Relations and relational operators

Atoms

Atoms are Alloy's primitive entities. An atom is:

- *indivisible:*
  - it can't be broken down into smaller parts
- *immutable:*
  - its properties don't change over time
- *uninterpreted:*
  - it doesn't have any built-in properties, the way that numbers or strings do, for example

…if you want to expose some properties of atoms, you introduce relations to capture these properties as additional structure.

Very few things in the world are truly atomic. An Alloy atom is a modeling abstraction. It represents an entity whose details are irrelevant or that we simply don't want to expose in our modeling and reasoning.

Relations

A ***relation*** is a structure that relates atoms

- A relation (table)…
  - consists of a set of ***tuples*** (rows)
    - the ***size*** is the number of tuples
    - any size is possible including 0
    - order of rows doesn't matter
  - each tuple is a sequence of atoms
    - all tuples must have the same length (***arity***)
    - order of atoms *does* matter

Set view

atoms

```
rel = {B0 -> N0 -> A0,
       B0 -> N1 -> A1,
       B1 -> N2 -> A2,
       B1 -> N3 -> A2}
```

Table view

B0	**N0**	**A0**
B0	**N1**	**A1**
B1	**N1**	**A2**
B1	**N2**	**A2**

size = 4

arity = 3

Relations

Terminology

- Unary relation (sets)
  - arity = 1 (1 column)
- Binary relation
  - arity = 2 (2 columns)
- Ternary relation
  - arity = 3 (3 columns)
- Multirelation
  - arity > 3
- Scalar (single values)
  - unary relation with only one tuple

Examples

Aircraft = {Aircraft0, Aircraft1, Aircraft2}
Int = {-8, -7, ..., 6, 7}

capacity = {Aircraft0 –> 6, (Aircraft1 –> 5}

See examples later.

We won't use these.

myName = {Savi}
yourAircraft = {Aircraft0}

Every value in Alloy's logic is really a relation!

Relations and Fields

- A field in a signature is really a relation from atoms of that signature to atoms of the type indicated in the field.
- Consider the Aircraft signature:

```
sig  Aircraft
{
    capacity : Int
    onboard: set Person
}
```

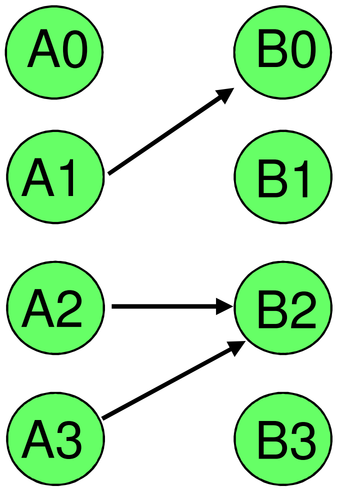
- capacity is a relation, mapping each Aircraft to a single Int
- onboard is also a relation, mapping each Aircraft to one or more Persons.

Functions and Injective relations

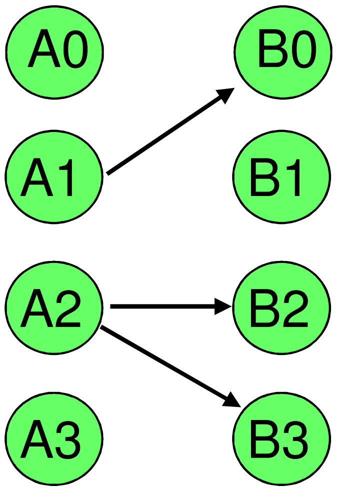
Consider binary relations from a set A to a set B.

- A binary relation that maps each A to at most one B is called a ***function***.
- A binary relation that maps at most one A to each B is ***injective***.

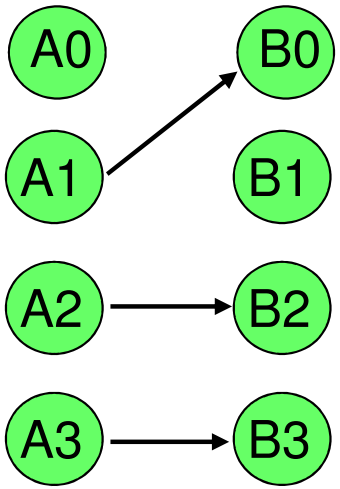
Examples



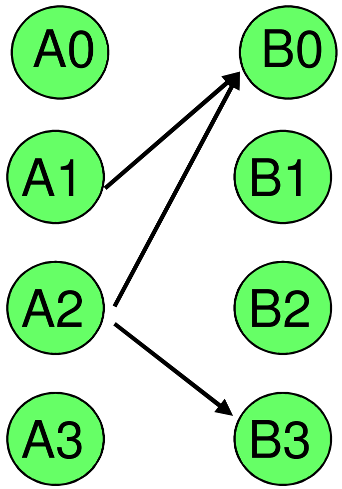
function,
but not injective



injective,
but not a function



function,
and injective



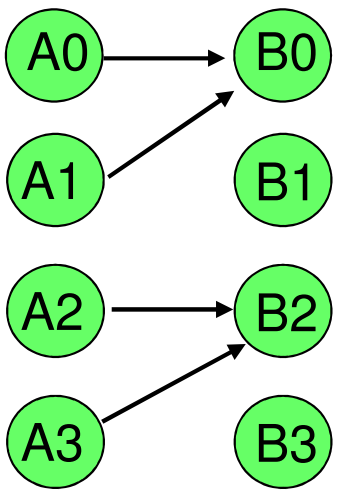
neither a function,
nor injective

Note: an empty relation is trivially functional and injective

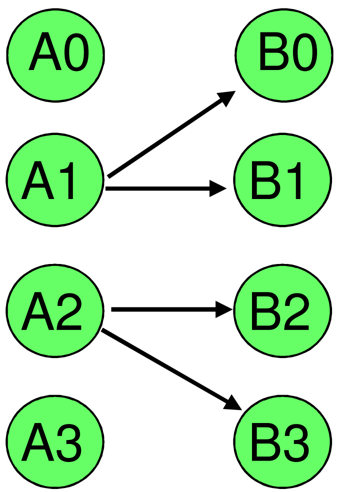
Total and surjective relations

- A binary relation that maps each A to at least one B is said to be ***total***.
- A binary relation that maps at least one A to each B is ***surjective*** or ***onto***.

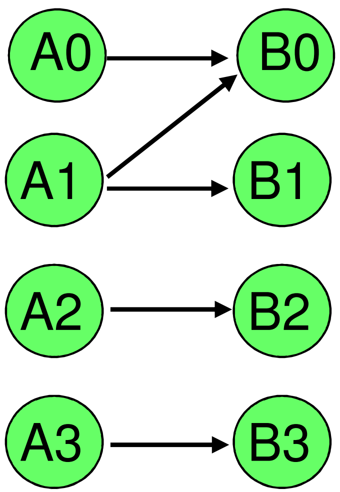
Examples



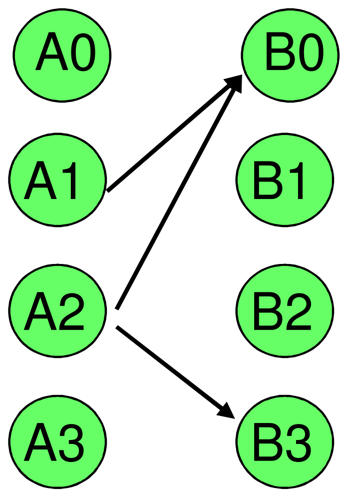
total, but not surjective



surjective, but not total



total and surjective



neither total nor surjective

Multiplicities in relations

Recall that a field in a signature is really a relation. If that field has a multiplicity constraint, this determines what kind of relation it is:

```
sig A {
    r1 : B           // r1 maps each A to exactly one B (total and functional - default)
    r2 : one B       // r2 maps each A to exactly one B (total and functional )
    r3 : lone B      // r3 maps each A to at most one B (functional)
    r4 : some B      // r4 maps each A to at least one B (total)
    r5 : set B       // r5 maps each A to any number of B (no constraints)
}
```

Each of these fields is really a binary relation from A to B.

Notice that the multiplicity **one** can be omitted as it is the default.

Multiplicities in relations

A field in a signature may itself be a relation. In this case, the field represents a *ternary* relation (a set of 3-tuples).

In the example below, each field is a relation containing tuples of the form A **->** B **->** C.

Think of this as a relation mapping each A to a *relation* from B to C. The multiplicities constrain the relation from B to C:

```
sig A {
    r1 : B -> C             // no multiplicity constraints
    r2 : B -> some C        // each B maps to at least one C (total)
    r3 : B -> lone C        // each B maps to at most one C (functional)
    r4 : B -> one C         // each B maps to exactly one C (total and functional)
    r5 : B some -> C        // at least one B maps to every C (surjective)
    r6 : B lone -> C        // at most one B maps to each C (injective)
    r7 : B one -> C         // exactly one B maps to each C (surjective and injective)
}
```

Multiplicities in relations

Left and right multiplicity constraints may be combined, e.g.

```
rel : A one −> one B   // surjective, injective, total and functional
                       // (aka a “one-to- one correspondence”)
```

Syntax; Pause for Thought

- The tuple A –> B can also be written as (A, B).

- Consider these relations.

  capacity = { Aircraft0 –> 7, Aircraft1 –> 6, Aircraft2 –> 6 }

  onboard = { Aircraft0 –> Person1, Aircraft0 –> Person2,
  Aircraft1 –> Person3 }

- Consider the Aircraft specification. Could the relations above be possible models of the specification? What about these:

  capacity = { Aircraft0 –> 1, Aircraft1 –> 6, Aircraft2 –> 6 }

  onboard = { Aircraft0 –> Person1, Aircraft0 –> Person2 }

Domain and Range

- The *domain* of a relation is the set of atoms in its first column
- The *range* of a relation is the set of atoms in its last column

Example

onboard = {Aircraft0 -> Person0, Aircraft0 -> Person1,
Aircraft1 -> Person2}

The domain of onboard is {Aircraft0, Aircraft1}

The range of onboard is {Person0, Person1, Person2}

The identity relation

- **iden** - identity relation (relates each element to itself)

Example

In a model with the following sets Aircraft and Person…

Aircraft = {Aircraft0, Aircraft1, Aircraft2}
Person = {Person0, Person1}

…**iden** has the following value…

iden = {Aircraft0 **->** Aircraft0, Aircraft1 **->** Aircraft1,
Aircraft2 **->** Aircraft2, Person0 **->** Person0,
Person1 **->** Person1}

Relational Operations

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

Product

Definition

- The ***product*** p -> q of two relations p and q is the relation consisting of all possible combinations of tuples from p and q.

Example

For
Aircraft = { Aircraft0, Aircraft1}
Person = { Person0, Person1 }

we have
Aircraft -> Person = {Aircraft0 -> Person0, Aircraft0 -> Person1,
Aircraft1 -> Person0, Aircraft1 -> Person1}

…the relation mapping all aircraft to all persons

Dot Join - Relation Composition

Intuition:

- p.q contains concatenations of tuples from p and q where the values of the last column of p and first column of q agree.

Example

onboard

maps an aircraft to the persons onboard

{ A0 -> P0,
A0 -> P2,
A1 -> P2,
A2 -> P3 }

telephone

maps persons to telephone numbers

{P0 -> T0,
P0 -> T1,
P1 -> T1,
P1 -> T2,
P2 -> T3,
P4 -> T3 }

onboard.telephone

maps an aircraft to telephone numbers of the passengers on board

{A0 -> T0,
A0 -> T1,
A0 -> T3,
A1 -> T3}

Dot Join

One of the most common uses of the dot join operator is navigation via “fields” of signatures.

Aircraft = {Aircraft0, Aircraft1}
Person = {Person0, Person1, Person2}

onboard = {Aircraft0 -> Person0,
Aircraft0 -> Person1,
Aircraft1 -> Person2}

Navigating (forward)

Aircraft.onboard = {Person0, Person1, Person2}
the set of persons on board any aircraft;
Aircraft0.onboard = {Person0, Person1}
the set of persons on board Aircraft0

Dot Join

Navigation can also proceed in the reverse direction…

sig Person {alias: **set** Person}

Person = {Person0, Person1, Person2, Person3}

alias = {Person0 –> Person1, Person0 –> Person2}

Navigating (forward and backward)

Person.alias = {Person1, Person2}

Forward direction: the set of persons who are someone's alias

alias.Person = {Person0}

Backward direction: the set of persons who have an alias

Dot Join

Dot join can be used to compose multiple relations…

Aircraft = {Aircraft0, Aircraft1}
Person = {Person0, Person1, Person2, Person3}

onboard = {Aircraft0 -> Person0, Aircraft0 -> Person1, Aircraft1 -> Person2}

alias = {Person0 -> Person1, Person0 -> Person2}

Aircraft.onboard.alias = {Person1, Person2}

Alloy first calculates Aircraft.onboard, then joins the result with alias

Box Join

The box operator [ ] is semantically identical to dot join…

q[p]

…has the same meaning as…

p.q

The box operator [ ] has a lower precedence than the dot operator, i.e., the dot operator binds more tightly…

a.b.c[d]

…is short for…

d.(a.b.c)

Box Join - Rationale

The box join notation [ ] is not strictly necessary, but is often more readable than the equivalent construct written using dot join…

Given a relation addr associating names and addresses, the expression

addr[n]

denotes the set of addresses associated with name n, and is equivalent to

n.addr

Box join is very handy when working with ternary relations.

Transpose (Inverse)

Definition

- The *transpose* (or *inverse*) ~r of a binary relation r is the relation formed by reversing the order of atoms in each tuple in r.

Example:

Given a relation representing an family that maps persons to their children…

```
hasChild = {Person0 -> Person1,
            Person0 -> Person2,
            Person2 -> Person3}
```

…its transpose is the relation that maps persons to their parents…

```
~hasChild = {Person1 -> Person0,
             Person2 -> Person0,
             Person3 -> Person2}
```

Transitive Closure

Definitions

- A binary relation is *transitive* if, whenever it contains the tuples a–>b and b–>c, it also contains a–>c, i.e.,
  - r.r **in** r
- The *transitive closure* ^r of a binary relation r, or just the closure for short, is the smallest relation that contains r and is transitive.

Intuition

- You can compute the transitive closure by taking the relation, adding the join of the relation with itself, then adding the join of the relation with that, and so on…
  - ^r = r + r.r + r.r.r + …
- Eventually you get to a point where adding another .r doesn't change anything, and then you can stop (technically, you've reached a *fixed-point*).

Transitive Closure

Example:

A relation hasChild which maps each person to their children.

Calculating…

```
hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5}
```

```
hasChild +
hasChild.hasChild
=
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5,
 P0 -> P3,
 P0 -> P4,
 P2 -> P5}
```

```
hasChild +
hasChild.
hasChild +
hasChild.
hasChild.
hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5,
 P0 -> P3,
 P0 -> P4,
 P2 -> P5,
 P0 -> P5)}
```

…at this point, we can't find anything else to add, so we are done.

```
^hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5,
 P0 -> P3,
 P0 -> P4,
 P2 -> P5,
 P0 -> P5)}
```

Transitive Closure

Example

A relation hasChild which maps each person to their children.

```
hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5}
```

Alternate Intuition: Reachability

The transitive closure of r can also be described as the relation that characterizes the atoms reachable (via the relation) from the each element in the domain of r in **one** or more steps through r.

```
hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5}
```

For example, from `P0` *we reach…*

`{P1, P2}` in one step
`{P3, P4}` in two steps
`{P5}` in three steps

```
^hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5,
 P0 -> P3,
 P0 -> P4,
 P2 -> P5,
 P0 -> P5)}
```

Reflexive Transitive Closure

Definitions

- A binary relation is *reflexive* if it contains the tuple a−>a for every atom a in **univ**, i.e.,
  - **iden in** r
- The *reflexive transitive closure* *r of a binary relation r is the smallest relation that contains r and is both reflexive and transitive.

Intuition

The reflexive transitive closure of r can also be described as the relation that characterizes the atoms reachable (via the relation **r**) from each element in **univ** in **zero** or more steps.

Reflexive Transitive Closure

Example

```
Person = {P0, P1, P2, P3}
hasChild = {P0 -> P1,
            P0 -> P2,
            P1 -> P3,
            P2 -> P3}
```

Constants

```
univ = {P0, P1, P2, P3}

iden =
  {P0 -> P0, P1 -> P1, P2 -> P2, P3 -> P3}
```

Transitive & Reflexive Transitive Closures

```
^hasChild = {P0 -> P1, P0 -> P2, P1 -> P3, P2 -> P3, P0 -> P3}
```

…from transitive closure

```
*hasChild = {P0 -> P1, P0 -> P2, P1 -> P3, P2 -> P3, P0 -> P3,
             P0 -> P0, P1 -> P1, P2 -> P2, P3 -> P3}
```

… reflexive tuples

Domain and Range Restrictions

Definitions

- **`s <: r`** contains those tuples of relation r that start with an element in set s (***domain restriction of r to s***)
- **`r :> s`** contains the tuples of r that end with an element in s (***range restriction of r to s***)

Examples

```
hasChild = {P0 -> P1,
            P0 -> P2,
            P3 -> P4}

myself = {P0}
friend = {P4}
```

```
myself <: hasChild =
              {P0 -> P1, P0 -> P2}

hasChild :> friend = {P3 -> P4}
```

Domain and Range Restrictions

Comparing Join and Domain / Range Restriction

```
hasChild = {P0 -> P1,
            P0 -> P2,
            P3 -> P4}

myself = {P0}
friend = {P4}
```

```
myself <: hasChild =
              {P0 -> P1, P0 -> P2}

hasChild :> friend = {P3 -> P4}
```

When working with sets such as myself and friend above, join and restriction. operations are similar. The difference is that join drops atoms from tuples whereas restrictions do not.

```
myself <: hasChild = { P0 -> P1, P0 -> P2 }
myself.hasChild    = { P1, P2}

hasChild :> friend = { P3 -> P4 }
hasChild.friend    = { P3 }
```

Override

Definition

- The override p ++ q of relation p by relation q is like the union, except that any tuple in p that matches a tuple of q by starting with same element is dropped. The relations p and q can have any matching arity of two or more.

Example

The capacity of aircraft is increased if the airline squeezes in more seats:

```
capacity = { A0 -> 6, A1 -> 7, A2 -> 5 }
changes = { A0 -> 7), A2 -> 7 }

capacity ++ changes = { A0 -> 7, A1 -> 7, A2 -> 7}
```

Acknowledgements

- Some of the material in this lecture has been adapted (with permission) from slides prepared by John Hatcliff and others for a course on Software Specifications at Kansas State University.