Métodos Formais
2022.2

Introduction to Alloy: Constraints

Áreas de Teoria e de Linguagens de Programação DCC/UFMG

Alloy Constraints

- Signatures and fields resp. define classes (of atoms) and relations between them
- Alloy models can be refined further by adding *formulas* expressing additional constraints over those classes and relations
- Several operators are available to express both logical and relational constraints

Logical operators

The usual logical operators are available, often in two forms

– not	!	(Boolean) negation
– and	&&	conjunction
– or	||	disjunction
– implies	=>	implication
– else		alternative
–	<=>	equivalence

Quantifiers

Alloy includes a rich collection of quantifiers

```
all x: S | F              F holds for every x in S
some x: S | F             F holds for some x in S
no x: S | F               F holds for no x in S
lone x: S | F             F holds for at most one x in S
one x: S | F              F holds for exactly one x in S
```

Predefined sets in Alloy

- There are three predefined set constants:
  - none : empty set
  - univ : universal set
  - ident : identity relation

- Example. For a model instance with just:

  Man = {(M0),(M1),(M2)}
  Woman = {(W0),(W1)}

  the constants have the values

  none = {}
  univ = {(M0),(M1),(M2),(W0),(W1)}
  ident ={(M0,M0),(M1,M1),(M2,M2),(W0,W0),(W1,W1)}

Everything is a Set in Alloy

- There are *no scalars*
  - We never speak directly about elements (or tuples) of relations
  - Instead, we can use *singleton* relations:

```
one sig Matt extends Person
```

- Quantified variables *always* denote singleton relations:

```
all x : S | ... x ...
```

`x = {t}` for some element t of S

Set operators

$+$	union
&	intersection
$-$	difference
in	subset
$=$	equality
!=	disequality

- Example. Married men:

```
Married & Man
```

Relational operators

```
->      arrow (cross product)
~       transpose
.       dot join
[]      box join
^       transitive closure
*       reflexive-transitive closure
<:      domain restriction
:>      image restriction
++      override
```

Relational composition (Join)

p . q

- p and q are two relations that are *not both unary*
- p.q is the relation you get by taking every combination of a tuple from p and a tuple from q and adding their join, if it exists

How to join tuples?

- What is the join of theses two tuples ?

```
( a1 , . . . , am)
( b1 , . . . , bn )
```

- If $\mathrm{am} \neq \mathrm{b1}$, then join is undefined
- If $\mathrm{am} = \mathrm{b1}$, then it is

```
( a1 , . . . , am−1,b2 , . . . , bn )
```

- Examples.

```
( a , b ) . ( a , c , d )              undefined
( a , b ) . ( b , c , d )     =        ( a , c , d )
```

- What about (a).(a)?

How to join tuples?

- What is the join of theses two tuples ?

  (a1 , . . . , am)
  (b1 , . . . , bn)

- If $\mathrm{am} \neq \mathrm{b1}$, then join is undefined
- If $\mathrm{am} = \mathrm{b1}$, then it is

  (a1 , . . . , am−1 , b2 , . . . , bn)

- Examples.

(a , b) . (a , c , d)		undefined
(a , b) . (b , c , d)	=	(a , c , d)

- What about (a).(a)? Not defined!
  - t1.t2 is not defined if t1 and t2 are *both* unary tuples

Example: family structure

```
abstract sig Person {
  children: set Person,
  siblings: set Person
}
sig Man, Woman extends Person {}
one sig Matt in Man {}
sig Married in Person {
  spouse: one Married
}
```

How would you use join to find Matt's children or grandchildren ?

Example: family structure

```
abstract sig Person {
  children: set Person,
  siblings: set Person
}
sig Man, Woman extends Person {}
one sig Matt in Man {}
sig Married in Person {
  spouse: one Married
}
```

How would you use join to find Matt's children or grandchildren ?

```
Matt.children            -- Matt's children
Matt.children.children   -- Matt's grandchildren
```

What if we want to find Matt's descendants?

Example: family structure

How would you model the *constraint*:

Every married man (woman) has a wife (husband)

Example: family structure

How would you model the *constraint*:

Every married man (woman) has a wife (husband)

```
all p: Married |
 (p in Man => p.spouse in Woman)
 and
 (p in Woman => p.spouse in Man)
```

A spouse can't be a sibling

Example: family structure

How would you model the *constraint*:

Every married man (woman) has a wife (husband)

```
all p: Married |
  (p in Man => p.spouse in Woman)
  and
  (p in Woman => p.spouse in Man)
```

A spouse can't be a sibling

```
no p: Married |
  p.spouse in p.siblings
```

Acknowledgments

These notes are heavily based on notes from Matt Dwyer, John Hatcliff, Rod Howell, Laurence Pilard and Cesare Tinelli.