



From:  
*FM 2006 Alloy*  
*Intro and Logic*

Greg Dennis and Rob Seater  
Software Design Group, MIT



# agenda

---

- Intro & Logic
- Language & Analysis
- Dynamic Modeling



M.C. Escher

# Observations

---

- Software is built on abstractions
  - Pick the right ones, and programming will flow naturally from design.
    - an idea reduced to its essential form
  - Pick the wrong ones, and programming will be a series of nasty surprises.
- Our current strategy does not guarantee that the designs are correct.
- Formal specifications help to verify correctness but are hard to read and write.

send:

The program  $\text{send}(e_1, e_2).P$  sends a message with contents  $e_2$  to the actor indicated by  $e_1$ :

$${}^a[\text{send}(e_1, e_2).P]_s \xrightarrow{\tau} {}^a[P]_s \mid \langle [[e_1]_s^a], [[e_2]_s^a] \rangle$$

where  $\tau$  represents an internal invisible step of computation.

# New Approach - Alloy

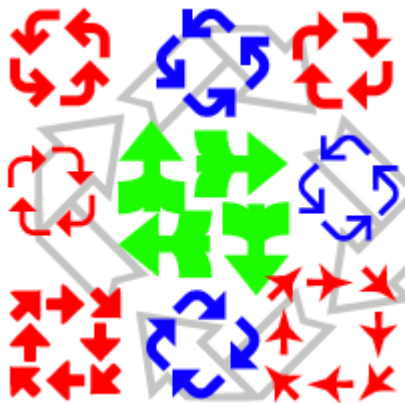
---

- Alloy takes from *formal specification* the idea of precise and expressive notation based on a tiny core of simple and robust concepts
- It replaces conventional analysis based on *theorem proving* with a fully automatic analysis.
- Analysis is not complete but examines a finite space of cases (configurable)
  - NO TEST CASES ARE REQUIRED
  - User specifies predicates to be checked

# four key ideas . . .

---

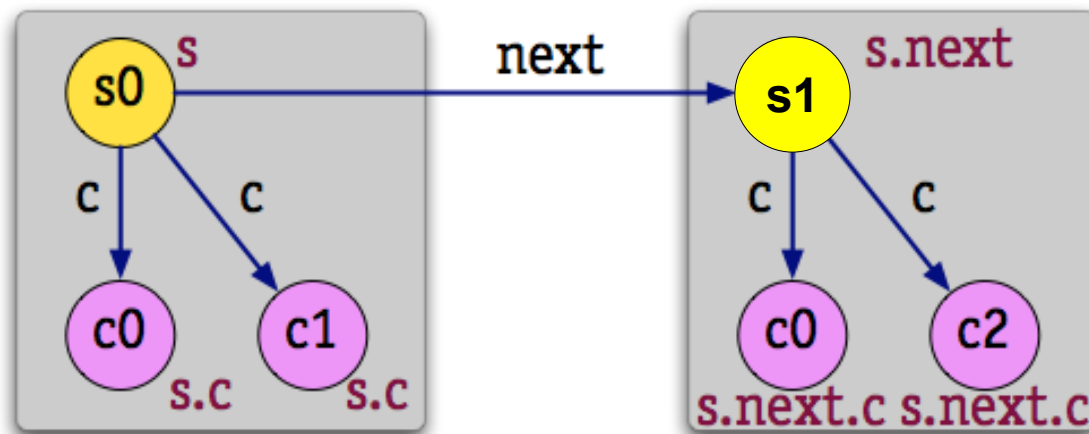
- 1) everything is a relation
- 2) non-specialized logic
- 3) counterexamples & scope
- 4) analysis by SAT



# 1) everything's a relation

---

- Alloy uses relations for
  - all datatypes – even sets, scalars, tuples
  - structures in space and time
- key operator is **dot** join
  - relational join
  - field navigation
  - ...



# why relations?

---

- easy to understand
  - binary relation is a graph or mapping
- easy to analyze
  - first order (tractable)
- uniform

*set of addresses associated with name n in set of books B*

Alloy:  $n.(B.addr)$

Z:  $\cup \{ b: B \bullet b.addr \mid \{n\} \mid \}$

OCL:  $B.addr[n] \rightarrow asSet()$

There is no problem in computer science that cannot be solved by an extra level of indirection.

– David Wheeler

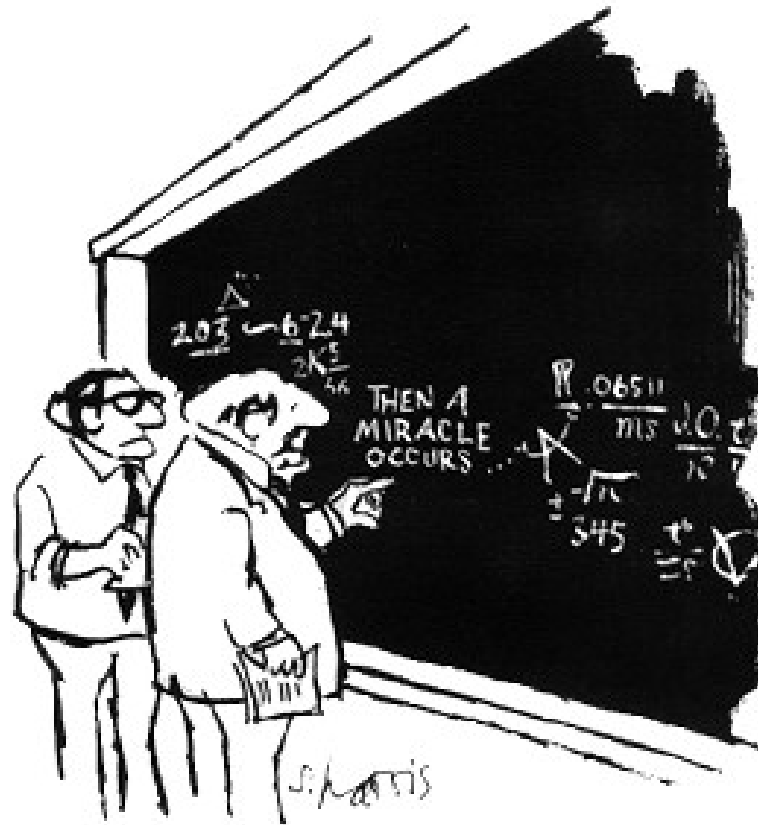


Wheeler

## 2) non-specialized logic

---

- No special constructs for state machines, traces, synchronization, concurrency . . .

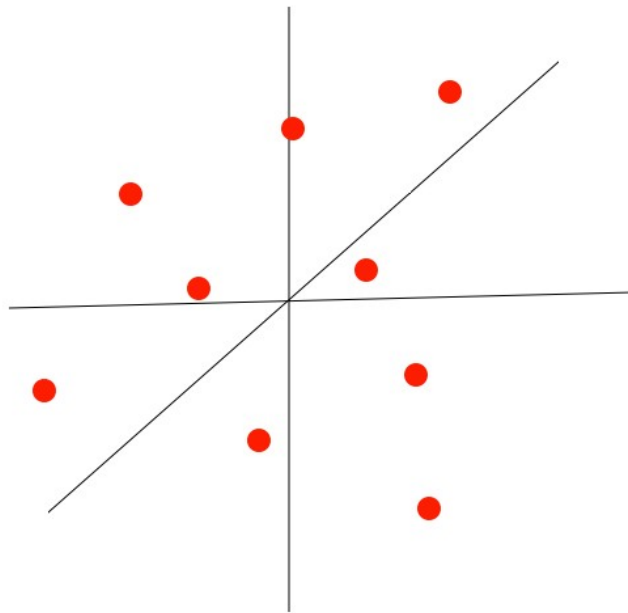


"I think you should be more explicit here in step two."

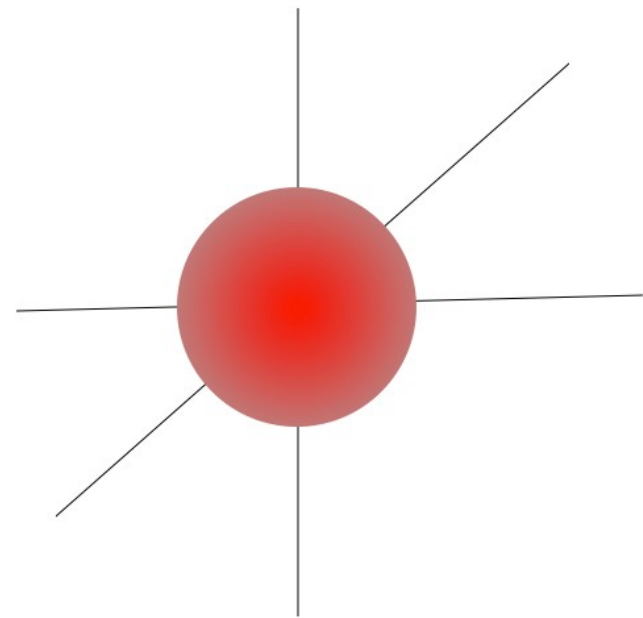
# 3) counterexamples & scope

---

- observations about design analysis:
  - most assertions are wrong
  - most flaws have small counterexamples



testing:  
a few cases of arbitrary size

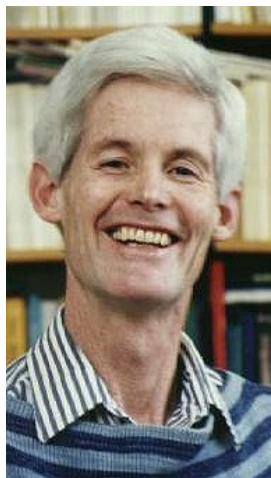


scope-complete:  
all cases within a small bound

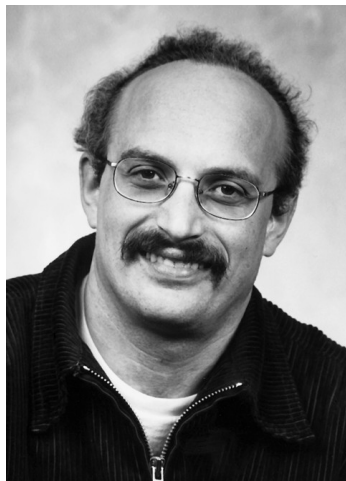
# 4) analysis by SAT

---

- SAT, the quintessential hard problem (Cook 1971)
  - SAT is hard, so reduce SAT to your problem
- SAT, the universal constraint solver (Kautz, Selman, ... 1990's)
  - SAT is easy, so reduce your problem to SAT
  - solvers: Chaff (Malik), Berkmin (Goldberg & Novikov), ...



Stephen  
Cook



Eugene  
Goldberg



Henry  
Kautz



Sharad  
Malik



Yakov  
Novikov

# SAT

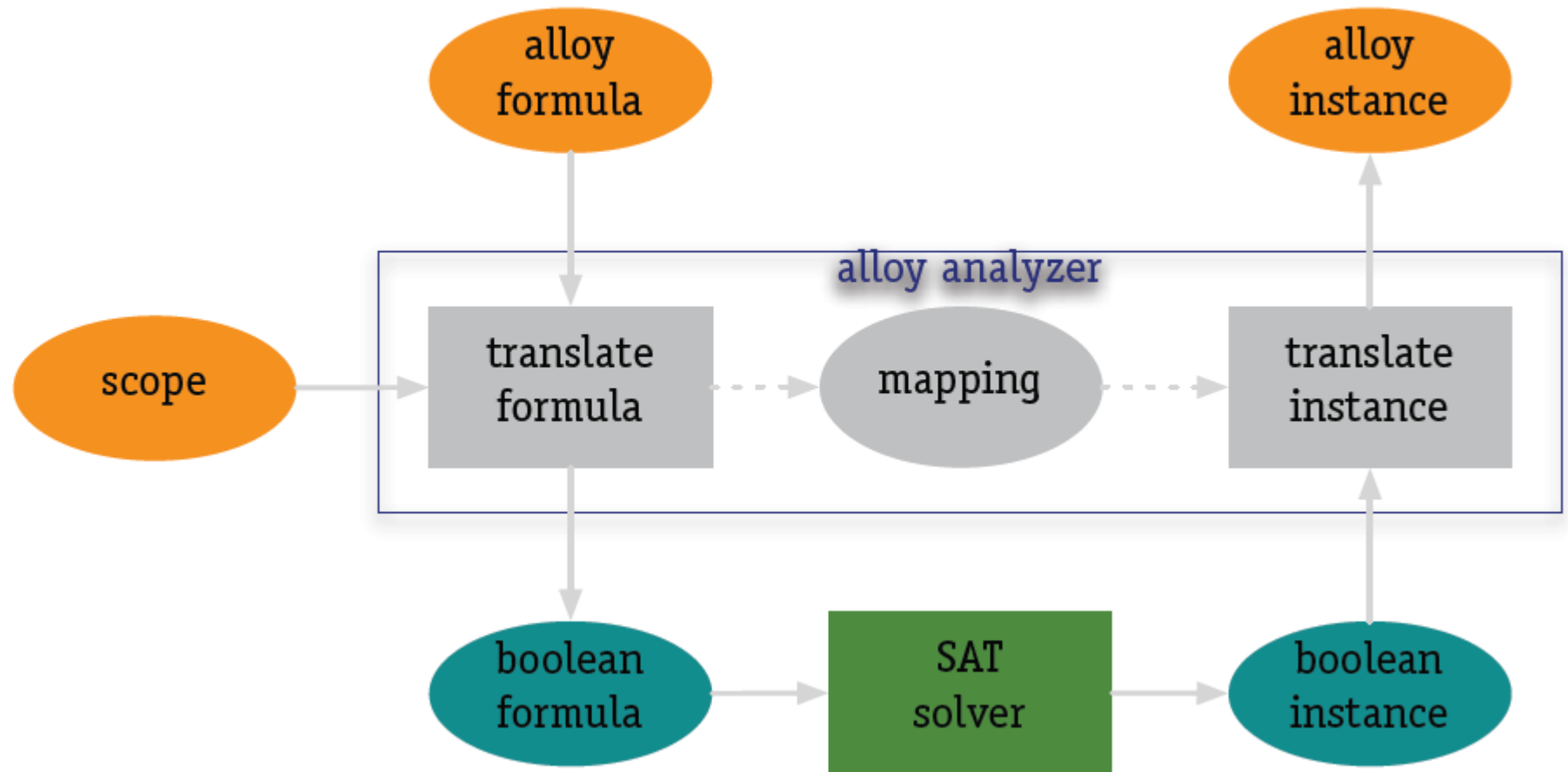
---

- The **Boolean satisfiability problem (SAT)** is a decision problem considered in complexity theory. An instance of the problem is a Boolean expression written using only **AND, OR, NOT**, variables, and parentheses. The question is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true?

$$(A \vee B \vee \neg B) \wedge (D \vee B) \dots$$

# Alloy analyzer architecture

---



# Try the Alloy Analyzer

---

- requires Java
  - <http://java.sun.com>
- download the Alloy Analyzer
  - <http://alloy.mit.edu>
- run the Analyzer
  - double click *alloy.jar* or
  - execute *java -jar alloy.jar* at the command line



# Example: modeling “ceilings and floors”

---

**sig** Platform {}

*there are “Platform” things*

**sig** Man {ceiling, floor: Platform}

*each Man has a ceiling and a floor Platform*

**pred** Above(m, n: Man) {m.floor = n.ceiling}

*Man m is “above” Man n if m's floor is n's ceiling*

**fact** {**all** m: Man | **some** n: Man | Above (n,m)}

*“One Man's Ceiling Is Another Man's Floor”*

# checking “ceilings and floors”

---

```
assert BelowToo {  
  all m: Man | some n: Man | Above (m,n)  
}
```

*"One Man's Floor Is Another Man's Ceiling"?*

**check BelowToo for 2**

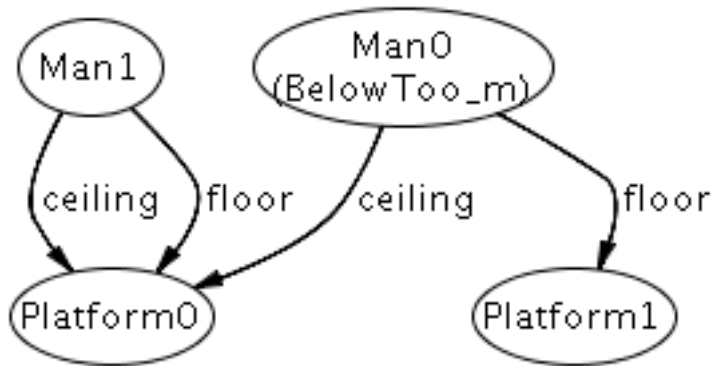
*check "One Man's Floor Is Another Man's Ceiling"*

*counterexample with 2 or less platforms and men?*

- clicking “Execute” ran this command
  - counterexample found, shown in graphic

# counterexample to “BelowToo”

---



McNaughton

# Alloy = logic + language + analysis

---

- logic
  - first order logic + relational calculus
- language
  - syntax for structuring specifications in the logic
- analysis
  - bounded exhaustive search for counterexample to a claimed property using SAT

# logic: relations of atoms

---

- atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted
- relations associate atoms with one another
  - set of tuples, tuples are sequences of atoms
- every value in Alloy logic is a relation!
  - relations, sets, scalars all the same thing

# logic: everything's a relation

---

- sets are unary (1 column) relations

Name = { (N0),  
          (N1),  
          (N2) }      Addr = { (A0),  
                              (A1),  
                              (A2) }      Book = { (B0),  
  (B1) }

- scalars are singleton sets

myName      = { (N1) }  
yourName    = { (N2) }  
myBook      = { (B0) }

- binary relation

names = { (B0, N0),  
          (B0, N1),  
          (B1, N2) }

- ternary relation

addrs = { (B0, N0, A0),  
          (B0, N1, A1),  
          (B1, N1, A2),  
          (B1, N2, A2) }

# logic: relations

---

addr = { (B0, N0, A0), (B0, N1, A1),  
(B1, N1, A2), (B1, N2, A2) }

<b>B0</b>	<b>N0</b>	<b>A0</b>
<b>B0</b>	<b>N1</b>	<b>A1</b>
<b>B1</b>	<b>N1</b>	<b>A2</b>
<b>B1</b>	<b>N2</b>	<b>A2</b>

size = 4

arity = 3

- rows are unordered
- columns are ordered but unnamed
- all relations are first-order
  - relations cannot contain relations, no sets of sets

# logic: address book example

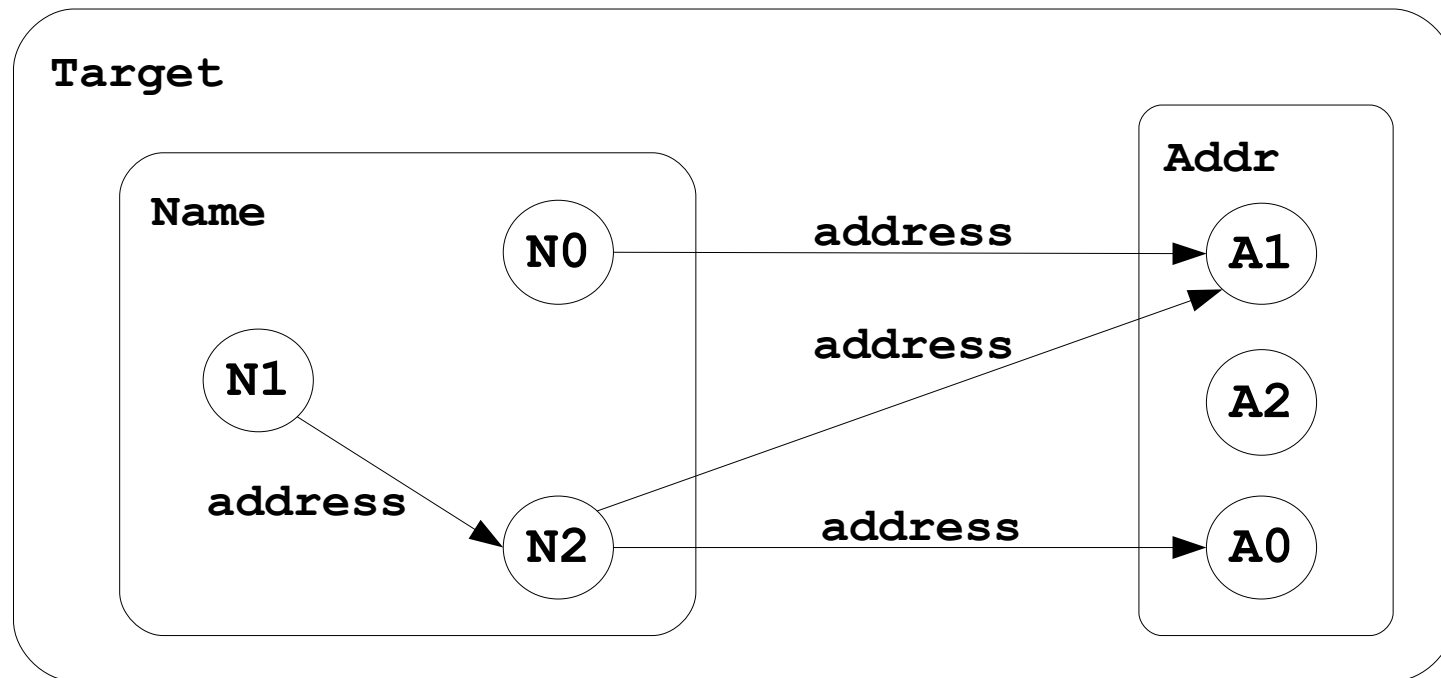
---

Name = { (N0), (N1), (N2) }

Addr = { (A0), (A1), (A2) }

Target = { (N0), (N1), (N2), (A0), (A1), (A2) }

address = { (N0, A1), (N1, N2), (N2, A1), (N2, A0) }



# logic: constants

---

<b>none</b>	<i>empty set</i>
<b>univ</b>	<i>universal set</i>
<b>iden</b>	<i>identity relation</i>

Name = { (N0), (N1), (N2) }

Addr = { (A0), (A1) }

**none** = { }

**univ** = { (N0), (N1), (N2), (A0), (A1) }

**iden** = { (N0, N0), (N1, N1), (N2, N2),  
(A0, A0), (A1, A1) }

# logic: set operators

---

+	<i>union</i>
&	<i>intersection</i>
-	<i>difference</i>
<b>in</b>	<i>subset</i>
=	<i>equality</i>

```
Name = { (N0), (N1), (N2) }
Alias = { (N1), (N2) }
Group = { (N0) }
RecentlyUsed = { (N0), (N2) }
```

```
Alias + Group = { (N0), (N1), (N2) }
Alias & RecentlyUsed = { (N2) }
Name - RecentlyUsed = { (N1) }
RecentlyUsed in Alias = false
RecentlyUsed in Name = true
Name = Group + Alias = true
```

```
greg = { (N0) }
rob = { (N1) }
```

```
greg + rob = { (N0), (N1) }
greg = rob = false
rob in none = false
```

```
cacheAddr = { (N0, A0), (N1, A1) }
diskAddr = { (N0, A0), (N1, A2) }
```

```
cacheAddr + diskAddr = 
cacheAddr & diskAddr = 
cacheAddr = diskAddr = 
```

# logic: product operator

---


-> *cross product*

```
Name = { (N0), (N1) }  
Addr = { (A0), (A1) }  
Book = { (B0) }
```

```
Name->Addr = { (N0, A0), (N0, A1),  
              (N1, A0), (N1, A1) }
```

```
Book->Name->Addr =  
  { (B0, N0, A0), (B0, N0, A1),  
    (B0, N1, A0), (B0, N1, A1) }
```

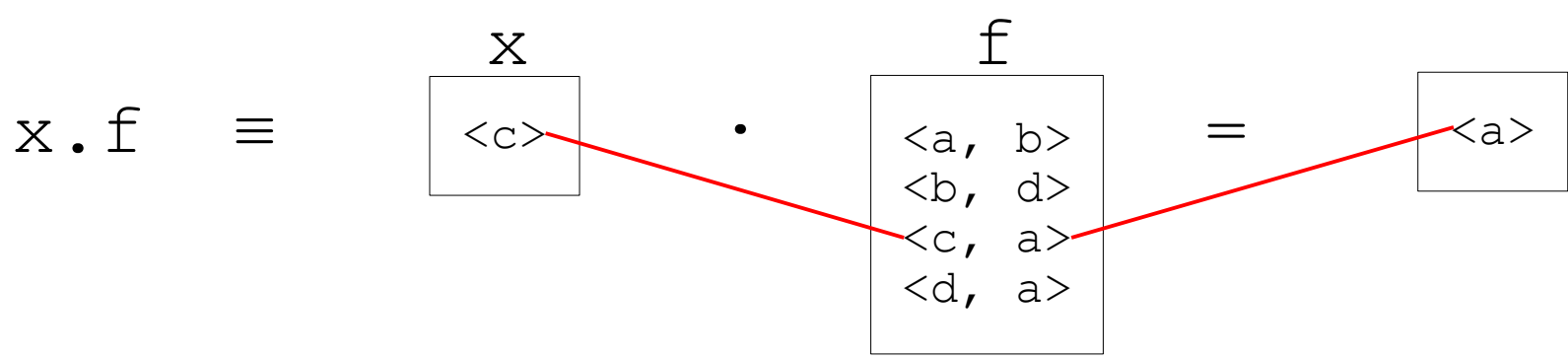
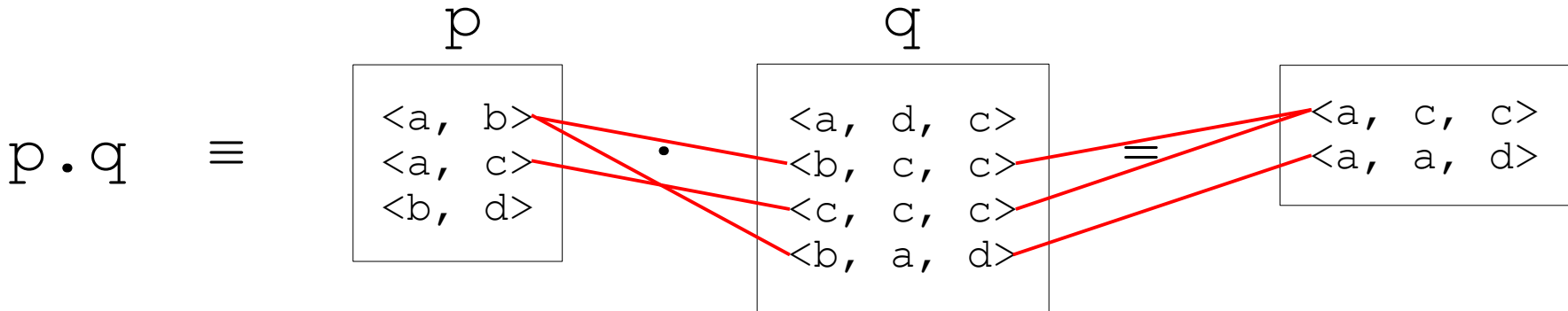
```
b = { (B0) }  
b' = { (B1) }  
address = { (N0, A0), (N1, A1) }  
address' = { (N2, A2) }
```

b->b' = 

b->address + b'->address' =  


# logic: relational join

---



# logic: join operators

---

.        *dot join*  
[]       *box join*

$e1[e2] = e2.e1$   
 $a.b.c[d] = d.(a.b.c)$

```
Book = { (B0) }
Name = { (N0), (N1), (N2) }
Addr = { (A0), (A1), (A2) }
Host = { (H0), (H1) }

myName = { (N1) }
myAddr = { (A0) }

address = { (B0, N0, A0), (B0, N1, A0), (B0, N2, A2) }
host = { (A0, H0), (A1, H1), (A2, H1) }

Book.address = { (N0, A0), (N1, A0), (N2, A2) }
Book.address[myName] = { (A0) }
Book.address.myName = {}

host[myAddr] = 
address.host = 
```

# logic: unary operators

---

$\sim$       *transpose*  
 $\wedge$       *transitive closure*  
 $*$       *reflexive transitive closure*  
*apply only to binary relations*

$$\begin{aligned}\wedge r &= r + r.r + r.r.r + \dots \\ *r &= \mathbf{idem} + \wedge r\end{aligned}$$

```
Node = { (N0), (N1), (N2), (N3) }
next = { (N0, N1), (N1, N2), (N2, N3) }

~next = { (N1, N0), (N2, N1), (N3, N2) }
^next = { (N0, N1), (N0, N2), (N0, N3),
          (N1, N2), (N1, N3),
          (N2, N3) }
*next = { (N0, N0), (N0, N1), (N0, N2), (N0, N3),
          (N1, N1), (N1, N2), (N1, N3),
          (N2, N2), (N2, N3), (N3, N3) }
```

```
first = { (N0) }
rest = { (N1), (N2), (N3) }

first.^next = rest
first.*next = Node
```

# logic: restriction and override

---

$<:$	<i>domain restriction</i>
$:>$	<i>range restriction</i>
$++$	<i>override</i>

$$p \ ++ \ q =$$
$$p \ - \ (\text{domain}(q) \ <: \ p) \ + \ q$$

```
Name      = { (N0), (N1), (N2) }
Alias      = { (N0), (N1) }
Addr       = { (A0) }
address    = { (N0, N1), (N1, N2), (N2, A0) }

address :> Addr = { (N2, A0) }
Alias <: address = address :> Name = { (N0, N1), (N1, N2) }
address :> Alias = { (N0, N1) }

workAddress = { (N0, N1), (N1, A0) }
address ++ workAddress = { (N0, N1), (N1, A0), (N2, A0) }
```

*The preferred for an alias, which is the workAddress if it exists, and otherwise the homeAddress*

# logic: boolean operators

---

!	<b>not</b>	<i>negation</i>
&&	<b>and</b>	<i>conjunction</i>
	<b>or</b>	<i>disjunction</i>
=>	<b>implies</b>	<i>implication</i>
,	<b>else</b>	<i>alternative</i>
<=>	<b>iff</b>	<i>bi-implication</i>

*four equivalent constraints:*

$F \Rightarrow G, H$

$F \text{ **implies** } G \text{ **else** } H$

$(F \ \&\& \ G) \ || \ ((!F) \ \&\& \ H)$

$(F \ \text{and} \ G) \ \text{or} \ ((\text{not } F) \ \text{and} \ H)$

# logic: quantifiers


---

```
all x: e | F
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F
```

```
all    F holds for every x in e
some  F holds for at least one x in e
no    F holds for no x in e
lone  F holds for at most one x in e
one   F holds for exactly one x in e
```

```
some n: Name, a: Address | a in n.address
some name maps to some address — address book not empty
```

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



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



```
all n: Name | no disj a, a': Address | (a + a') in n.address
```



# logic: quantified expressions

---

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

**some** Name  
*set of names is not empty*

**some** address  
*address book is not empty – it has a tuple*

**no** (address.Addr – Name)  
*nothing is mapped to addresses except names*

**all** n: Name | **lone** n.address  
*every name maps to at most one address*

# logic: comprehensions

---

$$\{x_1: e_1, x_2: e_2, \dots, x_n: e_n \mid F\}$$
$$\{n: \text{Name} \mid \mathbf{no} \ n.^{\wedge}\text{address} \ \& \ \text{Addr}\}$$

*set of names that don't resolve to any actual addresses*

$$\{n: \text{Name}, a: \text{Address} \mid n \rightarrow a \ \mathbf{in} \ ^{\wedge}\text{address}\}$$

*binary relation mapping names to reachable addresses*

# logic: if and let

---

```
if f then e1 else e2  
let x = e | constraint  
let x = e | expression
```

*four equivalent constraints:*

```
all n: Name |  
  some n.workAddress => n.address = n.workAddress  
  else n.address = n.homeAddress
```

```
all n: Name |  
  let w = n.workAddress, a = n.address |  
  some w => a = w else a = n.homeAddress
```

```
all n: Name |  
  let w = n.workAddress |  
  n.address = if some w then w else n.homeAddress
```

```
all n: Name |  
  n.address = let w = n.workAddress |  
  if some w then w else n.homeAddress
```

# logic: cardinalities

---

#r	<i>number of tuples in r</i>
0, 1, ...	<i>integer literal</i>
+	<i>plus</i>
-	<i>minus</i>

=	<i>equals</i>
<	<i>less than</i>
>	<i>greater than</i>
=<	<i>less than or equal to</i>
>=	<i>greater than or equal to</i>

**sum** x: e | ie

*sum of integer expression ie for all values of scalar x drawn from e*

**all** b: Bag | #b.marbles =< 3  
*all bags have 3 or less marbles*

#Marble = **sum** b: Bag | #b.marbles  
*the sum of the marbles across all bags  
equals the total number of marbles*

# 2 logics in one

---

- “everybody loves a winner”
- predicate logic
  - $\forall w \mid \text{Winner}(w) \Rightarrow \forall p \mid \text{Loves}(p, w)$
- relational calculus
  - $\text{Person} \times \text{Winner} \subseteq \text{loves}$
- Alloy logic – any way you want
  - **all** p: Person, w: Winner | p -> w **in** loves
  - Person -> Winner **in** loves
  - **all** p: Person | Winner **in** p.loves