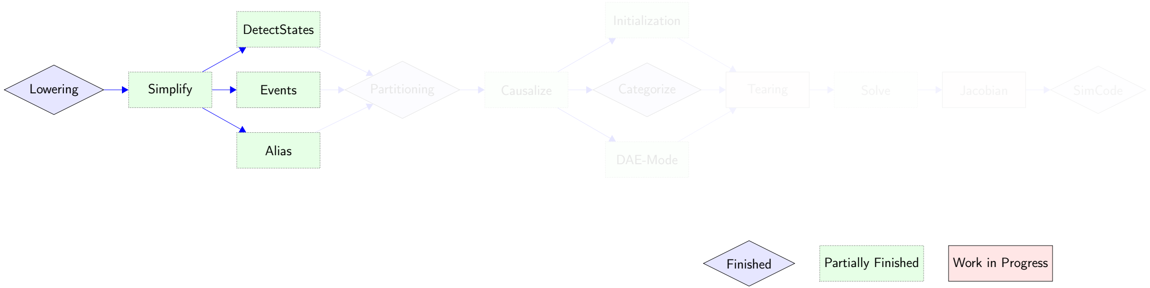
Backend Modules

Status on Array-Handling



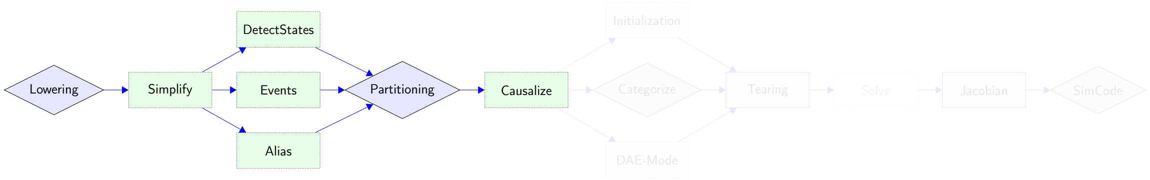
Backend Modules

Status on Array-Handling



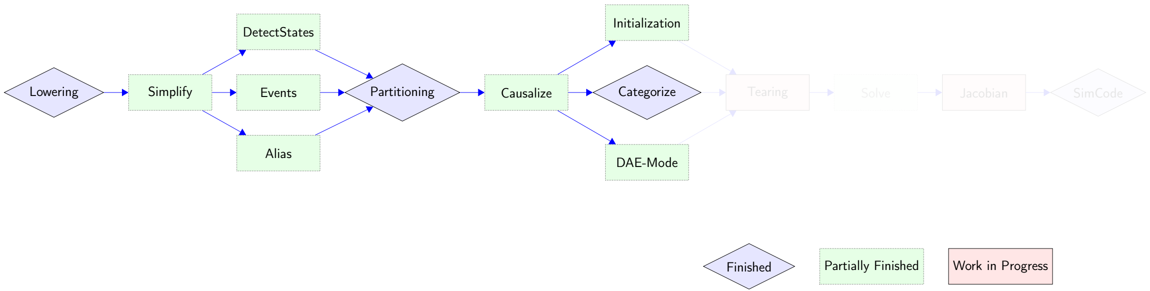
Backend Modules

Status on Array-Handling



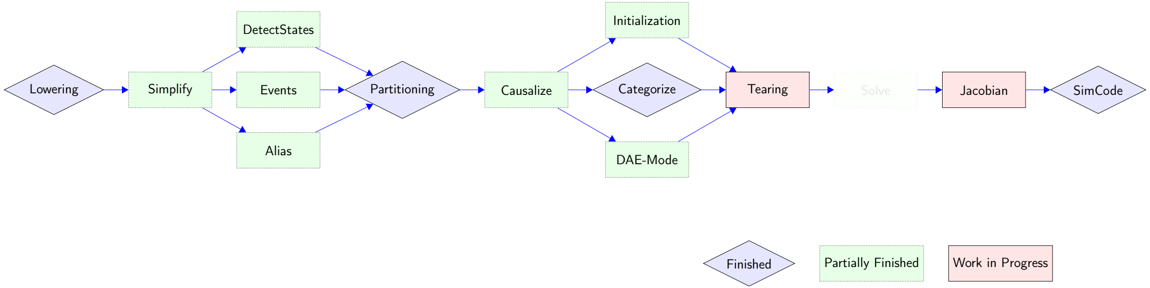
Backend Modules

Status on Array-Handling



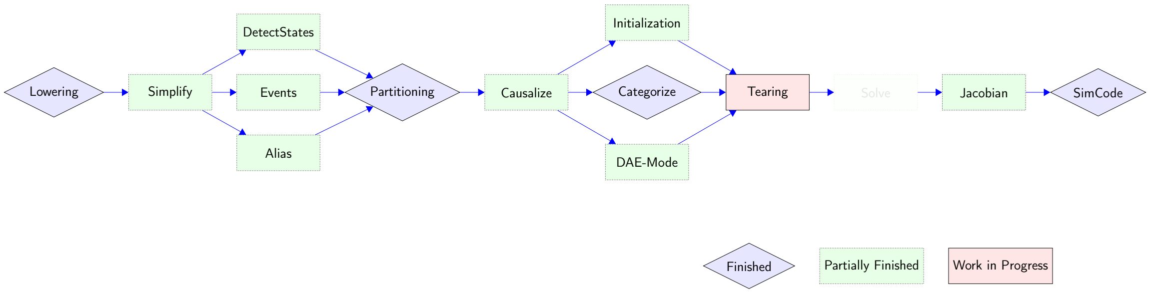
Backend Modules

Status on Array-Handling



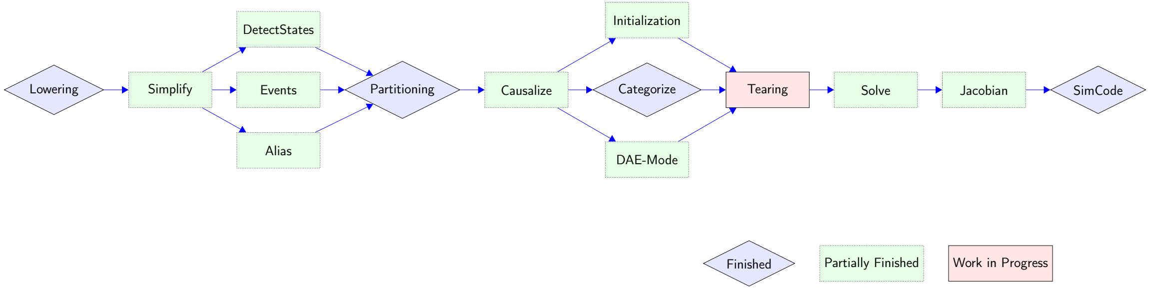
Backend Modules

Status on Array-Handling



Backend Modules

Status on Array-Handling



2. Two Step Sorting

Algorithm Outline

1. Pseudo-Array Matching
2. Scalar Sorting
3. Merge algebraic loop nodes
4. Merge array nodes
5. Array sorting
6. Sort array nodes internally

Advantages

- Force arrays to be solved in succession if possible
- Prevent entwining of arrays as much as possible

Algorithm Outline

1. Pseudo-Array Matching
2. Scalar Sorting
3. Merge algebraic loop nodes
4. Merge array nodes
5. Array sorting
6. Sort array nodes internally

Advantages

- Force arrays to be solved in succession if possible
- Prevent entwining of arrays as much as possible

Algorithm Outline

1. Pseudo-Array Matching
2. Scalar Sorting
3. Merge algebraic loop nodes
4. Merge array nodes
5. Array sorting
6. Sort array nodes internally

Advantages

- Force arrays to be solved in succession if possible
- Prevent entwining of arrays as much as possible

Algorithm Outline

1. Pseudo-Array Matching
2. Scalar Sorting
3. Merge algebraic loop nodes
4. Merge array nodes
5. Array sorting
6. Sort array nodes internally

Advantages

- Force arrays to be solved in succession if possible
- Prevent entwining of arrays as much as possible

Algorithm Outline

1. Pseudo-Array Matching
2. Scalar Sorting
3. Merge algebraic loop nodes
4. Merge array nodes
5. Array sorting
6. Sort array nodes internally

Advantages

- Force arrays to be solved in succession if possible
- Prevent entwining of arrays as much as possible

Algorithm Outline

1. Pseudo-Array Matching
2. Scalar Sorting
3. Merge algebraic loop nodes
4. Merge array nodes
5. Array sorting
6. Sort array nodes internally

Advantages

- Force arrays to be solved in succession if possible
- Prevent entwining of arrays as much as possible

Algorithm Outline

1. Pseudo-Array Matching
2. Scalar Sorting
3. Merge algebraic loop nodes
4. Merge array nodes
5. Array sorting
6. Sort array nodes internally

Advantages

- Force arrays to be solved in succession if possible
- Prevent entwining of arrays as much as possible

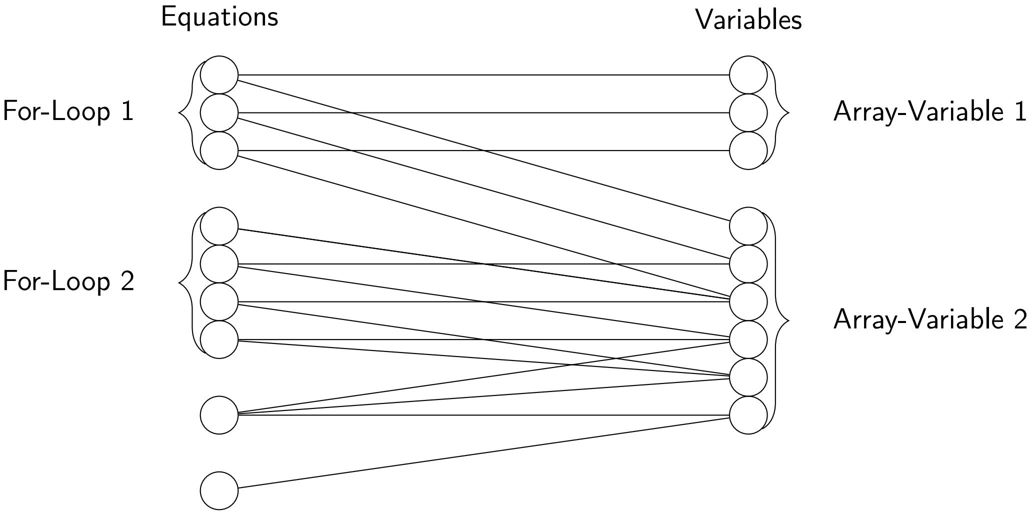
Algorithm Outline

1. Pseudo-Array Matching
2. Scalar Sorting
3. Merge algebraic loop nodes
4. Merge array nodes
5. Array sorting
6. Sort array nodes internally

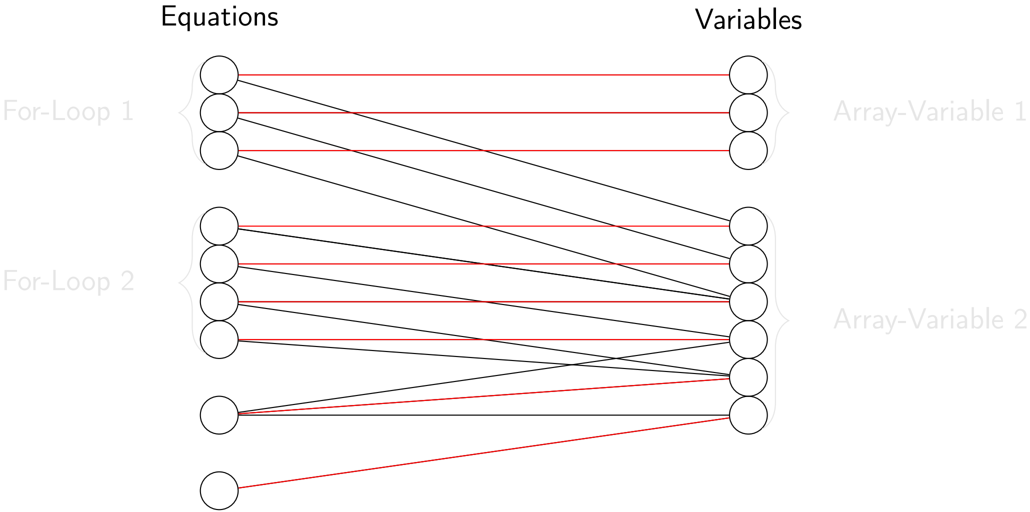
Advantages

- Force arrays to be solved in succession if possible
- Prevent entwining of arrays as much as possible

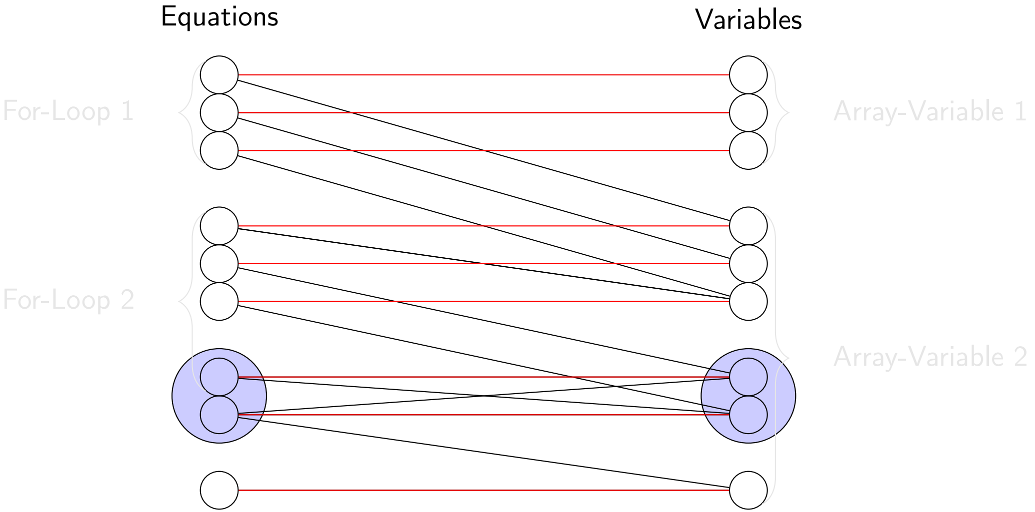
Abstract Graph



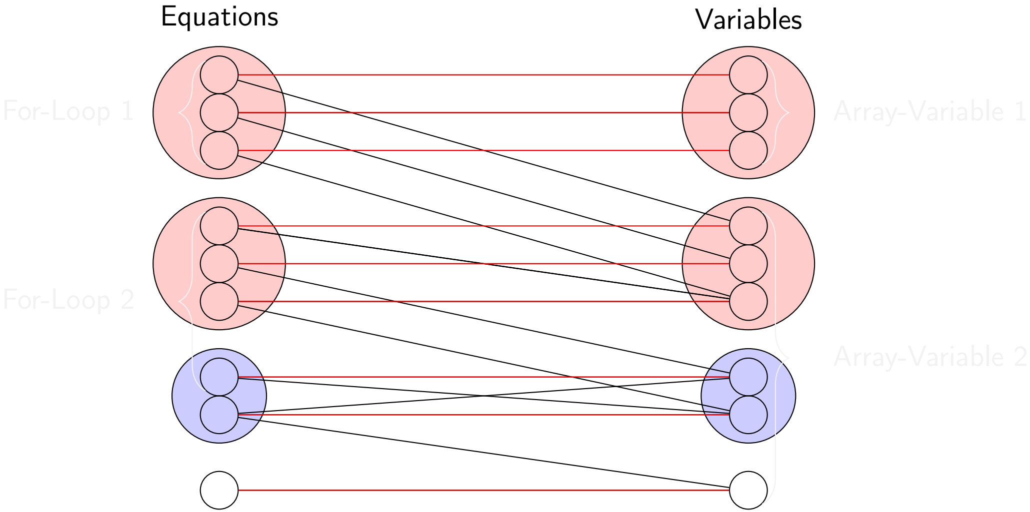
Matching


Equations
Variables
For-Loop 1
For-Loop 2
Array-Variable 1
Array-Variable 2

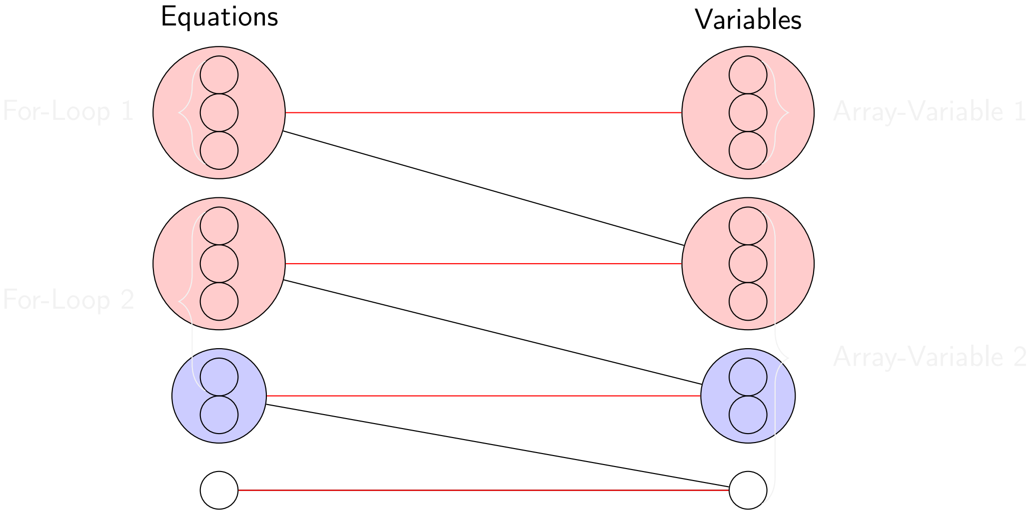
Merge algebraic loop nodes

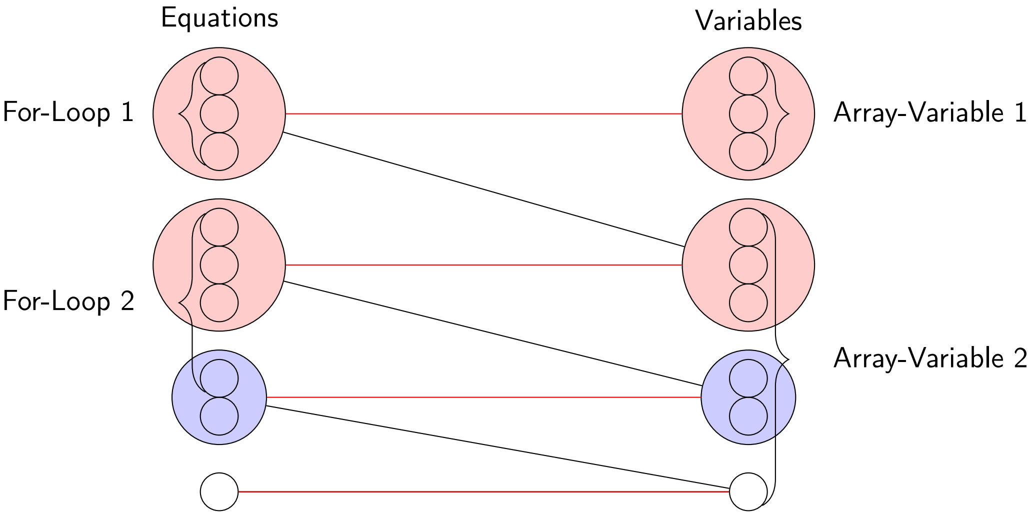


Merge array nodes



Merge edges


Equations
Variables
For-Loop 1
For-Loop 2
Array-Variable 1
Array-Variable 2


Equations
Variables
For-Loop 1
Array-Variable 1
For-Loop 2
Array-Variable 2

3. Generalized For-Loops

Example: Diagonal Slice Model

```
model diagonal_slice_for1
  Real x[4,4];
  Real y[4];
equation
  for i in 1:4 loop
    x[i,i] = i*cos(time);
  end for;
  for i in 1:4, j in 1:4 loop
    x[i,j] = y[j] + i*sin(j*time);
  end for;
end diagonal_slice_for1;
```

Expected Results

- The first for-loop will be solved for the diagonal elements of x
- The second for-loop will be split up into two for-loops:
  1. $i \neq j$ solves the remaining non-diagonal elements of x
  2. $i = j$ solves y

Example: Diagonal Slice Model

```
model diagonal_slice_for1
  Real x[4,4];
  Real y[4];
equation
  for i in 1:4 loop
    x[i,i] = i*cos(time);
  end for;
  for i in 1:4, j in 1:4 loop
    x[i,j] = y[j] + i*sin(j*time);
  end for;
end diagonal_slice_for1;
```

Expected Results

- The first for-loop will be solved for the diagonal elements of x
- The second for-loop will be split up into two for-loops:
  1. $i \neq j$ solves the remaining non-diagonal elements of x
  2. $i = j$ solves y

Example: Diagonal Slice Model

BLT-Blocks after Solve (-d=bltdump)

```
--- Alias of INI[1 | 1] ---
BLOCK 1: Generic Component (status = Solve.EXPLICIT)
----------------------------------------
### Variable:
        x[i, i]
### Equation:
        [FOR-] (4) ($RES_SIM_2)
        [----] for i in 1:4 loop
        [----]   [SCAL] (1) x[i, i] = CAST(Real, i) * cos(time) ($RES_SIM_3)
        [----] end for;
         slice: {3, 2, 1, 0}
```

Example: Diagonal Slice Model

BLT-Blocks after Solve (-d=bltdump)

```
--- Alias of INI[1 | 2] ---
BLOCK 2: Generic Component (status = Solve.EXPLICIT)
----------------------------------------
### Variable:
        y[j]
### Equation:
        [FOR-] (16) ($RES_SIM_0)
        [----] for {i in 1:4, j in 1:4} loop
        [----]   [SCAL] (1) y[j] = -(CAST(Real, i) * sin(CAST(Real, j) * time) - x[i, j]) (
            $RES_SIM_1)
        [----] end for;
         slice: {15, 10, 5, 0}
```

Example: Diagonal Slice Model

BLT-Blocks after Solve (-d=bltdump)

```
--- Alias of INI[1 | 3] ---
BLOCK 3: Generic Component (status = Solve.EXPLICIT)
----------------------------------------
### Variable:
        x[i, j]
### Equation:
        [FOR-] (16) ($RES_SIM_0)
        [----] for {i in 1:4, j in 1:4} loop
        [----]   [SCAL] (1) x[i, j] = y[j] + CAST(Real, i) * sin(CAST(Real, j) * time) ($RES_SIM_1)
        [----] end for;
         slice: {11, 7, 3, 14, 6, 2, 13, 9, 1, 12, ...}
```

Example: Diagonal Slice Model

SimCode Structures (-d=dumpSimCode)

```
INIT
========================================
  (3) single generic call [index  2] {3, 2, 1, 0}
  (2) single generic call [index  1] {15, 10, 5, 0}
  (1) single generic call [index  0] {11, 7, 3, 14, 6, 2, 13, 9, 1, 12, ...}

Algebraic Partition 1
========================================
  (6) Alias of 3
  (5) Alias of 2
  (4) Alias of 1

Generic Calls
========================================
  (0) [SNGL]: {{i | start:1, step:1, size: 4}, {j | start:1, step:1, size: 4}}
        x[i, j] = y[j] + CAST(Real, i) * sin(CAST(Real, j) * time)
  (1) [SNGL]: {{i | start:1, step:1, size: 4}, {j | start:1, step:1, size: 4}}
        y[j] = -(CAST(Real, i) * sin(CAST(Real, j) * time) - x[i, j])
  (2) [SNGL]: {{i | start:1, step:1, size: 4}}
        x[i, i] = CAST(Real, i) * cos(time)
```

Example: Diagonal Slice Model

SimCode Structures (-d=dumpSimCode)

```
INIT
========================================
  (3) single generic call [index  2] {3, 2, 1, 0}
  (2) single generic call [index  1] {15, 10, 5, 0}
  (1) single generic call [index  0] {11, 7, 3, 14, 6, 2, 13, 9, 1, 12, ...}

Algebraic Partition 1
========================================
  (6) Alias of 3
  (5) Alias of 2
  (4) Alias of 1

Generic Calls
========================================
  (0) [SNGL]: {{i | start:1, step:1, size: 4}, {j | start:1, step:1, size: 4}}
        x[i, j] = y[j] + CAST(Real, i) * sin(CAST(Real, j) * time)
  (1) [SNGL]: {{i | start:1, step:1, size: 4}, {j | start:1, step:1, size: 4}}
        y[j] = -(CAST(Real, i) * sin(CAST(Real, j) * time) - x[i, j])
  (2) [SNGL]: {{i | start:1, step:1, size: 4}}
        x[i, i] = CAST(Real, i) * cos(time)
```

Example: Diagonal Slice Model

SimCode Structures (-d=dumpSimCode)

```
INIT
========================================
  (3) single generic call [index  2] {3, 2, 1, 0}
  (2) single generic call [index  1] {15, 10, 5, 0}
  (1) single generic call [index  0] {11, 7, 3, 14, 6, 2, 13, 9, 1, 12, ...}

Algebraic Partition 1
========================================
  (6) Alias of 3
  (5) Alias of 2
  (4) Alias of 1

Generic Calls
========================================
  (0) [SNGL]: {{i | start:1, step:1, size: 4}, {j | start:1, step:1, size: 4}}
        x[i, j] = y[j] + CAST(Real, i) * sin(CAST(Real, j) * time)
  (1) [SNGL]: {{i | start:1, step:1, size: 4}, {j | start:1, step:1, size: 4}}
        y[j] = -(CAST(Real, i) * sin(CAST(Real, j) * time) - x[i, j])
  (2) [SNGL]: {{i | start:1, step:1, size: 4}}
        x[i, i] = CAST(Real, i) * cos(time)
```

Example: Diagonal Slice Model

Generated C-Code

```
void genericCall_0(DATA *data, threadData_t *threadData, int idx)
{
  int tmp = idx;
  int i_loc = tmp % 4;
  int i = 1 * i_loc + 1;
  tmp /= 4;
  int j_loc = tmp % 4;
  int j = 1 * j_loc + 1;
  tmp /= 4;
  (&data->localData[0]->realVars[0] /* x[1,1] variable */)[(i - 1) * 4 + (j-1)] = (&data->localData
     [0]->realVars[16] /* y[1] variable */)[j - 1] + (((modelica_real)i)) * (sin((((modelica_real)j)
     ) * (data->localData[0]->timeValue)));
}
```

Example: Diagonal Slice Model

Generated C-Code

```
void genericCall_1(DATA *data, threadData_t *threadData, int idx)
{
  int tmp = idx;
  int i_loc = tmp % 4;
  int i = 1 * i_loc + 1;
  tmp /= 4;
  int j_loc = tmp % 4;
  int j = 1 * j_loc + 1;
  tmp /= 4;
  (&data->localData[0]->realVars[16] /* y[1] variable */)[j - 1] = (-((((modelica_real)i)) * (sin
     ((((modelica_real)j)) * (data->localData[0]->timeValue))) - (&data->localData[0]->realVars[0]
     /* x[1,1] variable */)[(i - 1) * 4 + (j-1)]));
}
```

Example: Diagonal Slice Model

Generated C-Code

```
void genericCall_2(DATA *data, threadData_t *threadData, int idx)
{
  int tmp = idx;
  int i_loc = tmp % 4;
  int i = 1 * i_loc + 1;
  tmp /= 4;
  (&data->localData[0]->realVars[0] /* x[1,1] variable */)[(i - 1) * 4 + (i-1)] = (((modelica_real)i
     )) * (cos(data->localData[0]->timeValue));
}
```

Example: Diagonal Slice Model

Generated C-Code

```
/*
equation index: 1
type: SES_GENERIC_ASSIGN call index: 0
*/
void diagonal_slice_for1_eqFunction_1(DATA *data, threadData_t *threadData)
{
  TRACE_PUSH
  const int equationIndexes[2] = {1,1};
  const int idx_lst[12] = {11, 7, 3, 14, 6, 2, 13, 9, 1, 12, 8, 4};
  for(int i=0; i<12; i++)
    genericCall_0(data, threadData, idx_lst[i]); /*diagonal_slice_for1_genericCall*/
  TRACE_POP
}
```

Example: Entwined For-Loops Model

```
model entwine_for1
  Real x[10];
  Real y[10];
equation
  x[1] = 1;
  y[1] = 2;
  for j in 2:10 loop
    x[j] = y[j-1] * sin(time);
  end for;
  for i in 2:5 loop
    y[i] = x[i-1];
  end for;
  for i in 6:10 loop
    y[i] = x[i-1] * 2;
  end for;
end entwine_for1;
```

Expected Results

- The first two scalar equations will be solved for $x[1]$ and $y[1]$
- The three for loops will be solved as follows:
  1. alternating between the first and the second for $i = 2 : 5$
  2. alternating between the first and the third for $i = 6 : 10$

Example: Entwined For-Loops Model

```
model entwine_for1
  Real x[10];
  Real y[10];
equation
  x[1] = 1;
  y[1] = 2;
  for j in 2:10 loop
    x[j] = y[j-1] * sin(time);
  end for;
  for i in 2:5 loop
    y[i] = x[i-1];
  end for;
  for i in 6:10 loop
    y[i] = x[i-1] * 2;
  end for;
end entwine_for1;
```

Expected Results

- The first two scalar equations will be solved for $x[1]$ and $y[1]$
- The three for loops will be solved as follows:
  1. alternating between the first and the second for $i = 2 : 5$
  2. alternating between the first and the third for $i = 6 : 10$

```
BLOCK 3: Entwined Component (status = Solve.EXPLICIT)
------------------------------------------------
call order: {$RES_SIM_2, $RES_SIM_4, $RES_SIM_2, $RES_SIM_4, $RES_SIM_2, $RES_SIM_4, $RES_SIM_2,
     $RES_SIM_4, $RES_SIM_0, $RES_SIM_4, ...}
BLOCK: Generic Component (status = Solve.EXPLICIT)
------------------------------------------------
### Variable:
        y[i]
### Equation:
        [FOR-] (5) ($RES_SIM_0)
        [----] for i in 6:10 loop
        [----]   [SCAL] (1) y[i] = 2.0 * x[(-1) + i] ($RES_SIM_1)
        [----] end for;
         slice: {0, 1, 2, 3, 4}
BLOCK: Generic Component (status = Solve.EXPLICIT)
------------------------------------------------
### Variable:
        x[j]
### Equation:
        [FOR-] (9) ($RES_SIM_4)
        [----] for j in 2:10 loop
        [----]   [SCAL] (1) x[j] = y[(-1) + j] * sin(time) ($RES_SIM_5)
        [----] end for;
         slice: {0, 1, 2, 3, 4, 5, 6, 7, 8}
BLOCK: Generic Component (status = Solve.EXPLICIT)
------------------------------------------------
### Variable:
        y[i]
### Equation:
        [FOR-] (4) ($RES_SIM_2)
        [----] for i in 2:5 loop
        [----]   [SCAL] (1) y[i] = x[(-1) + i] ($RES_SIM_3)
        [----] end for;
         slice: {0, 1, 2, 3}
```

Example: Entwined For-Loops Model

SimCode Structures (-d=dumpSimCode)

```
INIT
========================================
  (6) x[1] := 1.0
  (5) y[1] := 2.0
  ### entwined call (4) ###
    (3) single generic call [index  2] {0, 1, 2, 3, 4, 5, 6, 7, 8}
    (2) single generic call [index  1] {0, 1, 2, 3}
    (1) single generic call [index  0] {0, 1, 2, 3, 4}

Algebraic Partition 1
========================================
  (12) Alias of 5
  (11) Alias of 6
  ### entwined call (10) ###
    (9) single generic call [index  1] {0, 1, 2, 3}
    (8) single generic call [index  2] {0, 1, 2, 3, 4, 5, 6, 7, 8}
    (7) single generic call [index  0] {0, 1, 2, 3, 4}

Generic Calls
========================================
  (0) [SNGL]: {{i | start:6, step:1, size: 5}}
        y[i] = 2.0 * x[(-1) + i]
  (1) [SNGL]: {{i | start:2, step:1, size: 4}}
        y[i] = x[(-1) + i]
  (2) [SNGL]: {{j | start:2, step:1, size: 9}}
        x[j] = y[(-1) + j] * sin(time)
```

Example: Entwined For-Loops Model

Generated C-Code

```
void entwine_for1_eqFunction_4(DATA *data, threadData_t *threadData)
{
  TRACE_PUSH
  const int equationIndexes[2] = {1,4};
  int call_indices[3] = {0, 0, 0};
  const int call_order[18] = {2, 1, 2, 1, 2, 1, 2, 1, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0};
  const int idx_lst_2[9] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
  const int idx_lst_1[4] = {0, 1, 2, 3};
  const int idx_lst_0[5] = {0, 1, 2, 3, 4};
  for(int i=0; i<18; i++)
  {
    switch(call_order[i])
    {
      case 2:
        genericCall_2(data, threadData, idx_lst_2[call_indices[0]]);
        call_indices[0]++;
        break;
      case 1:
        genericCall_1(data, threadData, idx_lst_1[call_indices[1]]);
        call_indices[1]++;
        break;
      case 0:
        genericCall_0(data, threadData, idx_lst_0[call_indices[2]]);
        call_indices[2]++;
        break;
      default:
        throwStreamPrint(NULL, "Call index %d at pos %d unknown for: ", call_order[i], i);
        break;
    }
  }
  TRACE_POP
```

4. Symbolic Simplification

Solving Equations for Variables

Current Implementation

- encoding expressions as a tree
- rewrite rules
- graph of equivalent expressions/equations
- heuristic graph traversal

Solving Equations for Variables

Current Implementation

- encoding expressions as a tree
- rewrite rules
- graph of equivalent expressions/equations
- heuristic graph traversal

Solving Equations for Variables

Current Implementation

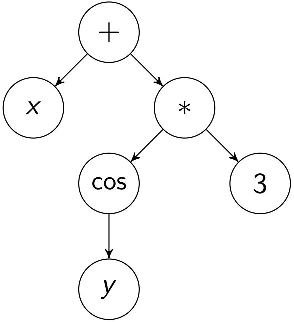
- encoding expressions as a tree
- rewrite rules
- graph of equivalent expressions/equations
- heuristic graph traversal

Solving Equations for Variables

Current Implementation

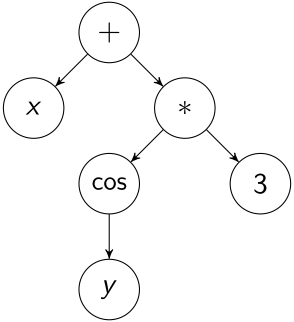
- encoding expressions as a tree
- rewrite rules
- graph of equivalent expressions/equations
- heuristic graph traversal

Expression Trees



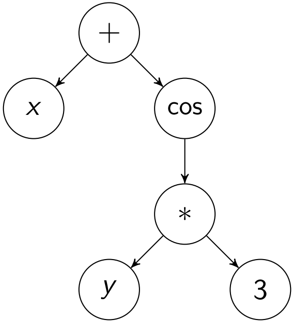
$$x + \cos y \cdot 3$$

Expression Trees



$$x + \cos(y) \cdot 3$$

Expression Trees



$$x + \cos(y \cdot 3)$$

Algebra/Rewrite Rules

$$a * b + a * c \Leftrightarrow a * (b + c)$$

$$(a + b) \cdot (a - b) \Leftrightarrow a^2 - b^2$$

$$a^m \cdot a^n \Leftrightarrow a^{m+n}$$

$$(AB)^T \Leftrightarrow B^T A^T$$

$$(M^T)^{-1} \Leftrightarrow (M^{-1})^T$$

$$z\bar{w} \Leftrightarrow \overline{(\bar{z}w)}$$

$$\cdots$$

Rewrite Rules

- Define equivalent terms
- Also possible for arrays and records

Algebra/Rewrite Rules

$$a * b + a * c \quad \Leftrightarrow \quad a * (b + c)$$

$$(a + b) \cdot (a - b) \quad \Leftrightarrow \quad a^2 - b^2$$

$$a^m \cdot a^n \quad \Leftrightarrow \quad a^{m+n}$$

$$(AB)^T \quad \Leftrightarrow \quad B^T A^T$$

$$(M^T)^{-1} \quad \Leftrightarrow \quad (M^{-1})^T$$

$$z\bar{w} \quad \Leftrightarrow \quad \overline{(\bar{z}w)}$$

$$\cdots$$

Rewrite Rules

- Define equivalent terms
- Also possible for arrays and records

Algebra/Rewrite Rules

Rewrite Rules

- Define equivalent terms
- Also possible for arrays and records

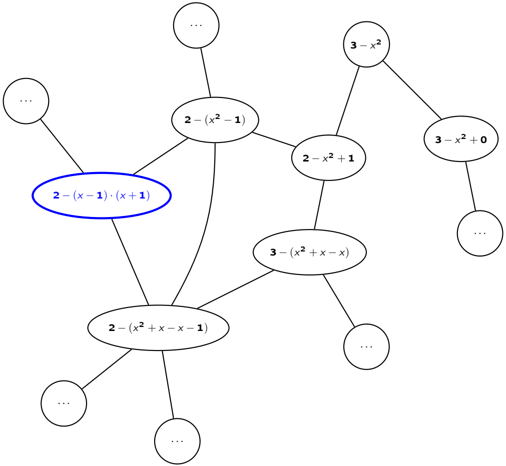
$$
\begin{aligned}
a * b + a * c &\Leftrightarrow a * (b + c) \\
(a + b) \cdot (a - b) &\Leftrightarrow a^2 - b^2 \\
a^m \cdot a^n &\Leftrightarrow a^{m+n} \\
(AB)^T &\Leftrightarrow B^T A^T \\
(M^T)^{-1} &\Leftrightarrow (M^{-1})^T \\
z\bar{w} &\Leftrightarrow \overline{(\bar{z}w)}
\end{aligned}
$$

$$\cdots$$

Equivalent Expressions

Equivalence Structure

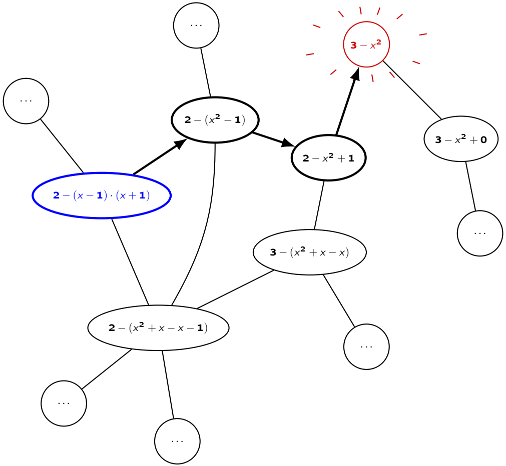
- vertex = expression
- edge = rewrite rule between e_1 and e_2
- conceptually infinite graph
- simplifying = graph search



Equivalent Expressions

Equivalence Structure

- vertex = expression
- edge = rewrite rule between e_1 and e_2
- conceptually infinite graph
- simplifying = graph search



OMC – Symbolic Simplify

Old Implementation

- destructive rewriting, loses intermediate expressions
- finds only local optima
- rewrites and rewrite order have to be carefully crafted by hand

New Implementation (WIP)

- non-destructive rewriting, potentially infinite
- finds global optima (if e-graph is saturated), cost function can be customized
- all possible rewrites are applied iteratively
- saturated e-graph reusable for next expression

OMC – Symbolic Simplify

Old Implementation

- destructive rewriting, loses intermediate expressions
- finds only local optima
- rewrites and rewrite order have to be carefully crafted by hand

New Implementation (WIP)

- non-destructive rewriting, potentially infinite
- finds global optima (if e-graph is saturated), cost function can be customized
- all possible rewrites are applied iteratively
- saturated e-graph reusable for next expression

E-Graph

E-Graphs and Equality Saturation

- E-Graph structure
- Equality Saturation
- Extraction
- Analysis

E-Graph

Informal Definition

e-graph is a set of e-classes

e-class is a set of e-nodes, has unique id

e-node is (symbol, list of e-class ids)

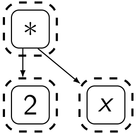
Example:

$$2x = x + x = x + x + 0 = x + x + 0 + 0 = \dots$$

E-Graph

Informal Definition

- e-graph is a set of e-classes
- e-class is a set of e-nodes, has unique id
- e-node is (symbol, list of e-class ids)



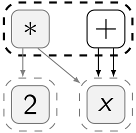
Example:

$$2x = x + x = x + x + 0 = x + x + 0 + 0 = \dots$$

E-Graph

Informal Definition

e-graph	is a set of e-classes
e-class	is a set of e-nodes, has unique id
e-node	is (symbol, list of e-class ids)



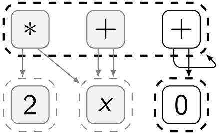
Example:

$$2x = x + x = x + x + 0 = x + x + 0 + 0 = \dots$$

E-Graph

Informal Definition

- **e-graph** is a set of e-classes
- **e-class** is a set of e-nodes, has unique id
- **e-node** is (symbol, list of e-class ids)



Example:

$$2x = x + x = x + x + 0 = x + x + 0 + 0 = \ldots$$

E-Graph

Equality Saturation

Input: An expression e

Output: *best* expression equivalent to e

$G \leftarrow$ initial e-graph from e
while *G is not saturated* **do**
    $M \leftarrow \emptyset$
    for $(l \rightarrow r) \in R$ **do**
        for *matches* (σ, c) *of l in G* **do**
            $M \leftarrow M \cup (r, \sigma, c)$
    for $(r, \sigma, c) \in M$ **do**
        $c' \leftarrow$ add $r[\sigma]$ to G and yield id
        merge c and c' in G
    rebuild G
return *best expression from G*

G is an e-graph

R is a set of rewrite rules

M is a set of matches

c, c' are e-classes

e, l, r are algebraic expressions

σ is a set of variable substitutions

E-Graph

Extraction

Get an expression out of the e-graph, according to some objective (cost function).

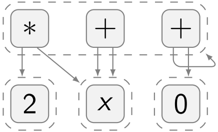
Simple cost function (e.g. minimum number of nodes): bottom-up, greedy traversal

E-Graph

Extraction

Get an expression out of the e-graph, according to some objective (cost function).

Simple cost function (e.g. minimum number of nodes): bottom-up, greedy traversal

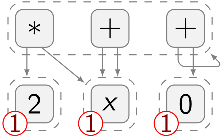


E-Graph

Extraction

Get an expression out of the e-graph, according to some objective (cost function).

Simple cost function (e.g. minimum number of nodes): bottom-up, greedy traversal

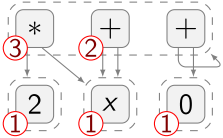


E-Graph

Extraction

Get an expression out of the e-graph, according to some objective (cost function).

Simple cost function (e.g. minimum number of nodes): bottom-up, greedy traversal

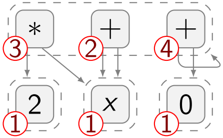


E-Graph

Extraction

Get an expression out of the e-graph, according to some objective (cost function).

Simple cost function (e.g. minimum number of nodes): bottom-up, greedy traversal



E-Graph

E-Class Analyses

Take some semilattice domain D and associate a value $d_c \in D$ to each e-class c.

`make(`n`)` $\rightarrow d_c$	construct new e-class
`join(`d_{c_1}`,` d_{c_2}`)` $\rightarrow d_c$	merge c_1, c_2 into c
`modify(`c`)` $\rightarrow c'$	optionally modify c based on d_c

Can be used to

- manipulate the e-graph, e.g. constant folding
- steer rewrites during equality saturation
- determine cost of e-nodes during extraction

E-Graph

E-Class Analyses

Take some semilattice domain D and associate a value $d_c \in D$ to each e-class c.

$\texttt{make}(n) \rightarrow d_c$	construct new e-class
$\texttt{join}(d_{c_1}, d_{c_2}) \rightarrow d_c$	merge c_1, c_2 into c
$\texttt{modify}(c) \rightarrow c'$	optionally modify c based on d_c

Can be used to

- manipulate the e-graph, e.g. constant folding
- steer rewrites during equality saturation
- determine cost of e-nodes during extraction

E-Graph

E-Class Analyses

Take some semilattice domain D and associate a value $d_c \in D$ to each e-class c.

$\texttt{make}(n) \rightarrow d_c$	construct new e-class
$\texttt{join}(d_{c_1}, d_{c_2}) \rightarrow d_c$	merge c_1, c_2 into c
$\texttt{modify}(c) \rightarrow c'$	optionally modify c based on d_c

Can be used to

- manipulate the e-graph, e.g. constant folding
- steer rewrites during equality saturation
- determine cost of e-nodes during extraction

E-Graph

Relational E-Matching

Representation

- An e-graph represents a term if any of its e-classes does.
- An e-class c represents a term if any e-node $n \in c$ does.
- An e-node $f(c_1, \ldots, c_k)$ represents a term $f(t_1, \ldots, t_k)$ if they have the same symbol and c_i represents t_i for all i.

Potential Bottleneck:

Pattern matching in the e-graph takes 60 to 90% of computation time!

Solution

Transform e-graph into data base $\rightarrow$ Conjunctive Queries are fast and can be optimized.

E-Graph

Relational E-Matching

Representation

- An e-graph represents a term if any of its e-classes does.
- An e-class c represents a term if any e-node $n \in c$ does.
- An e-node $f(c_1, \ldots, c_k)$ represents a term $f(t_1, \ldots, t_k)$ if they have the same symbol and c_i represents t_i for all i.

Potential Bottleneck:

Pattern matching in the e-graph takes 60 to 90% of computation time!

Solution

Transform e-graph into data base $\rightarrow$ Conjunctive Queries are fast and can be optimized.

E-Graph

Relational E-Matching

Representation

- An e-graph represents a term if any of its e-classes does.
- An e-class c represents a term if any e-node $n \in c$ does.
- An e-node $f(c_1, \ldots, c_k)$ represents a term $f(t_1, \ldots, t_k)$ if they have the same symbol and c_i represents t_i for all i.

Potential Bottleneck:

Pattern matching in the e-graph takes 60 to 90% of computation time!

Solution

Transform e-graph into data base $\rightarrow$ Conjunctive Queries are fast and can be optimized.

E-Graph

Relational E-Matching

Relational e-matching allows fast lookups on pre-saturated e-graphs:

1. Generate set of "training" expressions
2. Saturate an e-graph on that set
3. Store data base representation of e-graph
4. During compilation, perform queries

E-Graph

Relational E-Matching

Relational e-matching allows fast lookups on pre-saturated e-graphs:

1. Generate set of "training" expressions
2. Saturate an e-graph on that set
3. Store data base representation of e-graph
4. During compilation, perform queries

E-Graph

Relational E-Matching

Relational e-matching allows fast lookups on pre-saturated e-graphs:

1. Generate set of "training" expressions
2. Saturate an e-graph on that set
3. Store data base representation of e-graph
4. During compilation, perform queries

E-Graph

Relational E-Matching

Relational e-matching allows fast lookups on pre-saturated e-graphs:

1. Generate set of "training" expressions
2. Saturate an e-graph on that set
3. Store data base representation of e-graph
4. During compilation, perform queries

E-Graph

Relational E-Matching

Relational e-matching allows fast lookups on pre-saturated e-graphs:

1. Generate set of "training" expressions
2. Saturate an e-graph on that set
3. Store data base representation of e-graph
4. During compilation, perform queries

E-Graph

Current Status

- Experimental version in MetaModelica (Bugs included)
- Attempts to incorporate E-Graph implementation in Rust

E-Graph

Current Status

- Experimental version in MetaModelica (Bugs included)
- Attempts to incorporate E-Graph implementation in Rust

E-Graph

Next Step – Solving Equations with E-Graphs

First approach:

$$L = R \quad \Leftrightarrow \quad L - R = 0$$

BUT

Equations have a broader set of rewrite rules than expressions, i.e. equivalence transformations.

View equation as tuple of two expressions

$$L = R \quad \mapsto \quad (L, R)$$

Then e.g.

$$(L, R) \equiv (L + a, R + a)$$

Q: reusability?

E-Graph

Next Step – Solving Equations with E-Graphs

First approach:

$$L = R \qquad \Leftrightarrow \qquad L - R = 0$$

BUT

Equations have a broader set of rewrite rules than expressions, i.e. equivalence transformations.

View equation as tuple of two expressions

$$L = R \quad \mapsto \quad (L, R)$$

Then e.g.

$$(L, R) \equiv (L + a, R + a)$$

Q: reusability?

E-Graph

Next Step – Solving Equations with E-Graphs

First approach:

$$L = R \qquad \Leftrightarrow \qquad L - R = 0$$

BUT

Equations have a broader set of rewrite rules than expressions, i.e. equivalence transformations.

View equation as tuple of two expressions

$$L = R \quad \mapsto \quad (L, R)$$

Then e.g.

$$(L, R) \equiv (L + a, R + a)$$

Q: reusability?

E-Graph

Next Step – Solving Equations with E-Graphs

First approach:

$$L = R \qquad \Leftrightarrow \qquad L - R = 0$$

BUT

Equations have a broader set of rewrite rules than expressions, i.e. equivalence transformations.

View equation as tuple of two expressions

$$L = R \quad \mapsto \quad (L, R)$$

Then e.g.

$$(L, R) \equiv (L + a, R + a)$$

Q: reusability?

E-Graph

Rewrite Rule Inference Using Equality Saturation

Compared to a similar tool built on CVC4, Ruler synthesizes 5.8× smaller rulesets 25× faster without compromising on proving power. In an end-to-end case study, we show Ruler-synthesized rules which perform as well as those crafted by domain experts, and addressed a longstanding issue in a popular open source tool.

More systematic than heuristics

Instead of defining the rewrite rules by hand, let equality saturation do the job of finding the optimal rewrites.

5. Summary

Summary

Recent Development

- 2-Step Sorting
- Generalized For-Loops
- Jacobians and Sparsity Patterns

Current Development

- Generalized When, If and Array Equations
- Enable Sparse Solvers
- E-Graph based Symbolic Simplification in MetaModelica and Rust

Upcoming Plans

- Pseudo-Array Index Reduction
- E-Graph based Symbolic Solving

Summary

Recent Development

- 2-Step Sorting
- Generalized For-Loops
- Jacobians and Sparsity Patterns

Current Development

- Generalized When, If and Array Equations
- Enable Sparse Solvers
- E-Graph based Symbolic Simplification in MetaModelica and Rust

Upcoming Plans

- Pseudo-Array Index Reduction
- E-Graph based Symbolic Solving

Summary

Recent Development

- 2-Step Sorting
- Generalized For-Loops
- Jacobians and Sparsity Patterns

Current Development

- Generalized When, If and Array Equations
- Enable Sparse Solvers
- E-Graph based Symbolic Simplification in MetaModelica and Rust

Upcoming Plans

- Pseudo-Array Index Reduction
- E-Graph based Symbolic Solving

References

Chandrakana Nandi et al. “Rewrite Rule Inference Using Equality Saturation”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485496. URL: https://doi.org/10.1145/3485496.

Max Willsey et al. “egg: Fast and Extensible Equality Saturation”. In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

Yihong Zhang et al. “Relational E-Matching”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498696. URL: https://doi.org/10.1145/3498696.

Thank you for your attention!