CIS 771: Software Specifications

Lecture: Alloy Logic
(part D)

Outline

- Logical operators
- Quantification
- Cardinality specifications
- Let expressions/constraints
- Comprehension notation

Logical Operators

There are two forms of each logical operator: a shorthand and a verbose form

Verbose	*Shorthand*	*Operator Name*
`not`	`!`	negation
`and`	`&&`	conjunction
`or`	`||`	disjunction
`implies`	`=>`	implication
`else`	`,`	alternative
`iff`	`<=>`	bi-implication

Logical Operators

These operators are standard except for `else`...

The *`else`* operator is used with the implication operator:

```
F implies G else H
```

is equivalent to

```
(F and G) or ((not F) and H)
```

...G holds when F holds *...H holds when F does not hold*

Logical Operators

Implications are often nested…

```
C1 => F1,
C2 => F2,
C3 => F3
```

or equivalently

```
C1 implies F1
else C2 implies F2
else C3 implies F3
```

…under condition C1, F1 holds, and if not, then under condition C2, F2 holds, and if not, under condition C3, F3 holds

Logical Operators

Shorthand for conjunction of constraints…

`{F G H}` …is equivalent to… `F and G and H`

The negation symbol can be combined with comparison operators…

`a != b` …is equivalent to… `not (a = b)`
…is equivalent to… `a not= b`

Quantification

A quantified constraint has the form…

```
Q x: e | F
```

The forms of quantification in Alloy are…

all *x: e | F*	*F* holds for every *x* in *e*;
some *x: e | F*	*F* holds for some *x* in *e*;
no *x: e | F*	*F* holds for no *x* in *e*;
lone *x: e | F*	*F* holds for at most one *x* in *e*;
one *x: e | F*	*F* holds for exactly one *x* in *e*.

Quantification

Several variables can be bound in the same quantifier…

```
one x: e, y: e | F
```

… says that there is exactly one combination of values for *x* and *y* that makes *F* true.

Variables with the same bound can share a declaration…

```
one x, y: e | F
```

Restrict the bindings only to include ones in which the bound variables are disjoint from one another …

```
all disj x, y: e | F
```

… means that *F* is true for any distinct combination of values for x and y.

Quantification Examples

```
some n: Name, a: Address | a in n.address
```

…says that some name maps to some address (that is, the address book is not empty);

```
no n: Name | n in n.^address
```

…says that no name can be reached by lookups from itself (that is there are no cycles in the address book);

```
all n: Name | lone d: Address | d in n.address
```

…says that every name maps to at most one address;

```
all n: Name | no disj d, d': Address |
              d + d' in n.address
```

…says the same thing, but slightly differently: that for every name, there is no pair of distinct addresses that are amongst the results obtained by looking up the name.

Expressing Cardinality

Several quantifier forms can be used to state cardinality constraints on set-valued expressions…

some *e*	*e* has some tuples;
no *e*	*e* has no tuples;
lone *e*	*e* has at most one tuple;
one *e*	*e* has exactly one tuple.

Expressing Cardinality

Examples

`some Name`

…says that the set of names is not empty;

`some address`

…says that the address book (relation) is not empty: there is some pair mapping a name to an address;

`no (address.Addr - Name)`

…says that nothing is mapped to addresses except for names;

`all n: Name | lone n.address`

…says that every name maps to at most one address (more succinctly than in the previous example -- two slides ago);

`all n: Name | one n.address or no n.address`

…says the same thing.

For You To Do

- By hand, construct an example (find an `x, X, y, Y` and expression *P* that refers to `x` and `y`) that shows that the two constraints…
  - `all x : X | some y : Y |` *P*`(x,y)`
  - `some y : Y | all x : X |` *P*`(x,y)`

  …are not equivalent (explain your answer)
- Write a simple Alloy model and query (command) that, when executed, will generate an instance showing that the two constraints above are not equivalent.
- Does the second constraint imply the first for an arbitrary expression *P*? Demonstrate how you might use Alloy to provide evidence for this claim. Can you use Alloy to prove that the second constraint implies the first?
- Consider the following two statements:
  - `(all x : X |` *P*`(x)) <=> (not some x : X | not` *P*`(x))`
  - `(some x : X |` *P*`(x)) <=> (not all x : X | not` *P*`(x))`

  …which one of the constraints above is true? If the constraint is not true, does the implication hold in just one direction? Justify your answer with an explanation or a demonstration using Alloy.
- Consider the following two statements:
  - `(all x : X |` *P*`(x) and Q(x)) <=> (all x : X |` *P*`(x)) and (all x: X |` *Q*`(x))`
  - `(some x : X |` *P*`(x) and Q(x)) <=> (some x : X |` *P*`(x)) and (some x: X |` *Q*`(x))`
  - …which one of the constraints above is true? If the constraint is not true, does the implication hold in just one direction? Justify your answer with an explanation or a demonstration using Alloy.

Let Expressions/Constraints

When an expression appears repeatedly, or is a subexpression of a larger, complicated expression, you can factor it out via a let expression.

```
let x = e | A
```

is short for *A* with each occurrence of the variable *x* replaced by the expression *e*. The body of the let, *A*, and thus the form as a whole, can be a constraint or an expression.

Example

```
let addr = b.address | addr[n1] + addr[n2]
```


Let Expressions/Constraints

Examples

The preferred address for an alias a is the work address if it exists, otherwise the home address

```
all a: Alias |
  let w = a.workAddress |
     a.address = if some w then w else a.homeAddress
```

or

```
all a: Alias |
  a.address =
    let w = a.workAddress |
      if some w then w else a.homeAddress
```

Comprehensions

Comprehensions form relations from properties -- they specify what property a tuple must possess for it to belong to a relation.

$\{x_1\text{: } e_1,\ x_2\text{: } e_2,\ \ldots,\ x_n\text{: } e_n \mid F\}$

Example

```
{ n: Name | no n.^address & Addr }
```

…is the set of names that don't resolve to any actual addresses;

```
{ n: Name, a: Addr | n->a in ^address }
```

…is a relation mapping names to addresses that corresponds to the multi-level lookup.

Acknowledgements

- The material in this lecture is based on Section 3.5 from…
    - *Software Abstractions: Logic, Language, and Analysis,* Daniel Jackson, MIT Press, 2006.