Flattening

Jimmy Lee & Peter Stuckey

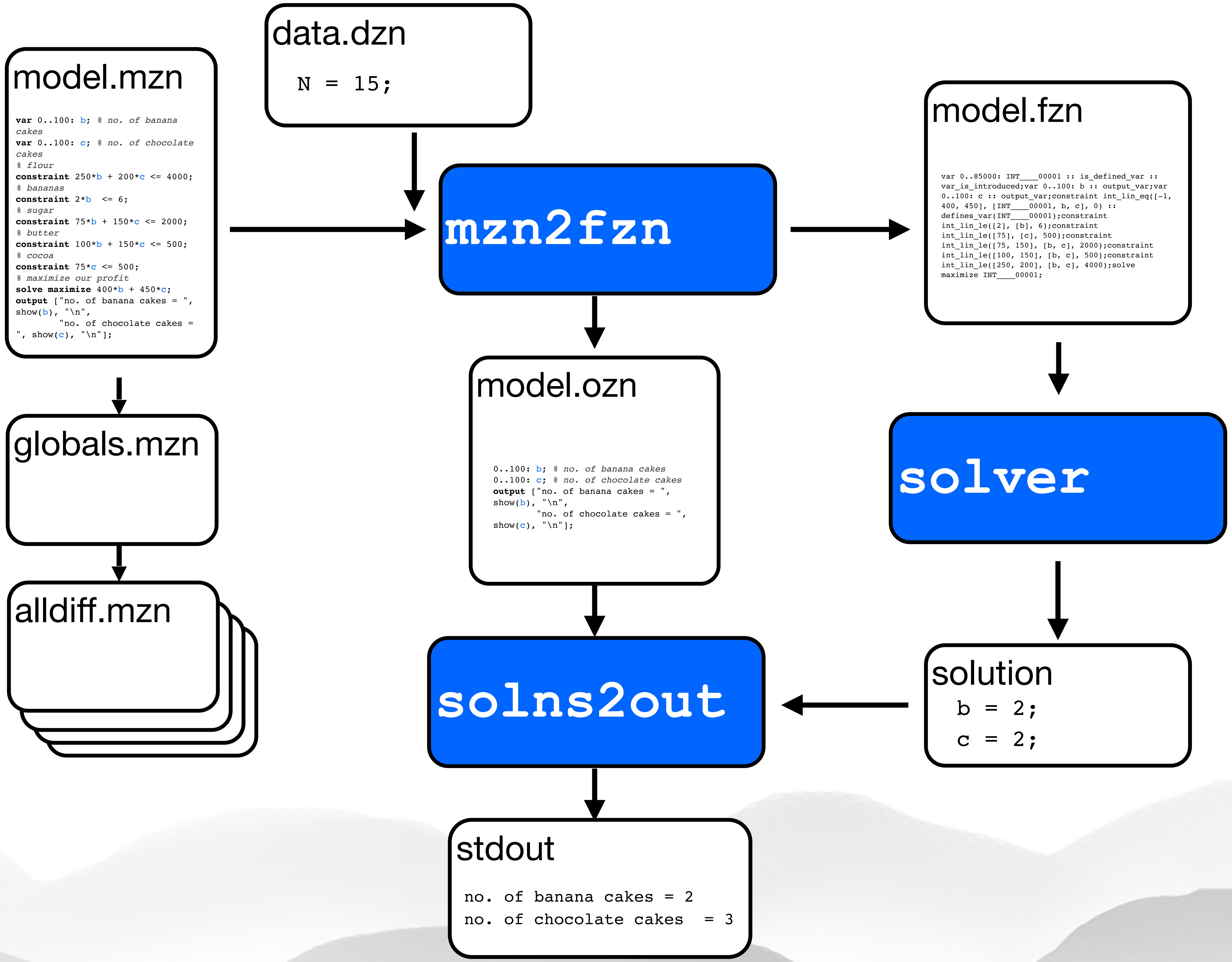
香港中文大學
The Chinese University of Hong Kong

THE UNIVERSITY OF MELBOURNE

Overview

- Flattening
  - flattening expressions
  - unrolling expressions
  - arrays
  - reification
  - predicates
  - let expressions

The MiniZinc Tool Chain



Flattening

- The process of taking a
  - model + data + globals definitions
- And creating
  - a FlatZinc model
    - variables (and parameters)
    - primitive constraints
    - solve item
    - output annotations

Flattening Expressions

- Simplifying expressions
- Evaluating fixed expressions
- Naming subexpressions (flattening)
- Bounds analysis
  - for newly introduced variables

- Common subexpression elimination

Flattening Expressions Example

- A small model

```
int: i = 3; int: j = 2;
var int: x; var 0..2: y; var 0..3: z;
x*y + y*z <= i*j;
```

- The resulting flat model is

```
var int: x;
var 0..2: y;
var 0..3: z;
var 0..6: INT01;
var int: INT02;
INT01 = y * z;
INT02 = x * y;
INT01 + INT02 <= 6;
```

subexpression bounds

subexpression name

subexpression constraint

subexpression usage

expression evaluation

Flattening Expressions Example

- A small model

```
int: i = 3; int: j = 2;
var int: x; var 0..2: y; var 0..3: z;
x*y + y*z <= i*j;
```

- The resulting FlatZinc is

```
var int: x;
var 0..2: y;
var 0..3: z;
var 0..6: INT01 :: is_defined_var;
var int: INT02 :: is_defined_var;
int_times(y,z,INT01) :: defines_var(INT01);
int_times(x,y,INT02) :: defines_var(INT02);
int_lin_le([1,1],[INT02,INT01],6);
```

subexpression linking

subexpression linking

Flattening Exercise

- Write down what you think results from

```
int: i = 3; int: j = 3;
var 0..5: x; var 0..2: y; var 0..3: z;
(x - i) * (x - j) + y + z + i + j >= 0;
```

- Did you notice the common subexpression

```
var 0..5: x; var 0..2: y; var 0..3: z;
var -3..2: INT01;
var -6..9: INT02;
INT01 = x - 3;
INT02 = INT01 * INT01;
INT02 + y + z + 6 >= 0;
```

- Don’t introduce two names for same exp

Flattening Exercise

- Write down what you think results from

```
int: i = 3; int: j = 3;
var 0..5: x; var 0..2: y; var 0..3: z;
(x - i) * (x - j) + y + z + i + j >= 0;
```

- Did you notice the common subexpression

```
var 0..5: x; var 0..2: y; var 0..3: z;
var -3..2: INT01 :: is_defined_var;
var -6..9: INT02 :: is_defined_var;
int_lin_eq([1,-1],[x,INT01],3) :: dv(INT01);
int_times(INT01,INT01,INT02) :: dv(INT02);
int_lin_le([-1,-1,-1],[z,y,INT02],6);
```

- Don't introduce two names for same exp

Common Subexpression Elimination (CSE)

- While flattening mzn2fzn checks
  - if the expression has been seen before
  - if so it uses the same name
- CSE is vital for
  - small models
  - efficient models
- But its not perfect, e.g.

```
(x - y) * (y - x) >= y - x;
```

- Leads to

```
int_lin_eq([1,-1,-1],[x,y,INT01],0);
int_lin_eq([1,-1,-1],[y,x,INT02],0);
int_times(INT01,INT02,INT03);
int_le(INT02,INT03);
```

Bounds Analysis

- Tight bounds on variables
  - help the solver
  - reduce the size of unrolling (see later)

- When introducing a variable
  - `INT01 = exp`
  - determine `l` = minimum possible value of `exp`
  - and `u` = maximum possible values of `exp`.
  - declare `var l..u: INT01;`

Bounds Analysis Example

- What bounds are determined for

```
var -2..2: x;
var 0..4: y;
constraint x * x + y * y <= 6;
```

- Resulting FlatZinc

```
var -2..2: x;
var 0..2: y;
var 0..4: INT01;
var 0..6: INT02;
constraint INT01 = x * x;
constraint INT02 = y * y;
constraint INT01 + INT02 <= 6;
```

- Could be improved (presolve is coming)

Linear Expressions

- Linear constraints are one of the most important kind of constraint

```
int: k = 4;
constraint x + 2*(y - x) + z <= k*z;
```

- Naively

```
constraint INT01 = y - x;
constraint INT02 = 2*INT01;
constraint INT03 = x + INT02;
constraint INT04 = INT03 + z;
constraint INT05 = 4 * z;
constraint INT04 <= INT05;
```

- Simplified

```
constraint int_lin_le([-1,2,-3],[x,y,z],0);
```

Unrolling

- Models are typically not fixed size
- Iterative constraints are everywhere

```
int: n;     set of int: OBJ = 1..n;
array[OBJ] of int: size;
array[OBJ] of int: value;
int: limit;
array[OBJ] of var int: x;
constraint forall(i in OBJ)(x[i] >= 0);
constraint sum(i in OBJ)(size[i]*x[i])<= limit;
solve maximize sum(i in OBJ)(value[i]*x[i]);
n = 4;
size = [5,8,9,12];
value = [3,5,7,8];
limit = 29;
```

Unrolling

- Iteration in MiniZinc is generator calls

```
sum(i in OBJ)(size[i]*x[i]) <= limit;
```

- Which are really comprehensions

```
sum([ size[i]*x[i]| i in OBJ ]) <= limit;
```

- Array comprehensions

```
constraint INT01 = 5 * x[1];
constraint INT02 = 8 * x[2];
constraint INT03 = 9 * x[3];
constraint INT04 = 12 * x[4];
array[1..4] of var int: AINT =
               [INT01,INT02,INT03,INT04];
constraint INT05 = sum(AINT);
constraint INT05 <= 29;
```

Unrolling conjunction and forall

- Top level conjunctions
  - just split into separate constraints

```
constraint forall(i in OBJ)(x[i] >= 0);
```

- Generates

```
array[1..4] of var bool: c =
 [x[1] >= 0, x[2] >= 0, x[3] >= 0, x[4] >= 0];
constraint forall(c);
```

- The result is

```
constraint x[1] >= 0;
constraint x[2] >= 0;
constraint x[3] >= 0;
constraint x[4] >= 0;
```

Flattening objectives

- Objectives in FlatZinc are single variables

```
solve maximize sum(i in OBJ)(value[i]*x[i]);
```

- Unrolls to

```
constraint INT06 = 3 * x[1];
constraint INT07 = 5 * x[2];
constraint INT08 = 7 * x[3];
constraint INT09 = 8 * x[4];
array[1..4] of var int: AINT1 =
                [INT06,INT07,INT08,INT09];
constraint INT10 = sum(AINT1);
solve maximize INT10;
```

Unrolling

- The final version of knapsack
  - after linear constraint simplification

```
array[1..4] of var int: x;
var int: INT10;
constraint x[1] >= 0;
constraint x[2] >= 0;
constraint x[3] >= 0;
constraint x[4] >= 0;
constraint 5*x[1] + 8*x[2] + 9*x[3] + 12*x[4]
<= 29;
constraint INT10 = 3*x[1] + 5*x[2] + 7*x[3] +
8*x[4];
solve maximize INT10;
```

Unrolling

- The final version of knapsack
  - after linear constraint simplification

```
array[1..4] of var int: x;
var int: INT10;
constraint int_le(0,x[1]);
constraint int_le(0,x[2]);
constraint int_le(0,x[3]);
constraint int_le(0,x[4]);
constraint int_lin_le([5,8,9,12],x,29);
constraint int_lin_eq([-1,3,5,7,8],
          [INT10,x[1],x[2],x[3],x[4]], 0);
solve maximize INT10;
```

Array Translation

- Arrays in FlatZinc
  - are one dimensional
  - start from index 1
- MiniZinc arrays need to be translated
  - modify multi-dimensional lookups to 1D
  - shift indices.
- Translation
  - `array[l1..u1, l2..u2] of int: x;`
  - expression `x[i,j]`

```
array[1..(u1-l1+1)*(u2-l2+1)] of int: x;
     x[(i - l1)*(u2-l2+1) + (j - l2 + 1)]
```

Array Translation Example

- Example 2D array

```
array[0..2,0..2] of var 0..2: x;
constraint sum(i in 0..2)(x[i,i]) <= 1;
constraint x[x[1,1],1] = 2;
```

- Flattening

```
array[0..2,0..2] of var 0..2: x;
constraint x[0,0] + x[1,1] + x[2,2] <= 1;
var int: INT01 = x[1,1];
constraint x[INT01,1] = 2;
```

- Converting to 1D

```
array[1..9] of var 0..2: x;
constraint x[1] + x[5] + x[9] <= 1;
var int: INT01 = x[5];
var int: INT02 = INT01 * 3 + (1 + 1);
constraint x[INT02] = 2;
```

Element Constraints

- The ability to lookup the entry in array using a variable index is crucial to the modelling power of MiniZinc (and other CP modelling languages)
- element constraint provides this functionality
  - `array_int_element`(index, array, result)
  - encodes array[index] = result
- For example

```
constraint x[INT02] = 2;
```

- Becomes

```
constraint array_int_element(INT02,x,2)
```

if-then-else-endif

- Flattening `if` *b* `then` *t* `else` *e* `endif`
  - evaluate *b* (assuming it is fixed)
  - if true then replace with *t*
  - else replace with *e*
- When *b* is not fixed
  - replace with [e,t][bool2int(b)+1] and flatten

```
constraint if b then x else y endif >= 0;
```

  - becomes

```
constraint [y,x][bool2int(b)+1] >= 0;
```

  - becomes

```
constraint bool2int(b,INT00);
constraint int_plus(INT00,1,INT01);
constraint array_int_element(INT01,[y,x],INT02);
constraint INT02 >= 0;
```

Flattening Boolean Expressions

- Recall that solvers only take a conjunction of constraints
  - so how do we translate e.g.

```
x > 0 -> bool2int(y > 0 /\ z > 0) + t >= u;
```

- We need to be able to "name" constraints
- Reification of a constraint c creates
  - a constraint $b \leftrightarrow c$
  - *b* is true iff *c* holds
  - *b* is false iff *c* does not hold
- FlatZinc primitives reified constraints
  - e.g. `int_lin_le`(constants, variables, lhs)

Reification Example

- Consider the expression

```
x > 0 -> bool2int(y > 0 /\ z > 0) + t >= u;
```

- Then flattening is analogous to other expressions

```
constraint BOOL01 <-> x > 0;
constraint BOOL02 <-> y > 0;
constraint BOOL03 <-> z > 0;
constraint BOOL04 <-> BOOL02 /\ BOOL03;

constraint INT01 = bool2int(BOOL04);
constraint BOOL05 <-> INT01 + t >= u;

constraint BOOL01 -> BOOL05
```

Reification Example

- Consider the expression

```
x > 0 -> bool2int(y > 0 /\ z > 0) + t >= u;
```

- Then flattening is analogous to other expressions

```
constraint int_le_reif(1,x,BOOL01);
constraint int_le_reif(1,y,BOOL02);
constraint int_le_reif(1,z,BOOL03);
constraint
array_bool_and([BOOL02,BOOL30],BOOL04);
constraint bool2int(BOOL04,INT01);
constraint int_lin_le_reif([-1,-1,1],
           [INT01,t,u],0,BOOL05);
constraint bool_le(BOOL01,BOOL05);
```

Flattening Boolean Expressions

- Avoiding negative contexts
  - push negation down to the bottom level

```
x > 0 -> bool2int(y > 0 /\ z > 0) + t >= u;
```

  - becomes

```
not x > 0 \/ bool2int(y > 0 /\ z > 0) + t >= u;
```

  - becomes

```
x <= 0 \/ bool2int(y > 0 /\ z > 0) + t >= u;
```

  - becomes

```
constraint BOOL01 <-> x <= 0;
constraint BOOL02 <-> y > 0;
constraint BOOL03 <-> z > 0;
constraint BOOL04 <-> BOOL02 /\ BOOL03;
constraint INT01 = bool2int(BOOL04);
constraint BOOL05 <-> INT01 + t >= u;
constraint BOOL01 \/ BOOL05
```

Flattening Predicates and Functions

- Predicates and functions act like macros
  - when we see an expression including them we expand it with the arguments, then flatten
- Given
  - f(x1, x2, …, xn) = exp(x1, x2, …, xn)
- Replace f(arg1, arg2, …, argn) by
  - exp(arg1, arg2, …, argn)

Flattening Predicates and Functions Example

```
predicate far_or_equal(var int:x1, var int:y1,
                       var int:x2, var int:y2)=
          man_dist(x1,y1,x2,y2) >= 4 \/
         (x1 = x2 /\ y1 = y2);

function var int: man_dist(var int:u1,
         var int:v1, var int:u2, var int:v2) =
         abs(u1 - u2) + abs(v1 - v2);

constraint far_or_equal(a,b,c,d);
```

Flattening Predicates and Functions Example

```
predicate far_or_equal(var int:x1, var int:y1,
                       var int:x2, var int:y2)=
          man_dist(x1,y1,x2,y2) >= 4 \/
         (x1 = x2 /\ y1 = y2);

function var int: man_dist(var int:u1,
         var int:v1, var int:u2, var int:v2) =
         abs(u1 - u2) + abs(v1 - v2);

constraint man_dist(a,b,c,d) >= 4 \/
           (a = c /\ b = d);
```

Flattening Predicates and Functions Example

```
predicate far_or_equal(var int:x1, var int:y1,
                       var int:x2, var int:y2)=
          man_dist(x1,y1,x2,y2) >= 4 \/
         (x1 = x2 /\ y1 = y2);

function var int: man_dist(var int:u1,
         var int:v1, var int:u2, var int:v2) =
         abs(u1 - u2) + abs(v1 - v2);

constraint abs(a - c) + abs(b - d) >= 4 \/
           (a = c /\ b = d);
```

Flattening Predicates and Functions Example

- Then flatten

```
constraint abs(a - c) + abs(b - d) >= 4 \/
           (a = c /\ b = d);
```

- becomes

```
constraint INT01 = a - c;
constraint INT02 = abs(INT01);
constraint INT03 = b - d;
constraint INT04 = abs(INT03);
constraint BOOL01 <-> INT02 + INT04 >= 4
constraint BOOL02 <-> a = c;
constraint BOOL03 <-> b = d;
constraint BOOL04 <-> BOOL02 /\ BOOL03
constraint BOOL01 \/ BOOL04;
```

Flattening Predicates with no definition

- If a global constraint *g* is native to a solver their is only a definition, not a declaration:

```
predicate
     alldifferent(array[int] of var int: a);
```

- How do we translate *g(x1, .., xn)*
- In the root context
  - leave unchanged (send to the solver)
- In a reified context?
  - try to use: *g_reif(x1, …, xn,b)*
- This might fail if it does not exist!

Flattening predicates with no definition

- Example library

```
predicate
      alldifferent(array[int] of var int: a);
predicate alldifferent_reif(
      array[int] of var int: a, var bool: b) =
 b <-> forall(i, j in index_set(a) where i < j)
             (a[i] != a[j]);
```

- Example code

```
constraint alldifferent([x,y,z]);
constraint alldifferent([y,z,t]) -> x = 0;
```

- Result

```
constraint alldifferent([x,y,z]);
constraint b <-> (y != z /\ y != t /\ z != t);
constraint b -> x = 0;
```

Flattening Let Expressions

- Let expressions allow us to introduce new variables
- FlatZinc consists only of
  - variables declarations
  - primitive constraints
- New variables must be “floated” to the top level
- Rename copies of new variables
- Complexities for relational semantics
  - partial functions,
  - local constraints

Flattening Let Expressions

- Flattening
  - exp(let { var int: x; constraint c } in exp2(x))
- rename variable to be new
  - exp(let { var int: y; constraint c } in exp2(y))
- name local constraint by new boolean
  - exp(let { var int: y; var bool: b = c; constraint b; } in exp2(y))
- float out variable declarations to top, and float constraint to nearest enclosing Boolean context

Flattening Let Expressions Example

- Consider the code

```
constraint not (8>=sum(i in 1..2)(sqrt(a[i])));
function var int:sqrt(var int: x) =
    let { var int: y;
    constraint y * y = x /\ y >= 0 } in y;
```

- Unrolling the sum gives

```
constraint not 8 >=
    (let { var int: y;
     constraint y * y = a[1] /\ y >= 0} in y) +
    (let { var int: y;
     constraint y * y = a[2] /\ y >= 0} in y);
```

Flattening Let Expressions Example

- Consider the code

```
constraint not 8 >= sum(i in 1..2)(sqrt(a[i]));
function var int:sqrt(var int: x)
    :: promise_total =
    let { var int: y;
    constraint y * y = x /\ y >= 0 } in y;
```

- Renaming the local variables gives

```
constraint not 8 >=
    (let { var int: y1;
     constraint y1*y1 = a[1] /\ y1>=0} in y1) +
    (let { var int: y2;
     constraint y2*y2 = a[2] /\ y2>=0} in y2);
```

Flattening Let Expressions Example

- Consider the code

```
constraint not 8 >= sum(i in 1..2)(sqrt(a[i]));
function var int:sqrt(var int: x) =
    let { var int: y;
    constraint y * y = x /\ y >= 0 } in y;
```

- Naming booleans gives

```
constraint not 8 >=
(let { var int: y1; constraint b1;
 var bool: b1 = y1*y1 = a[1] /\ y1>=0} in y1) +
(let { var int: y2; constraint b2;
 var bool: b2 = y2*y2 = a[2] /\ y2>=0} in y2);
```

Flattening Let Expressions Example

- Consider the code

```
constraint not 8 >= sum(i in 1..2)(sqrt(a[i]));
function var int:sqrt(var int: x) =
    let { var int: y;
    constraint y * y = x /\ y >= 0 } in y;
```

- Nearest enclosing Boolean context

```
constraint not 8 >=
(let { var int: y1; constraint b1;
 var bool: b1 = y1*y1 = a[1] /\ y1>=0} in y1) +
(let { var int: y2; constraint b2;
 var bool: b2 = y2*y2 = a[2] /\ y2>=0} in y2);
```

Flattening Let Expressions Example

- Consider the code

```
constraint not 8 >= sum(i in 1..2)(sqrt(a[i]));
function var int:sqrt(var int: x) =
    let { var int: y;
    constraint y * y = x /\ y >= 0 } in y;
```

- Float out declarations and constraints

```
var int: y1;
var bool: b1 = (y1*y1 = a[1] /\ y1>=0);
var int: y2;
var bool: b2 = (y2*y2 = a[2] /\ y2>=0);
constraint not (b1 /\ b2 /\ 8 >= y1 + y2);
```

Flattening Let Expressions Example

- Consider pushing negations

```
constraint 8 < sum(i in 1..2)(sqrt(a[i]));
function var int:sqrt(var int: x) =
    let { var int: y;
    constraint y * y = x /\ y >= 0 } in y;
```

- Float out declarations and constraints

```
var int: y1;
var bool: b1 = (y1*y1 = a[1] /\ y1>=0);
var int: y2;
var bool: b2 = (y2*y2 = a[2] /\ y2>=0);
constraint 8 < y1 + y2;
constraint b1;
constraint b2;
```

Flattening Let Expressions Example

- Consider pushing negations

```
constraint 8 < sum(i in 1..2)(sqrt(a[i]));
function var int:sqrt(var int: x) =
    let { var int: y;
    constraint y * y = x /\ y >= 0 } in y;
```

- Simplify true Booleans

```
var int: y1;

var int: y2;

constraint 8 < y1 + y2;
constraint y1*y1 = a[1] /\ y1>=0;
constraint y2*y2 = a[2] /\ y2>=0;
```

Flattening Let Expressions Example

- Consider pushing negations

```
constraint 8 < sum(i in 1..2)(sqrt(a[i]));
function var int:sqrt(var int: x) =
    let { var int: y;
    constraint y * y = x /\ y >= 0 } in y;
```

- Flatten top level conjunctions

```
var int: y1;
var int: y2;
constraint not (8 < y1 + y2);
constraint y1*y1 = a[1];
constraint y1>=0;
constraint y2*y2 = a[2];
constraint y2>=0;
```

Relational Semantics and Partial Functions

- Local variables defined by partial functions
  - need careful treatment
- The failure of the partial function must be captured in the right context (nearest)

```
var -3..3: y;
constraint (let { var int: x = 9 div y }
            in x * y != 9) -> y != 2;
```

- Translation

```
var {-3,-2,-1,1,2,3}: y1;
var int: x = 9 div y1;
var bool: b2 <-> y != 0;
constraint b2 -> y1 = y;
constraint (x * y != 9 /\ b2) -> y != 2;
```

Summary

- Understanding how MiniZinc works
  - helps in debugging models
  - helps in understanding why different modeling approaches are preferable

- Flattening
  - converts MiniZinc to a
    - conjunction of primitive constraints
  - which is what a solver can handle

Image Credits

All graphics by Cindy Cheng, Matthew Siu, Marti Wong, Xiaoan Zhu