User Defined Functions

Jimmy Lee & Peter Stuckey

香港中文大學
The Chinese University of Hong Kong

THE UNIVERSITY OF
MELBOURNE

Overview

- Functions
  - user-defined predicates provide “macro” facilities for constraints
  - user-defined functions provide “macro” facilities for expressions
- Restrictions
  - user-defined functions introduce complexities arising from partiality

Function Definition

- A function is defined in MiniZinc as

```
function <type>: <funcname>(
                    <type>: <argname>,
                    … ,<type>: <argname>)
           = <exp>
```

- Type of <exp> must be <type>
- Function use replaced by copies of body expression
- Functions must be defined

Functions with Variable Types

- Functions are most useful when they define variable expressions that can be reused
- For example: Manhattan distance

```
function var int:
    manhattan(var int:x1, var int:y1,
              var int:x2, var int:y2) =
    abs(x2 - x1) + abs(y2 - y1);
```

- We can use this to model constraints involving Manhattan distance
  - e.g. Noone adjacent to dangerous prisoners

```
forall(p in PRISONER, d in DANGER where p != d)
 (manhattan(r[p],c[p],r[d],c[d]) > 1);
```

Functions and Local Constraints

- Functions can include local variables and local constraints using let, e.g

```
function var int: myabs(var int: x) =
          let { var int: y;
                constraint int_abs(x,y) } in y;
```

- Defines an absolute value function
  - uses builtin constraint `int_abs`
  - note the use of a local constraint in the let
- Typically local constraints are required for local variables to be useful in functions
- Indeed most MiniZinc builtin functions are defined in this way

Local Constraints

- Local constraints “float” up to the nearest enclosing Boolean expression
- e.g.

```
constraint a != 0 \/ myabs(a) * b > c;
```

- Results in

```
var int: t;
constraint a != 0 \/ (int_abs(a,t) /\
                      t * b > c);
```

Local Constraints

- Let constructs allow new constraints to be introduced
  - at “any point” in the model
- Format

```
let {constraint <boolexpr> ;
            …
     constraint <boolexpr> [;]} in
<expr>
```

- Lets introduce local variables and constraints
- Local constraints “float” to the nearest enclosing Boolean context

Functions and Contexts

- Recall that
  - let constructs that introduce new variables without definition can only occur in positive contexts
- Functions with local variables
  - often do not give them a definition
- No user-defined functions in negative contexts!

Functions and Contexts

- For example

```
function var int: myabs(var int: x) =
        let { var int: y;
              constraint int_abs(x,y) } in y;
```

- Declares a new variable y
  - which is not defined (by equality)
  - BUT the constraint does define it!
- We should be able to use `myabs` in any context

Functions and Contexts

- We can annotate a function as total
  - means safe to use in non-positive contexts
  - will float to the root context
- E.g.

```
function var int: myabs(var int: x)
          :: promise_total =
          let { var int: y;
                constraint int_abs(x,y) } in y;
```

- Translating

```
constraint myabs(a) > 4 -> b < 4;
```

- Gives

```
var int: y;
constraint int_abs(a,y);
```

Recipe for Partial Functions

- What about if we want to define partial functions!
- Build a partial function with no local variables
  - that calls a total function with local variables
- For example consider
  - mydiv: implementing div for non-negative numbers
  - partial function (division by zero)

User-defined Partial Function Example

- `mydiv` for non-negative integers

```
function var int:mydiv(var int: x, var int: y)=
         assert(lb(x) >= 0 /\ lb(y) >= 0,
         “mydiv called with negative arguments”,
         let { constraint y != 0 } in
         safediv(x,y));
function var int:safediv(var int: x, var int: y)
         :: promise_total =
         let { var 0..ub(y)-1: r;
               var 0..ub(x): z;
               constraint x = y * z + r;
               constraint r < y } in
         z;
```

- Constraints in `mydiv` make `safediv` total

Overview

- Functions give us “macros” for complex expressions
- They improve common sub-expression elimination in MiniZinc
- We can include constraints in let constructs
  - most useful for defining functions
- Using `promise_total` we can make use functions with let in non positive contexts
- Splitting a user-defined function in two allows the use of partial user-defined functions in any context

EOF