Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

Spring 2013

Lecture 2

Alloy Analyzer

- Analyzes micro models of software
- Helps to
  - identify key properties of a software design
  - find conceptual errors (**not** implementation errors)
- **Small scope hypothesis**: many properties that do not hold have small counterexamples
- Exhaustively search for errors in all instances of bounded size

Example Applications

- Security protocols
- Train controllers
- File systems
- Databases
- Network protocols
- Software design/testing/repair/sketching

Many examples are shipped with Alloy. More can be found on the Alloy website.

Today: The Alloy Language

Chapters 3 and 4 of Daniel Jackson’s book

Atoms, Relations, Structures, and Models

- An Alloy model defines a set of structures (instances)
- A structure is built from atoms and relations
- An atom is a primitive entity that is
  - *indivisible*: it can’t be broken down into smaller parts
  - *immutable*: its properties do not change over time
  - *uninterpreted*: it does not have any build-in properties, unlike, e.g. integer numbers
- A relation is a mathematical object that relates atoms. It consists of a set of tuples, each tuple being a sequence of atoms.

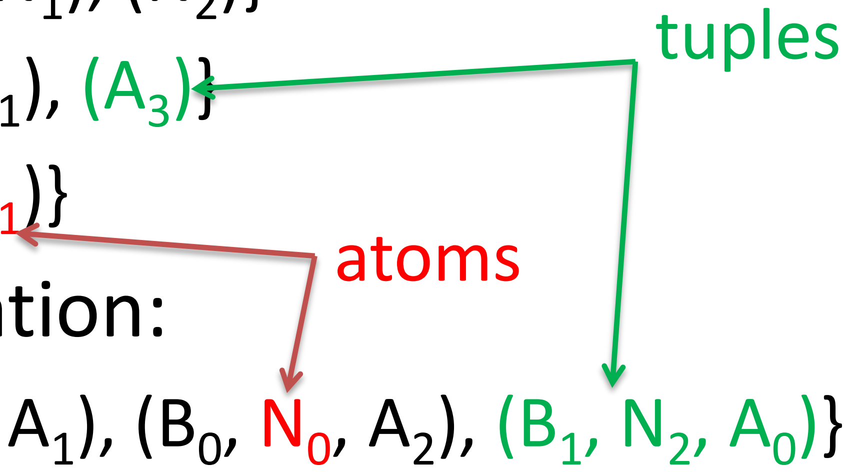
Atoms and Relations: Examples

- Three unary relations:

  Name = $\{(N_0), (N_1), (N_2)\}$

  Addr = $\{(A_0), (A_1), (A_3)\}$

  Book = $\{(B_0), (B_1)\}$



tuples

atoms

- A ternary relation:

  addr = $\{(B_0, N_0, A_1), (B_0, N_0, A_2), (B_1, N_2, A_0)\}$

Size and Arity of Relations

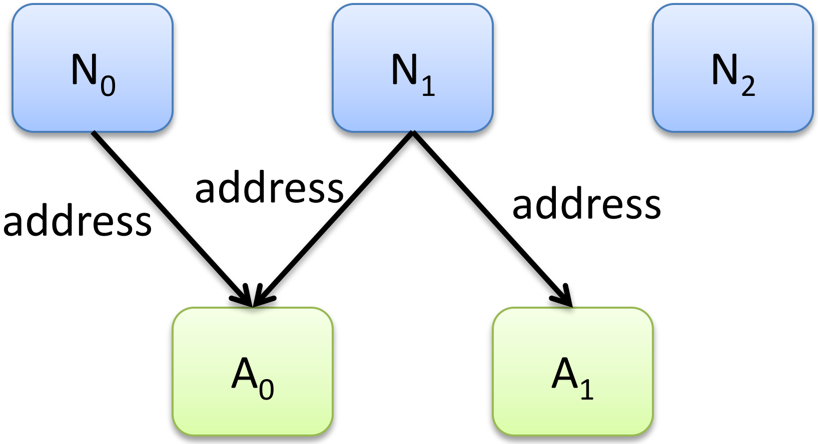
- The size (also cardinality) of a relation is the number of tuples in the relation.
- The arity of a relation is the number of atoms in each tuple of the relation.
- Examples:
  - A unary relation of size 3:
    $\text{Name} = \{(N_0), (N_1), (N_2)\}$
  - A ternary relation of size 2:
    $\text{addr} = \{(B_0, N_0, A_1), (B_0, N_1, A_2)\}$

Visualizing Structures: Snapshots

Name = $\{(N_0), (N_1), (N_2)\}$

Addr = $\{(A_0), (A_1)\}$

address = $\{(N_0, A_0), (N_1, A_0), (N_1, A_1)\}$



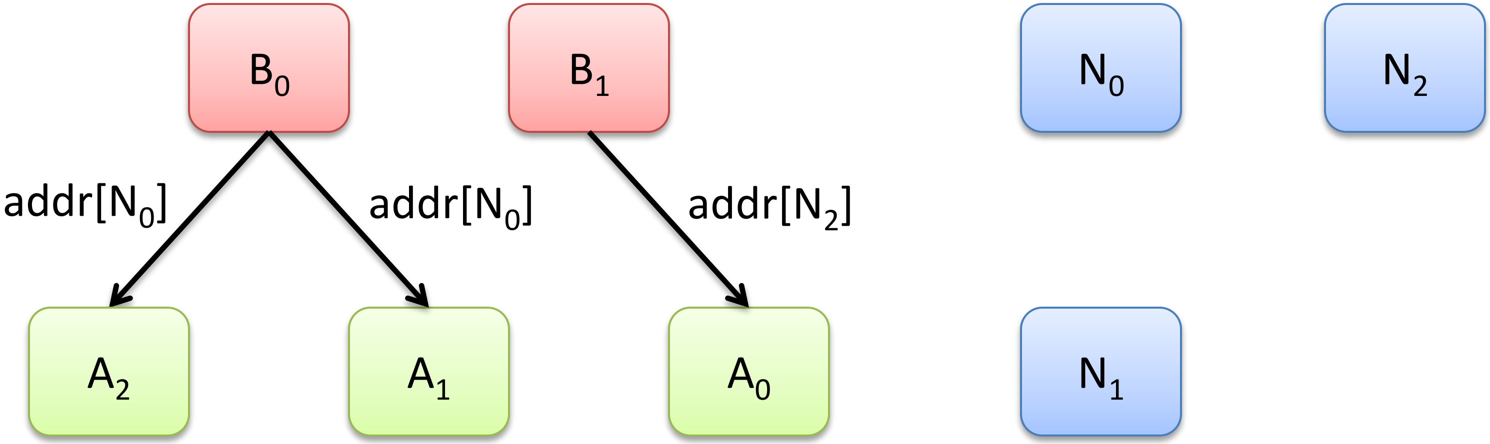
Visualizing Structures: Snapshots

Name = $\{(N_0), (N_1), (N_2)\}$

Addr = $\{(A_0), (A_1), (A_3)\}$

Book = $\{(B_0), (B_1)\}$

addr = $\{(B_0, N_0, A_1), (B_0, N_0, A_2), (B_1, N_2, A_0)\}$



Scalars, Sets, and Relations

In Alloy, everything is a relation, including scalars and sets

- Sets of atoms are unary relations

  Name = $\{(N_0),(N_1),(N_2)\}$

- Scalars are singleton sets

  n = $\{(N_0)\}$

- The following objects are treated as identical:

  x

  (x)

  {x}

  {(x)}

Domain and Range

- The domain of a relation is the set of atoms in the first position of its tuples
- The range of a relation is the set of atoms in the last position of its tuples
- Example:

  address = $\{(N_0, A_0), (N_1, A_1), (N_2, A_1)\}$

  domain(address) = $\{(N_0), (N_1), (N_2)\}$

  range(address) = $\{(A_0), (A_1)\}$

Alloy Specifications

- Signatures and Fields
  sets and relations defining the model
- Predicates and Functions
  operations and test predicates
- Facts
  assumptions about the model
- Assertions
  properties and conjectures
- Commands
  simulation, testing, and verification

Signatures and Fields

- Signatures
  - define the entities of your model
- Fields
  - define relations between signatures
- Signature constraints
  - multiplicity constraints on relations/signatures
  - signature facts

Signatures

- A signature introduces a set of atoms
- The declaration

  ```
  sig A {}
  ```

  introduces a set named A

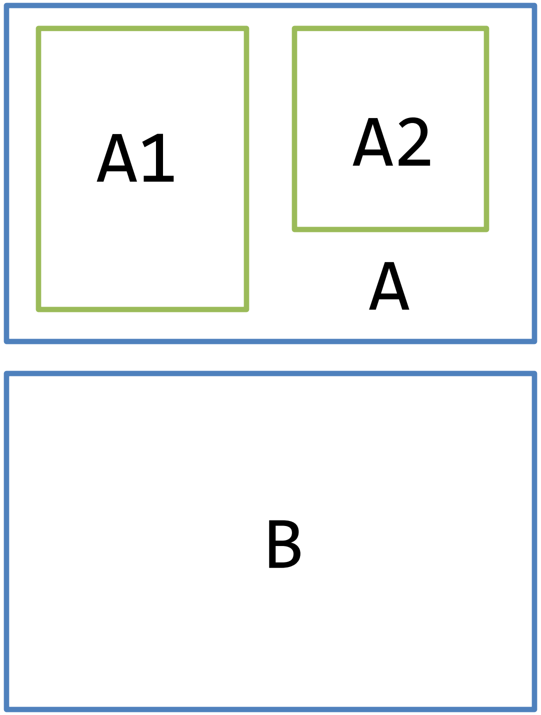
- A set can be introduced as a subset of another set

  ```
  sig A1 extends A {}
  ```

Signatures

- Signatures declared independently of each other define disjoint sets

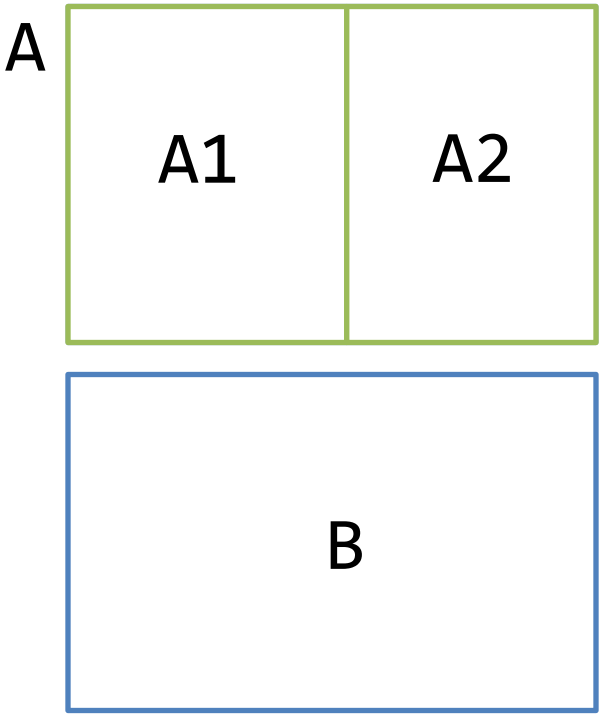
```
sig A {}
sig B {}
sig A1 extends A {}
sig A2 extends A {}
```



Signatures

- An abstract set only contains the elements of its extensions

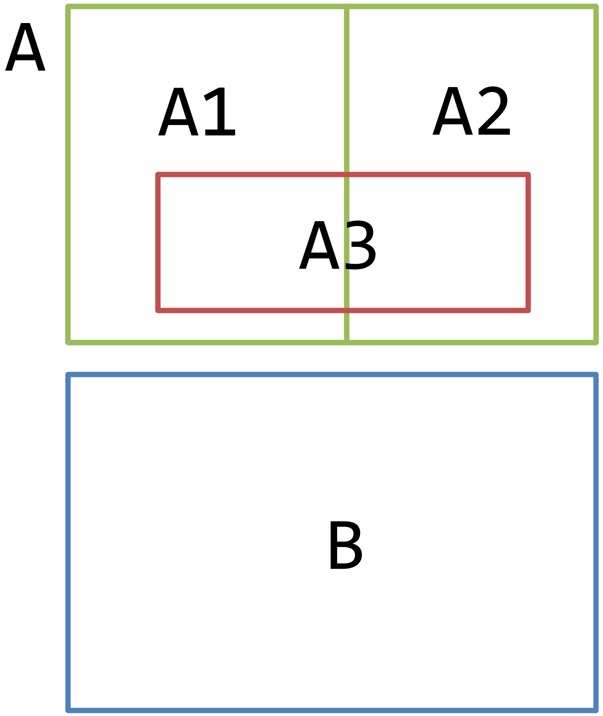
```
abstract sig A {}
sig B {}
sig A1 extends A {}
sig A2 extends A {}
```



Signatures

- Signatures can also be declared as subsets of other signatures without being disjoint

```
abstract sig A {}
sig B {}
sig A1 extends A {}
sig A2 extends A {}
sig A3 in A {}
```



Fields

- Relations are declared as fields of signatures
  - The declaration

    ```
    sig A {f: e}
    ```

    introduces a relation `f` whose domain is `A` and whose range is defined by the expression `e`.
- Examples:
  - A binary relation `f`: $A \times A$

    ```
    sig A {f: A}
    ```

  - A ternary relation `g`: $B \times A \times A$

    ```
    sig B {g: A -> A}
    ```

Multiplicities

Multiplicities constrain the sizes of sets and relations

- A multiplicity keyword placed before a signature declaration constraints the number of element in the signature's set

```
    m sig A {}
```

- We can also make multiplicities constraints on fields:

```
    sig A {f: m e}
    sig A {f: e1 m -> n e2}
```

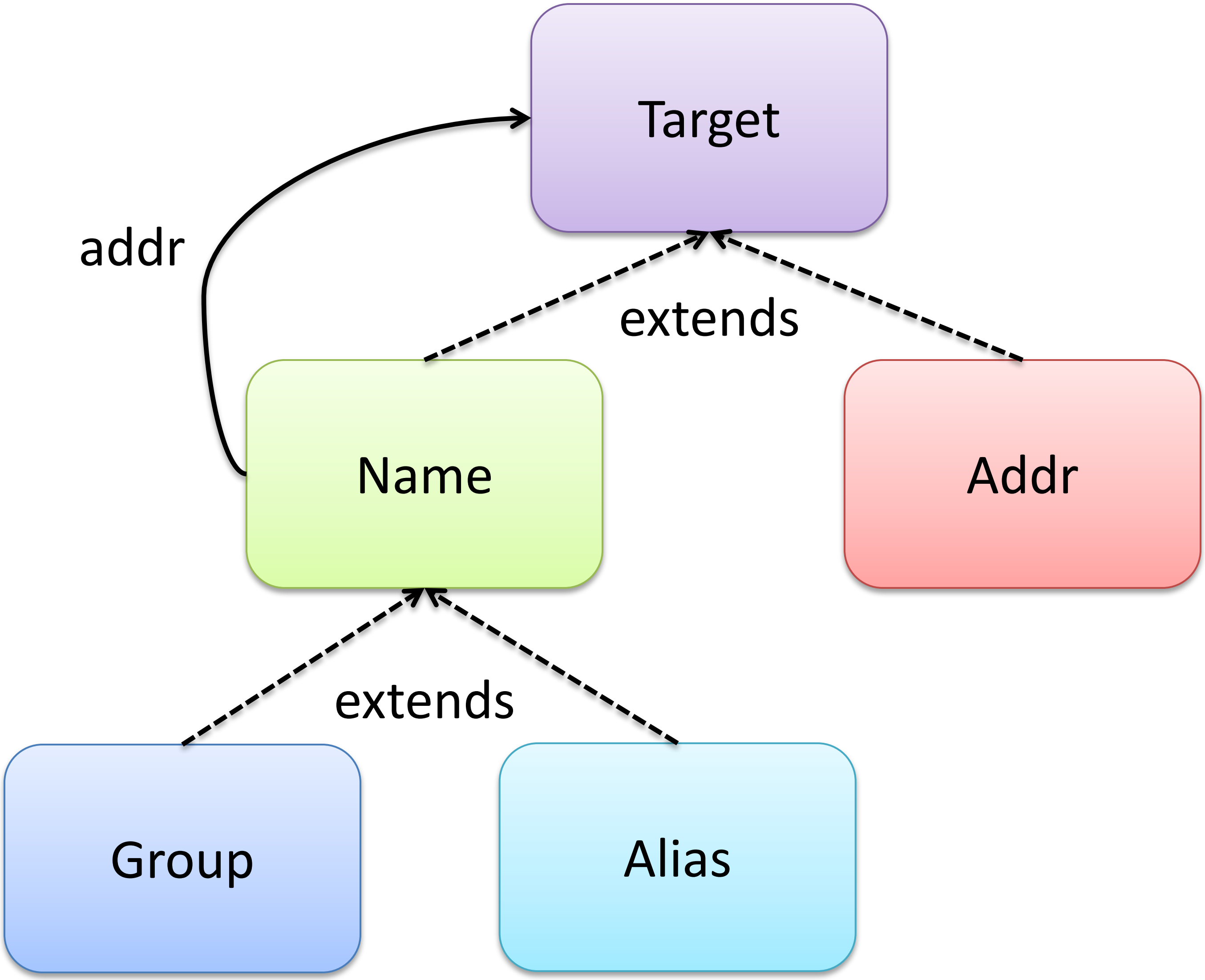
- There are four multiplicities
  - **`set`** : any number
  - **`some`** : one or more
  - **`lone`** : zero or one
  - **`one`** : exactly one

Multiplicities

- Examples
  - `RecentlyUsed: set Name`
  - `senderName: lone Name`
  - `addr: Alias ->lone Addr` //addr is partial function
  - `f: A ->one B` // f is total function
  - `f: A lone->one B` // f is total and injective function
- The default multiplicity keyword for unary relations is **one**

  `r: e` is equivalent to `r: one e`

Example: Hierarchical Address Book



Predefined Sets and Relations

- There are three predefined constants
  - `none` : empty set
  - `univ` : universal set
  - `iden` : identity
- Example: in the structure

  Name = $\{(N_0), (N_1), (N_2)\}$
  Addr = $\{(A_0), (A_1)\}$

  these constants are interpreted as

  `none` = $\{\}$
  `univ` = $\{(N_0), (N_1), (N_2), (A_0), (A_1)\}$
  `iden` = $\{(N_0, N_0), (N_1, N_1), (N_2, N_2), (A_0, A_0), (A_1, A_1)\}$

Set Operators

Alloy’s set operators are

`+`	union
`&`	intersection
`-`	difference
`in`	(subset) inclusion
`=`	equality

`Examples:`

- $\{(A_0),(A_1),(A_3)\} + \{(A_1),(A_2)\} = \{(A_0),(A_1),(A_2),(A_3)\}$
- $\{(N_0,A_1),(N_1,A_2)\} \mathbin{\&} \{(N_1,A_2),(N_2,A_1)\} = \{(N_1,A_2)\}$

Relational Operators

Alloy’s relational operators are

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

Arrow Product

- p -> q
  - p is n-ary relation, q is m-ary relation
  - p -> q is the (n+m)-ary relation obtained by pairwise concatenating all tuples in p and q

    $p \text{ -> } q = \{(x_1,\ldots,x_n,y_1,\ldots,y_m) \mid (x_1,\ldots,x_n) \in p,\ (y_1,\ldots,y_m) \in q\}$

- Examples:

  $n = \{(N)\}$, $a = \{(A_0)\}$, $Addr = \{(A_0),(A_1),(A_2)\}$, $Book = \{(B_0),(B_1)\}$

  $n \text{ -> } a = \{(N, A_0)\}$

  $n \text{ -> } Addr = \{(N, A_0), (N, A_1), (N, A_2)\}$

  $Book \text{ -> } n \text{ -> } Addr = \{(B_0,N, A_0), (B_0,N, A_1), (B_0,N, A_2),$
  $(B_1,N, A_0), (B_1,N, A_1), (B_1,N, A_2)\}$

Dot Join on Tuples

- Join of tuples $\{(x_1,\ldots,x_n)\}$ and $\{(y_1,\ldots,y_m)\}$ is
  - none if $x_n \neq y_1$
  - $\{(x_1,\ldots,x_{n-1},y_2,\ldots,y_m)\}$ if $x_n = y_1$
- Examples:
  - $\{(A,B)\}.\{(A,C)\} = \{\}$
  - $\{(A,B)\}.\{(B,C)\} = \{(A,C)\}$
  - $\{(A)\}.\{(A,C\}) = \{(C)\}$
  - $\{(A)\}.\{(A)\} = ?$ undefined
- Dot join is undefined if **both** n=1 and m=1

Dot Join

- `p.q`
  - p is n-ary relation, q is m-ary relation with $n>1$ or $m>1$
  - `p.q` is $(n+m-1)$-ary relation obtained by taking every combination of a tuple from p and a tuple from q and adding their join, if it exists.

    $p.q = \{(x_1,\ldots,x_{n-1},y_2,\ldots,y_m) \mid (x_1,\ldots,x_n) \in p,\ (y_1,\ldots,y_m) \in q,\ x_n=y_1\}$

Dot Join: Example

to maps messages to the names of the recipients:

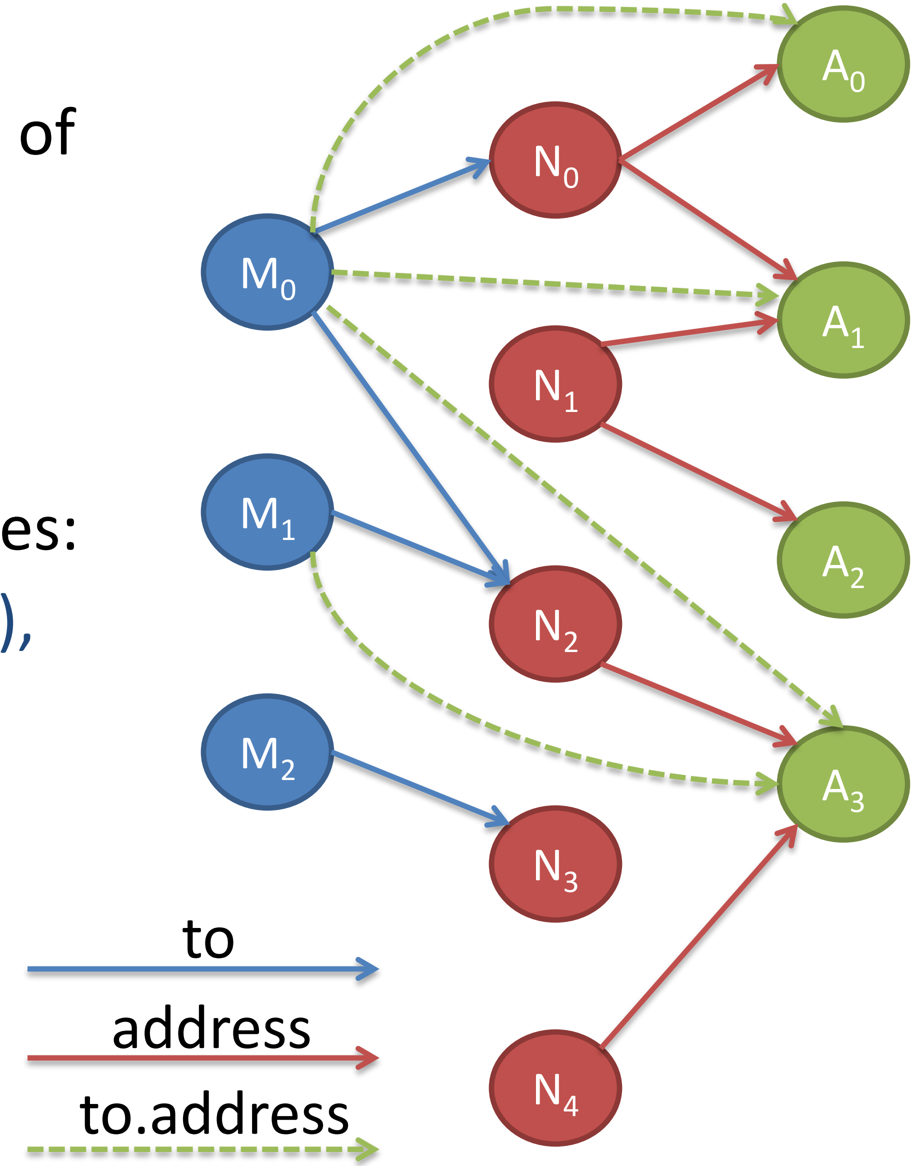
$to = \{(M_0,N_0),(M_0,N_2),$
$(M_1,N_2),(M_2,N_3)\}$

address maps names to addresses:

$address = \{(N_0,A_0),(N_0,A_1),(N_1,A_1),$
$(N_1,A_2),(N_2,A_3),(N_4,A_3)\}$

to.address maps messages to addresses of recipients:

$to.address = \{(M_0,A_0),(M_0,A_1),$
$(M_0,A_3),(M_1,A_3)\}$



Dot Join: Exercise

- Given the relations mother and father, how do you express the grandfather relation?

  grandfather =

Box Join

- `e1 [e2]`
  - is semantically equivalent to `e2.e1`

- Dot binds stronger than box

  `a.b.c [d]` is short for `d.(a.b.c)`

- Example: `b.addr[n]` denotes the addresses associated with name `n` in book `b`

Transpose

- `~ p`
  - denotes the mirror image of relation `p`

    $\sim p = \{(x_n,\ldots,x_1) \mid (x_1,\ldots,x_n) \in p\}$
- Example:

  $address = \{(N_0, A_0), (N_1, A_0), (N_2, A_1)\}$

  $\sim address = \{(A_0, N_0), (A_0, N_1), (A_1, N_2)\}$
- some useful facts:
  - `~(~p.~q)` is equal to `q.p`
  - if `p` is a unary and `q` binary then `p.~q` is equal to `q.p`
  - `p.~p` relates atoms in the domain of `p` that are related to the same element in the range of `p`
  - `p.~p in iden` states that `p` is injective

(Reflexive) Transitive Closure

- `^r`
  - `r` is binary relation
  - transitive closure `^r` is smallest transitive relation containing `r`

    `^r = r + r.r + r.r.r + ...`

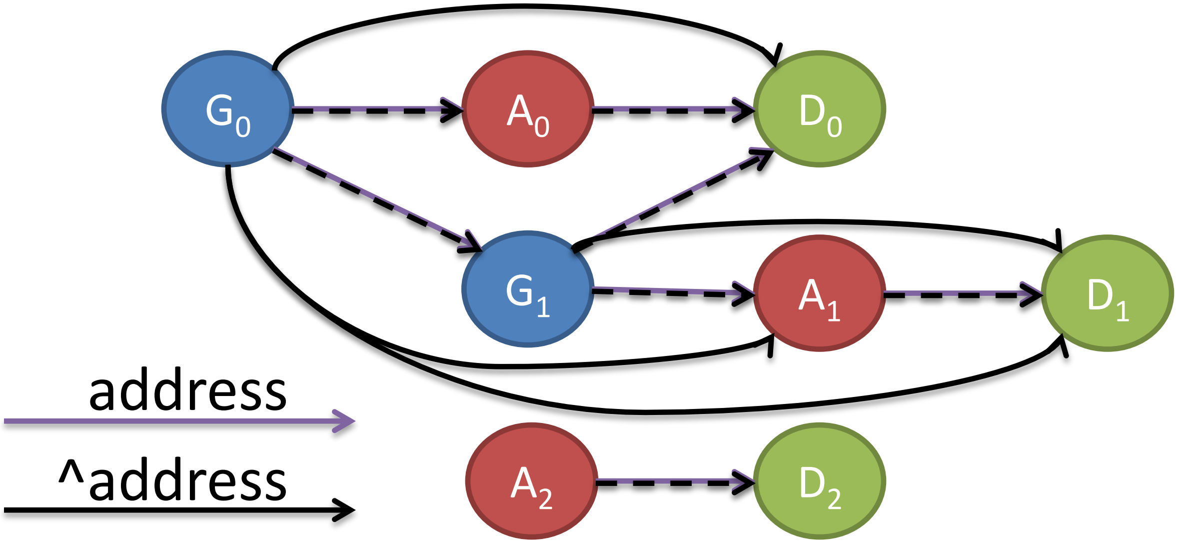
- reflexive transitive closure: `*r = ^r + iden`

Transitive Closure: Example

A relation address representing an address book with multiple levels (including groups of aliases):

$\text{address} = \{(G_0,A_0),(G_0,G_1),(A_0,D_0),(G_1,D_0),(G_1,A_1),(A_1,D_1),(A_2,D_2)\}$

$\hat{}\text{address} = \{(G_0,A_0),(G_0,G_1),(A_0,D_0),(G_1,D_0),(G_1,A_1),(A_1,D_1),(A_2,D_2),$
$(G_0,D_0),(G_0,A_1),(G_1,D_1),$
$(G_0,D_1)\}$



Transitive Closure: Exercise

- How would you express the ancestor relation in a family tree, given the children relation?

ancestor =

Domain and Range Restriction

- `s <: r`
  - `s` is a set and `r` a relation
  - `s <: r` is the relation obtained by restricting the domain of `r` to `s`
- `r :> s`
  - `r :> s` is the relation obtained by restricting the range of `r` to `s`
- Example:

siblings = $\{(M_0,W_0),(W_0,M_0),(W_1,W_2),(W_2,W_1)\}$

women = $\{(W_0),(W_1),(W_2)\}$

sisters = siblings :> women

Override

- p ++ q
  - p and q are relations
  - like union, but any tuple in p that starts with the same element as a tuple in q is dropped
- Example:

  homeAddress = $\{(N_0,A_1), (N_1,A_2), (N_2,A_3)\}$

  workAddress = $\{(N_0,A_0), (N_1,A_2)\}$

  homeAddress ++ workAddress = $\{(N_0,A_0), (N_1,A_2), (N_2,A_3)\}$
- Example: insertion of a key k with value v into a hashmap m:

  m’ = m ++ k->v

Logical Operators

Alloy’s logical operators are

`not`	`!`	negation
`and`	`&&`	conjunction
`or`	`||`	disjunction
`implies`	`=>`	implication
`else`	`,`	alternative
`iff`	`<=>`	biimplication

```
Examples: F implies G else H
          (F and G) or ((not F) and H)
```

Operator Precedence

```
||
<=>
=>
&&
!
=   !=   in
+   -
++
&
->
<:
:>
[]
.
~   *
```

lowest

highest

Quantifiers

- A quantified constraint takes the form

  `Q x: e | F`

- The forms of quantification in Alloy are

`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 fails for every x in S
`lone x: S | F`	F holds for at most 1 x in S
`one x: S | F`	F holds for exactly 1 x in S

Quantifiers

- Quantifiers can also be applied to expressions
  - **`some`** `e`	e has some tuple
  - **`no`** `e`	e has no tuples
  - **`lone`** `e`	e has at most one tuple
  - **`one`** `e`	e has exactly one tuple

Let Expressions

- `let x = e | A`
  - let expression can factor out a complicated subexpression `e` by introducing a short hand `x`
  - let expressions cannot be recursive, i.e., `x` is not allowed to appear in `e`
- Example: preferred address of an alias a is the work address, if it exists

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

Comprehensions

- `{x: e | F}`
  - the set of values `x` drawn from set `e` for which `F` holds
  - general form $\{x_1\!: e_1, \ldots, x_n\!: e_n \mid F\}$ defines an n-ary relation

- Example: relation mapping names to addresses in a multilevel address book

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

Facts

- Facts define additional assumptions about the signatures and fields of a model
- Alloy will only consider instances that also satisfy all facts
- Example:

```
fact Biology {
   no p: Person | p in p.^(mother + father)
}
```

Signature Facts

- Signature facts express assumptions about each element of a signature
- The declaration

  ```
  sig A { ... } { F }
  ```

  is equivalent to

  ```
  sig A { ... }
  fact {all this: A | F’}
  ```

  where `F’` is like `F` but with all appearances of fields `g` of `A` in `F` replaced by `this.g`

Signature Facts: Example

There are no cycles in the `addr` relation of an address book:

```
sig Book { addr : Name -> Target }
{ no n : Name | n in n.^addr }
```

Assertions

- An assertion is a constraint that is intended to follow from the facts of the model.
- Useful to
  - find flaws in the model
  - express properties in different ways
  - act as regression tests
- The analyzer checks assertions and produces a counterexample instance, if an assertion does not hold.
- If an assertion does not hold, you typically want to
  - move that constraint into a fact or
  - refine your specification until the assertion holds.

Assertions: Example

```
assert addIdempotent  {
  all b,b',b'' : Book, n : Name, a : Addr |
     add [b, b', n, a] and
     add [b', b'', n, a] implies
        b'.addr = b''.addr
}
```

Run Command

- Used to ask Alloy to generate an instance of the model
- May include conditions
  - Used to guide AA to pick model instances with certain characteristics
  - E.g., force certain sets and relations to be nonempty
  - In this case, not part of the “true” specification
- Alloy only executes the first run command in a file

Scope

- Limits the size of instances considered to make instance finding feasible
- Represents the maximum number of tuples in each top-level signature
- Default scope is 3

Run Command: Examples

- `run {}`
  - no condition
  - scope is 3

- `run {} for 5 but exactly 2 Book`
  - no condition
  - scope is 5 for all signatures except for signature Book, which should be of size exactly 2

- `run {some Book && some Name} for 2`
  - condition forces Book and Name to be nonempty
  - scope is 2

Functions and Predicates

- Functions and predicates define short hands that package expressions or constraints together. They can be
  - named and reused in different contexts (facts, assertions and conditions of run)
  - parameterized
  - used to factor out common patterns
- Predicates are good for:
  - Constraints that you don't want to record as facts
  - Constraints that you want to reuse in different contexts
- Functions are good for
  - Expressions that you want to reuse in different contexts

Functions

A function is a named expression with 0 or more arguments.

Examples

- The parent relation
```
fun parent [] : Person->Person {~children}
```
- The lookup function
```
fun lookup [b : Book, n : Name] : set Addr {
    n.^(b.addr) & Addr
}
```

Predicates

A predicate is a named constraint with 0 or more arguments.

Example:

```
pred ownGrandFather [p: Person] {
    p in p.grandfather
}
```

no person is its own grand father

```
no p: Person | ownGrandFather[p]
```