THE UNIVERSITY OF MELBOURNE 香港中文大學 The Chinese University of Hong Kong

Option Types

Jimmy Lee & Peter Stuckey

香港中文大學 The Chinese University of Hong Kong THE UNIVERSITY OF MELBOURNE

Overview

- Many models involve decisions that are only meaningful if other decisions are made in a particular way
  - e.g. If I choose a configuration with rack-205e I must choose the number of internal slots between 2 and 5
- How do we represent optional decisions
  - if they are not meaningful they cannot constrain other decisions
- Minizinc uses option types
- You may have met option types in error messages!

2

THE UNIVERSITY OF MELBOURNE 香港中文大學 The Chinese University of Hong Kong

CompatibleAssignment Problem

- Given *n* workers and 2*m* tasks, arranged in two rows: 1..*m* and *m*+1..2*m*, assign workers to tasks to maximize profit, but where workers assigned to two adjacent tasks must be compatible

```
int: n;
set of int: W = 1..n;
int: m;
set of int: T = 1..2*m;
array[W,T] of int: profit;
array[W,W] of bool: compatible;

array[W] of var T: task;
alldifferent(task);
maximize sum(w in W)(profit[w,task[w]]);
```


Modeling Compatibility

- Using Implication
- If two workers are adjacent then they must be compatible

```
forall(w1, w2 in W)
      (task[w2] = task[w1] + 1 /\
       task[w1] != m ->
       compatible[w1,w2]);
```

- A large number of weak constraints

Modeling Compatibility

- Using Option Types
- We would like to invert the `task` function
  - But its not a bijection
  - The inverse is a partial function
  - We need to map each task to a worker or to no worker
- Option types add an extra value <> to a type
- Now we can invert the task function

```
array[T] of var opt W: worker;
inverse(task,worker);
```

  - Note that this is a new global constraint `inverse` that works on optional integers

5

Modeling Compatibility

- Using the inverse function we can model compatibility much more directly

```
forall(t in 1..2*m-1 where t != m)

  (compatible[worker[t],worker[t+1]]);
```

- Importantly `compatible`[*w1*,*w2*] must hold if either of *w1* = <> or *w2* = <>

6

Option Types

- Each base type has an option extended version, adding the value <>
  - opt int
  - opt float
  - opt bool
- Option type variables act like
  - a normal variable if they take a value different from <>
  - as if they were not part of the constraint if they take the value <>
  - e.g. alldifferent([3,<>,6,1,<>,<>,8,7,5]) holds

7

Option Types

- Option type variables in expressions:
  - will act like an identity if one exists in that position
    - e.g. `<> + 3 = 3, 2 - <> = 2, <> * <> = 1`
  - will propagate <> if there is no identity in that position
    - e.g. `<> - 2 = <>,  <> / 4 = <>`
- If you use option types in user-defined predicates and functions
  - you need to define the behavior

8

Hidden Option Types

- You are already using option types
- Where are option types hidden
- How can I avoid them

Hidden Option Types

- Implicit uses:
- Iteration over variable sets

```
var set of 1..n: x;
sum(i in x)(size[i]) <= cap;
```

- Is syntactic sugar for

```
sum(i in 1..n)
   (if i in x then size[i] else <> endif)
<= cap;
```

- The <> acts like 0 in a sum since its +

Hidden Option Types

- Implicit uses:
- Variable where clause

```
var set of 1..n: x;
sum(i in 1..n where i in x)(size[i])
<= cap;
```

- Is also syntactic sugar for

```
sum(i in 1..n)
   (if i in x then size[i] else <> endif)
<= cap;
```

11

Avoiding Option Types

- Avoid iterating over variable sets
- Replacement translations
  - sum

```
sum(i in x)(size[i]) <= cap;
sum(i in ub(x))
   (bool2int(i in x)*size[i]) <= cap;
```

  - forall

```
forall(i in x)(size[i] <= cap);
forall(i in ub(x))
      (i in x -> size[i] <= cap);
```

  - exists

```
exists(i in x)(size[i] <= cap);
exists(i in ub(x))(i in x /\ size[i]<=cap);
```

12

Avoiding Option Types

- Avoid using variable where clauses
- Replacement translations
  - sum

```
sum(i in S where i >= x)(size[i]) <= cap;
sum(i in S)(bool2int(i >= x)*size[i]) <= cap;
```

  - forall

```
forall(i in S where i >= x)(size[i] <= cap);
forall(i in S)
      (i >= x -> size[i] <= cap);
```

  - exists

```
exists(i in S where i >= x)(size[i] <= cap);
exists(i in S)
      (i >= x /\ size[i] <= cap);
```

13

More complicated cases

- We can do the same for max and min
  - but its more complex

```
m = min(i in x)(size[i]);
```

- Can be replaced by

```
int: larger = 1 + max(i in ub(x))(ub(size[i]));
m = min(i in ub(x))
       (if i in x then size[i]
        else larger endif);
```

- Note that this returns `larger` if x is empty, whereas the original expression fails
  - normally we don't expect this to fail

14

More complicated cases

- Consider the following example

```
var set of 1..n:x
array[1..n] of var 1..m:y;
alldifferent([ y[i] | i in x ]);
```

- First `alldifferent` on option types
  - is not likely to be native
- How can we avoid hidden option types
  - consider a constraint that encodes the result

```
var set of 1..n:x
array[1..n] of var 1..m:y;
alldifferent_except_0([ bool2int(i in x)*y[i]
                      | i in ub(x) ]);
```

15

Overview

- Many problems involve decisions that
  - only make sense if other decisions are first made
- Option types provide a concise way to express this
- Whenever you iterate over a variable set
  - including with a variable where clause
  - option types are used
- Normally they should silently perform as intended, But you may prefer to replace these iterations to avoid option types

16