CIS 771:
Software Specifications

Lecture: Alloy Logic
(part C)

Outline

- Constants
- Set operations
- Relational operations

Constants

Special sets/relations constants

- **`none`** -- empty set (contains no elements)
- **`univ`** -- universal set (contains all elements)
- **`iden`** -- identity relation (relates each element to itself)

Examples

For the given model for Name and Address signatures…

```
Name = {(N0), (N1), (N2)}
Addr = {(D0), (D1)}
```

…the constants above have the following values…

```
none = {}
univ = {(N0), (N1), (N2), (D0), (D1)}
iden = {(N0, N0), (N1, N1), (N2, N2), (D0, D0), (D1, D1)}
```


Operations

Operation (with intuitive semantics)

- `+` (Union)
  - a tuple is in `p + q` when it is in `p` or in `q` (or both)
- `-` (Difference)
  - a tuple is in `p - q` when it is in `p` but not in `q`
- `&` (Intersection)
  - a tuple is in `p & q` when it is in `p` and in `q`
- `in` (Subset)
  - `p in q` is true when every tuple of `p` is also a tuple of `q`
- `=` (Equality)
  - `p = q` is true when `p` and `q` have the same tuples

Examples for Singletons

Consider the following set definitions…

```
Name = {(G0), (A0), (A1)}
Alias = {(A0), (A1)}
Group = {(G0)}
RecentlyUsed = {(G0), (A1)}
```

Examples of intersection, union, difference…

- `Alias & RecentlyUsed = {(A1)}`
  - gives the set of atoms that are aliases *and* have been recently used
- `Alias + Group = {(G0), (A0), (A1)}`
  - gives the set of atoms that are aliases *or* groups
- `Name - RecentlyUsed = {(A0)}`
  - gives the set of atoms that are names but have not been recently used;

Examples for Singletons

Consider the following set definitions…

```
Name = {(G0), (A0), (A1)}
Alias = {(A0), (A1)}
Group = {(G0)}
RecentlyUsed = {(G0), (A1)}
```

Examples of intersection, union, difference…

- `RecentlyUsed in Alias`
  - says that every thing that has been recently used is an alias, and is false, because of the tuple `{(G0)}`, which is recently used but not an alias;
- `RecentlyUsed in Name`
  - says that every thing that has been recently used is a name, and is true;
- `Name = Group + Alias`
  - says that every name is a group or an alias, and is true.

Examples for Relations

Consider the following set definitions…

```
cacheAddr = {(A0, D0), (A1, D1)}
diskAddr = {(A0, D0), (A1, D2)}
```

Examples of intersection, union, difference…

- `cacheAddr + diskAddr = {(A0, D0), (A1, D1), (A1, D2)}`
  - is the relation that maps a name to an address if it's mapped in the cache or on disk;
- `cacheAddr & diskAddr = {(A0, D0)}`
  - is the relation that maps a name to an address if it's mapped in the cache and on disk;

Examples for Relations

Consider the following set definitions…

```
cacheAddr = {(A0, D0), (A1, D1)}
diskAddr = {(A0, D0), (A1, D2)}
```

Examples of intersection, union, difference…

- `cacheAddr - diskAddr = {(A1, D1)}`
  - is the relation that maps a name to address if it's mapped in cache but not on disk;
- **`none in`** `diskAddr`
  - says that the empty relation is contained in the relation `diskAddr`, and is true, irrespective of the value of `diskAddr`;
- `cacheAddr = diskAddr`
  - says that the mappings in the cache are the same as those on disk, and is false, because of the tuple `(A1, D1)` in `cacheAddr` and `(A1, D2)` in `diskAddr`.

For You To Do

Given the sets below that can correspond to some Alloy address book model...

```
Book = {(B0), (B1)}
WorkBook = {(B0)}
HomeBook = {(B1)}

Addr = {(A1), (A2), (A3), (A4), (A5)}
USAddr = {(A1), (A3), (A7)}
InternationalAddr = {(A2), (A4)}
Name = {(N0), (N1), (N3), (N7)}

addr = {(B0,N0,A1), (B0,N1,A2), (B1,N3,A2), (B1,N3,A1), (B1,N0,A1)}
```

- What is the value of each of the following Alloy constants…
  - `univ`
  - `iden`
- What is the value of each of the following Alloy expressions…
  - `univ - Addr`
  - `(Addr - USAddr) = (InternationalAddr + none)`
  - `Addr & USAddr`
  - `USAddr in Addr`
  - `(Addr & USAddr) + InternalAddr`
  - `(Addr & USAddr) + (Book - WorkBook)`
  - `WorkBook.addr & HomeBook.addr`
  - `WorkBook.addr + HomeBook.addr`

Relational Operations

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

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

```
Name = {(N0), (N1)}
Addr = {(D0), (D1)}
Book = {(B0)}
```

we have

```
Name -> Addr = {(N0, D0), (N0, D1), (N1, D0), (N1, D1)}
```

...the relation mapping all names to all addresses;

```
Book -> Name -> Addr = {(B0, N0, D0), (B0, N0, D1),
                        (B0, N1, D0), (B0, N1, D1)}
```

...the relation associating books, names and addresses in all possible ways.

When p, q are sets, p->q is a binary relation.

If one of p or q has an arity of two or more, then p->q will be a multi-relation

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

`to`

maps a message to the names to which it should be sent

```
{(M0, N0),
 (M0, N2),
 (M1, N2),
 (M2, N3)}
```

`address`

maps names to addresses

```
{(N0, D0),
 (N0, D1),
 (N1, D1),
 (N1, D2),
 (N2, D3),
 (N4, D3)}
```

`to.address`

maps a messages to addresses to which it should be sent

```
{(M0, D0),
 (M0, D1),
 (M0, D3),
 (M1, D3)}
```

Dot Join -- Relation Composition

Step by step (with a smaller example)…

```
to = {(M0, N0),      address = {(N0, D0),
      (M0, N2)}                (N0, D1),
                               (N1, D1),
                               (N1, D2),
                               (N2, D3),
                               (N4, D3)}
```

Step 1:

form product (to->address)

```
{(M0, N0, N0, D0),
 (M0, N0, N0, D1),
 (M0, N0, N1, D1),
 (M0, N0, N1, D2),
 (M0, N0, N2, D3),
 (M0, N0, N4, D3),
 (M0, N2, N0, D0),
 (M0, N2, N0, D1),
 (M0, N2, N1, D1),
 (M0, N2, N1, D2),
 (M0, N2, N2, D3),
 (M0, N2, N4, D3)}
```

Step 2:

remove tuples where values of last column of p and first column of q do not match

Step 3:

from remaining tuples, drop last column of p and first column of q

```
to.address =
{(M0, D0),
 (M0, D1),
 (M0, D3)}
```


Dot Join

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

```
Alias = {(A0), (A1)}
Group = {(G0)}
Addr  = {(D0), (D1), (D2)}

address = {(G0, A0),
           (G0, A1),
           (A0, D0),
           (A1, D1)}
```

Consider signatures Alias, Group, Addr (which correspond to sets of singletons)

```
sig Alias, Group {
  address: Addr}
```

The address field is a binary relation between Alias/Group and Addr

Navigating (forward)

`Alias.address = {(D0), (D1)}`
the set of results obtained by looking up any alias in the address book;

`Group.address = {(A0), (A1)}`
the set of results obtained by looking up any group in the address book;

Dot Join

Step-by-step with the `Alias.address` ...

```
Alias = {(A0), (A1)}   address = {(G0, A0),
                                  (G0, A1),
                                  (A0, D0),
                                  (A1, D1)}
```

Step 1:
form product (Alias->address)

```
{(A0, G0, A0),
 (A0, G0, A1),
 (A0, A0, D0),
 (A0, A1, D1),
 (A1, G0, A0),
 (A1, G0, A1),
 (A1, A0, D0),
 (A1, A1, D1)}
```

Step 2:
remove tuples where values of last column of Alias and first column of address do not match

Step 3:
from remaining tuples, drop last (and only) column of Alias and first column of address

```
Alias.address =
{(D0), (D1)}
```


Dot Join

Navigation can also proceed in the reverse direction...

```
Alias = {(A0), (A1)}
Group = {(G0)}
Addr  = {(D0), (D1), (D2)}
```

Consider signatures Alias, Group, Addr (which correspond to sets of singletons)

```
address = {(G0, A0),
           (G0, A1),
           (A0, D0),
           (A1, D1)}
```

```
sig Alias, Group {
  address: Addr}
```

The address field is a binary relation between Alias/Group and Addr

Navigating (backward)

`address.Alias = {(G0)}`
the set of names that when looked up in the address book yield aliases;

`address.Group = {}`
the set of names that when looked up in the address book yield groups;

Dot Join

Step-by-step with the `address.Alias` …

```
Alias = {(A0), (A1)}   address = {(G0, A0),
                                  (G0, A1),
                                  (A0, D0),
                                  (A1, D1)}
```

Step 1:

form product (address->Alias)

```
{(G0, A0, A0),
 (G0, A1, A0),
 (A0, D0, A0),
 (A1, D1, A0),
 (G0, A0, A1),
 (G0, A1, A1),
 (A0, D0, A1),
 (A1, D1, A1)}
```

Step 2:

remove tuples where values of last column of address and first column of Alias do not match

Step 3:

from remaining tuples, drop last (and only) column of address and first column of Alias

```
address.Alias =
{(G0)}
```


Dot Join

Example -- navigating to fields that represent mappings themselves… (tenary relations)

Given a particular address book `b`, and a ternary relation `addr` associating books, names and addresses…

```
b = {(B0)}
addr = {(B0, N0, D0),
        (B0, N1, D1),
        (B1, N2, D2)}
```

Computing the name-address mapping for book b…

```
b.addr = {(N0, D0), (N1, D1)}
```

Dot Join

Dot Join is associative (except for some special cases)…

Given a relation `addr` of arity four that contains the tuple `b->n->a->t` when `b` maps name `n` to address `a` at time `t`, and a book `b` and time `t`,…

```
addr = {(B0, N0, D0, T0),      t = {(T1)}
        (B0, N0, D1, T1),
        (B0, N1, D2, T0),      b = {(B0)}
        (B0, N1, D2, T1),
        (B1, N2, D3, T0),
        (B1, N2, D4, T1)}
```

Computing the name-address mapping for book b at time t…

```
(b.addr).t = {(N0, D1), (N1, D2)}
b.(addr.t) = {(N0, D1), (N1, D2)}
```


For You To Do…

- Using the values of sets from the previous slide, calculate by hand the values of `(b.addr).t` and `b.(addr.t)` using the "three step" approach illustrated earlier for calculating dot joins.
- Given the set values below, use the "three step" approach to calculate `to.address`
- Given the set values below, use the "three step" approach to calculate `address.to`

```
to = {(M0, N0),      address = {(N0, D0),
      (M0, N1)}                 (N0, D1),
                                (N1, D1),
                                (N1, D2),
                                (N2, D3),
                                (N4, D3)}
```

Dot Join

Special case where dot join is not associative…

- `(a.b).c` and `a.(b.c)` are not always equivalent
- Because of the dropped column, the arity of the join is always one less than the sum of the arities of its arguments.
- Example
  - If `s` and `t` are unary, and `r` is ternary the expression `t.r` will be binary and `s.(t.r)` will be unary.
  - The expression `s.t` however would have zero arity and is thus illegal, so `(s.t).r` is likewise illegal, and is not equivalent to `s.(t.r)`

Dot Join

Useful Tip: Dot Join can be used to project out the domain or range of a relation…

```
sig Name, Addr {}
sig Book {
    addr: Name -> lone Addr
}
b = {(B0)}
```

- `Names.(b.addr)` returns the range of `(b.addr)` -- all addresses in `b`'s address book
- `(b.addr).Addr` returns the domain of `(b.addr)` -- all names in `b`'s address book
- In these examples, no tuples are dropped (everything matches) -- we are relying on the join's "dropping of tuple columns" to remove the domain atoms in the first case, and the range atoms in the second case

Box Join

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

```
e1[e2]
```
…has the same meaning as…
```
e2.e1
```

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 operator notation `[]` suggests to the reader that, e.g., a key to a lookup table indexes into that table in a mannar similar to notation used for array indexing…

Given a ternary relation `addr` associating books, names, and addresses, the expression

```
b.addr[n]
```

denotes the set of addresses associated with name `n` in book `b` (the key `n` associated with lookup is in the box), and is equivalent to

```
n.(b.addr)
```

(where the key `n` heads the navigation sequence)

Transpose

Definition

- The *transpose* ~*r* of a binary relation *r* takes its mirror image, forming a new relation by reversing the order of atoms in each tuple

Example

Given a relation representing an address book that maps names to the addresses they stand for…

```
address = {(N0, D0), (N1, D0), (N2, D2)}
```

…its transpose is the relation that maps each address to the names that stand for it…

```
~address = {(D0, N0), (D0, N1), (D2, N2)}
```


For You To Do…

Here are some cool ways to use ~ to state properties of relations…

- `s.~r = r.s`
  - is the image of the set `s` navigating backwards through the relation `r`;
- `r.~r`
  - is the relation that associates two atoms in the domain of the relation `r` when they map to a common element;
  - when `r` is a function, `r.~r` is the equivalence relation that equates atoms with the same image.
- `r.~r in iden`
  - says that `r` is injective
- `~r.r in iden`
  - says that `r` is functional

For you to do…

- For each of the properties above, use a simple example with explicitly defined relations and tuples that illustrates the property.
- For example, for the second bullet, say "if `mother` is the relation that maps a child to its mother, the expression `mother.~mother` is the sibling relation that maps a child to its siblings (and also to itself)" …then go ahead and define a example mother relation to illustrate the effects of the operators and explain how the result illustrates the property.

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 `addr` representing an address book with multiple levels (which maps aliases and groups to groups, aliases and addresses)…

```
addr =
{(G0, A0),
 (G0, G1),
 (A0, D0),
 (G1, D0),
 (G1, A1),
 (A1, D1),
 (A2, D2)}
```

Calculating…

```
addr =
{(G0, A0),
 (G0, G1),
 (A0, D0),
 (G1, D0),
 (G1, A1),
 (A1, D1),
 (A2, D2)}
```

```
addr +
addr.addr =
{(G0, A0),
 (G0, G1),
 (A0, D0),
 (G1, D0),
 (G1, A1),
 (A1, D1),
 (A2, D2),
 (G0, D0),
 (G0, A1),
 (G1, D1)}
```

```
addr +
addr.addr +
addr.addr.addr =
{(G0, A0),
 (G0, G1),
 (A0, D0),
 (G1, D0),
 (G1, A1),
 (A1, D1),
 (A2, D2),
 (G0, D0),
 (G0, A1),
 (G1, D1),
 (G0, D1)}
```

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

```
^addr =
{(G0, A0),
 (G0, G1),
 (A0, D0),
 (G1, D0),
 (G1, A1),
 (A1, D1),
 (A2, D2),
 (G0, D0),
 (G0, A1),
 (G1, D1),
 (G0, D1)}
```

Transitive Closure

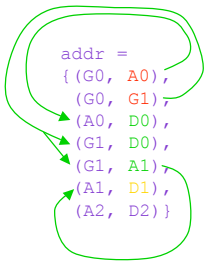
Example

A relation `addr` representing an address book with multiple levels (which maps aliases and groups to groups, aliases and addresses)…

```
addr =
{(G0, A0),
 (G0, G1),
 (A0, D0),
 (G1, D0),
 (G1, A1),
 (A1, D1),
 (A2, D2)}
```

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`.



For example, from `G0` we reach…

`{A0, G1}` in one step
`{D0, A1}` in two steps
`{D1}` in three steps

```
^addr =
{(G0, A0),
 (G0, G1),
 (A0, D0),
 (G1, D0),
 (G1, A1),
 (A1, D1),
 (A2, D2),
 (G0, D0),
 (G0, A1),
 (G1, D1),
 (G0, D1)}
```


Reflexive Transitive Closure

Definitions

- A binary relation is *reflexive* if it contains the tuple `a->a` for every atom `a`, 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 transitive closure of `r` can also be described as the relation that characterizes the atoms reachable (via the relation) from the each element in **univ** in **zero** or more steps through `r`.

Reflexive Transitive Closure

Example

```
Book = {(B0),(B1)}
Name = {(N0),(N1)}
Addr = {(D0),(D1)}
b = {(B0)}
addr = {(B0,N0,N1), (B0,N1,D0),(B1,N1,D1)}
```

Constants

```
univ = {(B0),(B1),(N0),(N1),(D0),(D1)}
iden = {(B0,B0),(B1,B1),(N0,N0),(N1,N1),(D0,D0),(D1,D1)}
```

Transitive & Reflexive Transitive Closures

```
^(b.addr) = {(N0,N1),(N1,D0),(N0,D0)}

*(b.addr) = {(N0,N1),(N1,D0),(N0,D0),
             (N0,N0),(N1,N1),(D0,D0),
             (B0,B0),(B1,B1),(D1,D1)}
```

…from transitive closure

…relevant reflexive tuples

…"irrelevant" reflexive tuples from `univ`

For You To Do

Consider the following sets/relations…

```
Group = {(G0),(G1),(G2)}
Addr  = {(A0),(A1),(A2)}
Dest  = {(D0),(D1),(D2)}
```

```
addr =
{(G0, A0),
 (G0, A1),
 (A0, D0),
 (G1, D0),
 (G1, G0),
 (G2, G1),
 (G2, A2),
 (A1, D1),
 (A2, D2)}
```

- Find an expression of the form `addr. [...] .addr` such that `addr. [...] .addr = ^addr`. Show the values of each of each step in the composition above, i.e., write the value of `addr`, `addr.addr`, `addr.addr.addr`, etc. For each of these note what new tuples are added over the previous step and why.
- Show the value of the expression: `*addr`
- Write a little Alloy model with associated commands to prove that the answers that you have provided for `addr. [...] .addr` and `*addr` above are correct.

Domain and Range Restrictions

Definitions

- `s <: r` contains those tuples of relation `r` that start with an element in set `s`
- `r :> s` contains the tuples of `r` that end with an element in `s`

Examples

```
addr = {(N0,A1),
        (N0,A2),
        (N1,A3)}

Name = {(N0)}
Addr = {(A3)}
```

```
Name <: addr = {(N0,A1),(N0,A2)}

addr :> Addr = {(N1,A3)}
```


Domain and Range Restrictions

Example

Given a relation representing a multi-level address book and sets representing the aliases, groups and addresses

```
address = {(G0, A0), (G0, G1), (A0, D0),
           (G1, D0), (G1, A1), (A1, D1), (A2, D2)}
Alias = {(A0), (A1), (A2)}
Group = {(G0), (G1)}
Addr = {(D0), (D1), (D2)}
```

- `address :> Addr = {(A0, D0), (G1, D0), (A1, D1), (A2, D2)}`
  - contains the entries that map names to addresses (and not to other names);
- `address :> Alias = {(G0, A0), (G1, A1)}`
  - contains the entries that map names to aliases;
- `Group <: address = {(G0, A0), (G0, G1), (G1, D0), (G1, A1)}`
  - contains the entries that map groups.

Domain and Range Restrictions

Comparing Join and Domain / Range Restriction

```
addr = {(N0,A1),
        (N0,A2),
        (N1,A3)}

Name = {(N0)}
Addr = {(A3)}
```

```
Name <: addr = {(N0,A1),(N0,A2)}

addr :> Addr = {(N1,A3)}
```

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

```
Name <: addr = {(N0,A1),(N0,A2)}
Name.addr    = {(A1),(A2)}

addr :> Addr = {(N1,A3)}
addr.Addr    = {(N1)}
```


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

An address book might be represented by two relations, *`homeAddress`* and *`workAddress`*, mapping an alias to email addresses at home and at work:

```
homeAddress = {(A0, D1), (A1, D2), (A2, D3)}
workAddress = {(A0, D0), (A1, D2)}

homeAddress ++ workAddress = {(A0, D0), (A1, D2), (A2, D3)}
```

Override

Example

Insertion of a key `k` with value `v` into a hashmap can be modelled by representing the value of the map before and after as two relations `m` and `m'` from keys to values, satisfying

```
m' = m ++ k -> v
```

Example

The environment `e` of an executing Java program can be viewed (simplistically) as a relation mapping variables to object references. The effect of an assignment

```
x = y
```

with a variable on both sides is

```
e' = e ++ x -> y.e
```


For You To Do

Consider the following sets/relations…

```
Group = {(G0),(G1),(G2)}
Addr  = {(A0),(A1),(A2)}
Dest  = {(D0),(D1),(D2)}
a0 = {(A0)}  d0 = {(D0)}
a1 = {(A1)}  d1 = {(D1)}
```

```
addr =
{(G0, A0),
 (G0, A1),
 (A0, D0),
 (G1, D0),
 (G1, G0),
 (G2, G1),
 (G2, A2),
 (A1, D1),
 (A2, D2)}
```

```
batchupdate1 =
{(G0, D0),
 (G1, A1)}

batchupdate2 =
{(G0, A2),
 (G1, A0)}
```

- Give the value of the following expressions:
  - `Group <: addr`
  - `Addr <: addr`
  - `addr :> Dest`
  - `addr :> Group`
- Give the value of the following expressions:
  - `addr ++ a0->d1`
  - `addr ++ a0->d1 ++ a0->d1`
  - `addr ++ a0->d1 ++ a1->d1`
  - `addr ++ batchupdate1`
  - `addr ++ (batchupdate1 + batchupdate2)`

Acknowledgements

- The material in this lecture is based on Sections 3.4 from…
  - *Software Abstractions: Logic, Language, and Analysis,* Daniel Jackson, MIT Press, 2006.