Alloy Primer

Sam Coy

April 3, 2017

Introduction

Alloy is a language and tool for relational modelling, and can be found at `http://alloy.mit.edu`. This document aims to give a brief and accessible overview of Alloy's semantics.

Relations

A *relation* is a subset of the Cartesian product of a series of sets. For example, given the sets $A = \{1, 2, 3\}$ and $B = \{4, 5\}$: $\emptyset$, $\{(1, 4), (3, 5)\}$, and $\{(1, 4), (2, 5), (3, 4), (3, 5)\}$ are all relations between A and B. The individual elements of a relation, such as $(1, 4)$, are called *tuples*. In Alloy's syntax, a relation between two sets is notated using an arrow (->), for example a relation between A and B is denoted A->B. For a single set, a relation is any subset of that set (see below). For more than two sets, the notation just chains, giving A->B->C for three sets, and so on.

Alloy represents everything as a relation of some order, which allows for a consistent syntactic approach when dealing with any structure. As a stylistic point, for this guide, types will start with capital letters, and instances of types will start with a lower case letter. Here is the syntax for relation declarations in Alloy.:

```
boss: Employee -> Employee
empDesk: Employee -> Office -> Desk
empDesk1 = e1 -> o1 -> d1 // specific tuples of a type can be
                          // put in a relation as well, using =
```

In the above examples, *boss* and *empDesk* are relations between the specified sets, however the tuples in the relation not specified (it could be **any** valid set of tuples between those two sets). *empDesk1* is a specific relation (of the same type as *empDesk*), containing one tuple, $(e1, o1, d1)$ in mathematical notation, or e1->o1->d1 in Alloy notation. We can specify relations with more than one tuple, using a + to concatenate. For example: rel =a1->b1 + a2->b2. Note that **=**, not **:**, is used to define a relation with specific tuples as opposed to types.

As unary relations, sets are notated differently - by default instead of a relation of arbitrary size, e:Employee is of size one. The cardinality can be specified as follows:

```
e: set Employee // any number of Employees
e: one Employee // one Employee (note that "one" can be omitted)
e: some Employee // one or more Employees
e: lone Employee // one or zero Employees
e: Employee // one Employee
```

In each of the above examples, the variable *e* represents a certain set of Employee objects.

Also note that similar cardinality restrictions can be applied to relation declaration as well, using A m->n B syntax, where A and B are types and m and n are cardinality specifiers. If no specifier is given, it defaults to "set". For example:

```
boss: Employee -> lone Employee // each Employee on the left is mapped
                                // to one or zero Employees on the right
mainOffice: Office one -> Desk // only one Office appears on the left
                           // it is mapped to each Desk in a set of Desks
```

Dot Joins

A dot join is a binary operation on two relations in Alloy. Given two relations of type R1:A->...->B->C and R2:C->D->...->E (in that order), a dot join R1.R2 returns a new relation RDOT:A->...->B->D->...->E, where a tuple a->...->b->d->...->e is in RDOT if r1=a->...->b->c1 is in R1, r2=c2->d->...->e is in R2, and c1=c2. It's important to note that if either R1 or R2 is a unary relation (a set), this is fine; if R2:C for example, then RDOT: A->B, where a tuple a->b is in RDOT provided the c's are equal for r1 and r2. For example:

```
rel1 = n1->a1 + n2->a2 + n1->a2
rel2 = a2
rel3 = a1->a3

// Using the above relations:
rel1.rel2 = n2 + n1 // from the second and third tuples of rel1
rel1.rel3 = n1->a3 // from the first tuple of rel1
rel2.rel3 = none // alloy's notation for the empty set
                 // no tuple in rel3 begins with a2
```

It is possible in Alloy to "trim" a relation using dot joins. For example, in the *empDesk* relation from earlier (empDesk: Employee->Office->Desk), we can get the relation mapping an Employee to their office by dot joining *empDesk* with Desk. This is because the Desk type is actually a set (unary relation) of all instances of Desk.

From the definition of the dot join earlier, this results in a relation containing all Employee->Office relations, since the rightmost element for all relations in *empDesk* is **guaranteed** to match an element in Desk, because that is simply all Desks.

Logic and Quantifiers

Logic

Alloy has a standard set of logical operators.

- **not/!**
- **and/&&**
- **or/||**
- **implies/=>**
- **iff/<=>**

Both forms (text and symbol) can be used interchangeably.

Set Operations and Cardinality

Alloy supports a range of operations on sets. The binary set operations are as follows and only work if the two sets (or relations) are of the same type. The top three all output a relation, the last two output a boolean value.

- **+** (Union)
- **-** (Difference)
- **&** (Intersection)
- **in** (Subset)
- **=** (Equality)

The cardinality of a set (or relation) can be found using the **#** operator, and can then be compared to integer literals or the cardinality of other sets. As an aside, any such comparison uses standard mathematical notation, except for $\leq$, where Alloy uses **=<**. An example:

```
rel = n1->a0 + n0->a2 + n0->a0 + n2->a1
rel2 = a0 + a2
rel3 = a0 + a1

#rel = 4 // true
#rel > 5 // false
#rel =< 3 // false
#(rel2 + rel3) = 3 // true
#(rel2 & rel3) = 1 // true
```

Quantifiers

Alloy supports four quanitifers, examples of their use are given below:

```
no e:Employee | <pred> // no Employees satisfy <pred>
lone e:Employee | <pred> // at most one Employee satisfies <pred>
one e:Emplyee | <pred> // exactly one Employee satisfies <pred>
some e:Employee | <pred> // at least one Employee satisfies <pred>
all e:Employee | <pred> // all Employees satisfy <pred>
```

The predicate can be anything that has a boolean value, for example a cardinality check, a set operation that returns a boolean value, or a **pred** (see below).