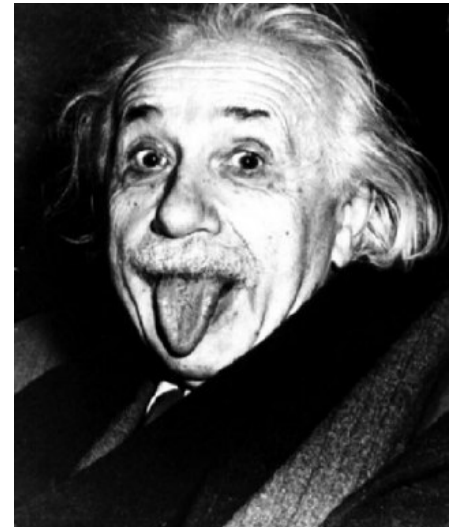


# **Alloy Language and Analysis**

based on slides from  
Greg Dennis and Rob Seater  
Software Design Group, MIT



# alloy language & analysis

---

- language = syntax for structuring specifications in logic
  - shorthands, puns, sugar
- analysis = tool for finding solutions to logical formulas
  - searches for and visualizes counterexamples



# “I'm My Own Grandpa” Song

---

- popular radio skit originally written in the 1930's
- expanded into hit song by “Lonzo and Oscar” in 1948



# “I'm My Own Grandpa” in Alloy

---

```
module grandpa

abstract sig Person {
  father: lone Man,
  mother: lone Woman
}

sig Man extends Person {
  wife: lone Woman
}

sig Woman extends Person {
  husband: lone Man
}

fact {
  no p: Person |
    p in p.^(mother + father)
  wife = ~husband
}
```

```
assert noSelfFather {
  no m: Man | m = m.father
}

check noSelfFather

fun grandpas[p: Person] : set Person {
  p.(mother + father).father
}

pred ownGrandpa[p: Person] {
  p in grandpas[p]
}

run ownGrandpa for 4 Person
```

# language: module header

---

```
module grandpa
```

- first non-comment of an Alloy model
- Modul and packages are the basis for modularization of big models and import for reuse

# language: signatures

---

**sig** A {}  
*set of atoms A*

**sig** A {}  
**sig** B {}  
*disjoint sets A and B (no A & B)*

**sig** B **extends** A {}  
*set B is a subset of A (B in A)*

**sig** B **extends** A {}  
**sig** C **extends** A {}  
*B and C are disjoint subsets of A  
(B in A && C in A && no B & C)  
but we can not infer that A = (B+C)*

**abstract sig** A {}  
**sig** B **extends** A {}  
**sig** C **extends** A {}  
*A partitioned by disjoint subsets B and C  
(no B & C && A = (B + C))*

**sig** B **in** A {}  
*B is a subset of A – not necessarily  
disjoint from any other set*

**sig** C **in** A + B {}  
*C is a subset of the union of A and B  
Elements of C are either from A **or** B*

**one sig** A {}  
**lone sig** B {}  
**some sig** C {}  
*A is a singleton set  
B is a singleton or empty  
C is a non-empty set*

# grandpa: signatures

---

```
abstract sig Person {  
  . . .  
}  
  
sig Man extends Person {  
  . . .  
}  
  
sig Woman extends Person {  
  . . .  
}
```

```
abstract sig Object {}  
sig File, Dir  
  extends Object {}  
sig Alias extends File {}  
sig Temp in Object {}
```

- all men and women are persons
- no person is both a man and a woman
- all persons are either men or women
- A file system whose objects are classified as files or directories, with aliases that are treated as files, and temporary objects, which may be files or directories

# Multiple Inheritance

---

- Is not directly supported but can be explicitly declared
- Example:  
“Jasmine tea is both flavored *and* a China tea”

```
sig Tea {}  
  
sig ChinaTea extends Tea {}  
  
sig FlavoredTea in Tea {}  
  
sig JasmineTea extends ChinaTea {}  
  
fact {JasmineTea in FlavoredTea}
```

# language: fields

---

- Fields declare relationship properties.
- No behavior can be declared

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

*f is a binary relation with domain A  
and range given by expression e  
f is constrained to be a function  
(f: A -> one e) or (all a: A | a.f: e)*

```
sig A {  
  f1: one e1,  
  f2: lone e2,  
  f3: some e3,  
  f4: set e4  
}
```

*(all a: A | a.fn : m e)*

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

*two fields with same constraints*

```
sig A {f: e1 m -> n e2}  
(f: A -> (e1 m -> n e2)) or  
(all a: A | a.f: e1 m -> n e2)
```

```
sig Book {  
  names: set Name,  
  addrs: names -> Addr  
}
```

*dependent fields*

*(all b: Book | b.addrs: b.names -> Addr)*

# grandpa: fields

---

```
abstract sig Person {  
  father: lone Man,  
  mother: lone Woman  
}  
  
sig Man extends Person {  
  wife: lone Woman  
}  
  
sig Woman extends Person {  
  husband: lone Man  
}
```

- fathers are men and everyone has at most one
- mothers are women and everyone has at most one
- wives are women and every man has at most one
- husbands are men and every woman has at most one

# Grouping fields

---

- Support sharing expressions

```
sig A {disj f, g: e}  
  implies the constraint  
  all this: A | no this.f & this.g
```

and

```
sig A {part f, g: e}  
  implies the same constraint, and additionally  
  all this: A | e in this.f + this.g
```

- A cat regards all cats that aren't friends as enemies

```
sig Cat {part friends, enemies: set Cat}
```

# language: facts

---

```
fact { F }  
fact f { F }  
sig S { ... } { F }
```

*facts introduce constraints that  
are assumed to always hold*

```
sig Host {}  
sig Link {from, to: Host}  
  
fact {all x: Link | x.from != x.to}  
no links from a host to itself  
  
fact noSelfLinks {all x: Link | x.from != x.to}  
same as above  
  
sig Link {from, to: Host} {from != to}  
same as above, with implicit 'this.'
```

# grandpa: fact

---

```
fact {  
  no p: Person |  
    p in p.^(mother + father)  
  wife = ~husband  
}
```

- Radio station with non-overlapping frequency bands???

- no person is his or her own ancestor
- a man's wife has that man as a husband  
“and the other direction”

```
sig RadioStation {  
  band: set Freq}  
fact NoOverlapping {  
  no disj s, s': RadioStation |  
    some s.band & s'.band  
}
```

# language: functions

---

```
fun f[x1: e1, ..., xn: en] : e { E }
```

*functions are named expression with declaration parameters and a declaration expression as a result invoked by providing an expression for each parameter*

```
sig Name, Addr {}  
sig Book {  
  addr: Name -> Addr  
}  
  
fun lookup[b: Book, n: Name] : set Addr {  
  b.addr[n]  
}  
  
fact everyNameMapped {  
  all b: Book, n: Name | some lookup[b, n]  
}
```

# language: predicates

---

```
pred p[x1: e1, ..., xn: en] { F }
```

*named formula with declaration parameters*

```
sig Name, Addr {}  
sig Book {  
  addr: Name -> Addr  
}  
  
pred contains[b: Book, n: Name, d: Addr] {  
  n->d in b.addr  
}  
  
fact everyNameMapped {  
  all b: Book, n: Name |  
    some d: Addr | contains[b, n, a]  
}
```

**Assumptions are placed in *fact* paragraphs constraints to be used in different contexts are packaged as *predicates*.**

# grandpa: function and predicate

---

```
fun grandpas[p: Person] : set Person {  
  p.(mother + father).father  
}  
  
pred ownGrandpa[p: Person] {  
  p in grandpas[p]  
}
```

- a person's grandpas are the fathers of one's own mother and father

# language: “receiver” syntax

---

```
fun f[x: X, y: Y, ...] : Z {...x...}  
fun X.f[y:Y, ...] : Z {...this...}
```

```
f[x, y, ...]  
x.f[y, ...]
```

```
pred p[x: X, y: Y, ...] {...x...}  
pred X.p[y:Y, ...] {...this...}
```

```
p[x, y, ...]  
x.p[y, ...]
```

```
fun Person.grandpas : set Person {  
  this.(mother + father).father  
}  
  
pred Person.ownGrandpa {  
  this in this.grandpas  
}
```



# language: assertions

---

```
assert a { F }
```

*constraint intended to follow  
from facts of the model*



```
sig Node {  
  children: set Node  
}  
  
one sig Root extends Node {}  
  
fact {  
  Node in Root.*children  
}  
  
// invalid assertion:  
assert someParent {  
  all n: Node | some children.n  
}  
  
// valid assertion:  
assert someParent {  
  all n: Node - Root | some children.n  
}
```

# Remember

---

- The constraints of a model are organized into *paragraphs*.
- Assumptions are placed in *fact* paragraphs;
- Implications to be checked are placed in *assertions*;
- Constraints to be used in different contexts are packaged as *predicates*;
- Reusable expressions are packaged as *functions*.

# language: check command

---

```
assert a { F }  
check a scope
```

*instructs analyzer to search for  
counterexample to assertion within scope*

*if model has facts M  
finds solution to M && !F*

```
check a  
top-level sigs bound by 3
```

```
check a for default  
top-level sigs bound by default
```

```
check a for default but list  
default overridden by bounds in list
```

```
check a for list  
sigs bound in list,  
invalid if any unbound
```

```
abstract sig Person {}  
sig Man extends Person {}  
sig Woman extends Person {}  
sig Grandpa extends Man {}
```

```
check a  
check a for 4  
check a for 4 but 3 Woman  
check a for 4 but 3 Man, 5 Woman  
check a for 4 Person  
check a for 4 Person, 3 Woman  
check a for 3 Man, 4 Woman  
check a for 3 Man, 4 Woman, 2 Grandpa
```

```
// invalid:  
check a for 3 Man  
check a for 5 Woman, 2 Grandpa
```

# grandpa: assertion check

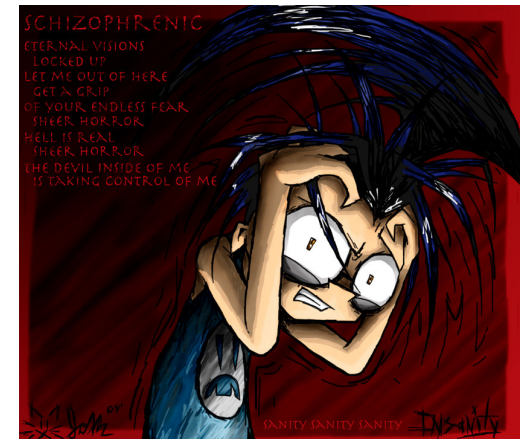
---

```
fact {
  no p: Person | p in p.^(mother + father)
  wife = ~husband
}

assert noSelfFather {
  no m: Man | m = m.father
}

check noSelfFather
```

- sanity check
- command instructs analyzer to search for counterexample to *noSelfFather* within a scope of at most 3 *Persons*
- *noSelfFather* assertion follows from fact



# language: run command

---

```
pred p[x: X, y: Y, ...] { F }  
run p scope
```

*instructs analyzer to search for  
instance of predicate within scope*

*if model has facts M, finds solution to  
M && (some x: X, y: Y, ... | F)*



```
fun f[x: X, y: Y, ...] : R { E }  
run f scope
```

*instructs analyzer to search for  
instance of function within scope*

*if model has facts M, finds solution to  
M && (some x: X, y: Y, ..., result: R | result = E)*

# grandpa: predicate simulation

---

```
fun grandpas[p: Person] : set Person {
  p.(mother + father).father
}

pred ownGrandpa[p: Person] {
  p in grandpas[p]
}

run ownGrandpa for 4 Person
```

- command instructs analyzer to search for configuration with at most 4 people in which a man is his own grandfather

# language: modules

---

- Modules correspond one-to-one with files.
- Every analysis is applied to a single module; any other modules containing relevant model fragments must be explicitly imported.

graphs.als

```
module library/graphs
pred Acyclic (r: univ -> univ) {no ^r & iden}
```

filesSystem.als

```
module filesystem
open library/graphs
sig Object {}
sig Directory extends Object {contents: set Object}
fact {Acyclic (contents)}
```

# language: parametric modules

---

- A module can be *parameterized* by one or more signature parameters, given as a list of identifiers in brackets after the module name

```
module library/graphs[t]
pred Acyclic (r: t -> t) {no ^r & iden}
```

and two modules that use it

```
module family
open library/graphs[Person]

sig Person {parents: set Person}
fact {Acyclic (parents)}
```

```
module fileSystem
open library/graphs [Object]

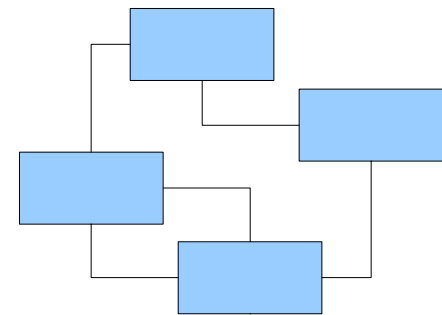
sig Object {}

sig Directory extends Object
{contents: set Object}
fact {Acyclic (contents)}
```

# Static models

---

- static models
  - describes states, not behaviors
  - properties are *invariants*
  - e.g. that a list is sorted



# pattern: definition

---

- define a new term using existing terms
  - declare new relation and constrain to existing relations
  - constraint often written as equality, e.g.

```
sig Person {  
  spouse: lone Person,  
  parents: set Person,  
  inlaws: set Person  
}  
fact { inlaws = spouse.parents }
```

# pattern: multirelation

---

- use higher-arity relation to model relationship between more than two entities
- address book example:

```
sig Book {  
  addrs: Name -> Addr  
}
```



# pattern: singleton

---

- particular elements of set play important roles
- use **one** multiplicity to make a singleton sig

```
one sig Root extends Directory {}
```



# Visualization

---

- Download *grandpa.als* from the tutorial website
- Click “Execute”
- Click “Show”
- Click “Theme”

