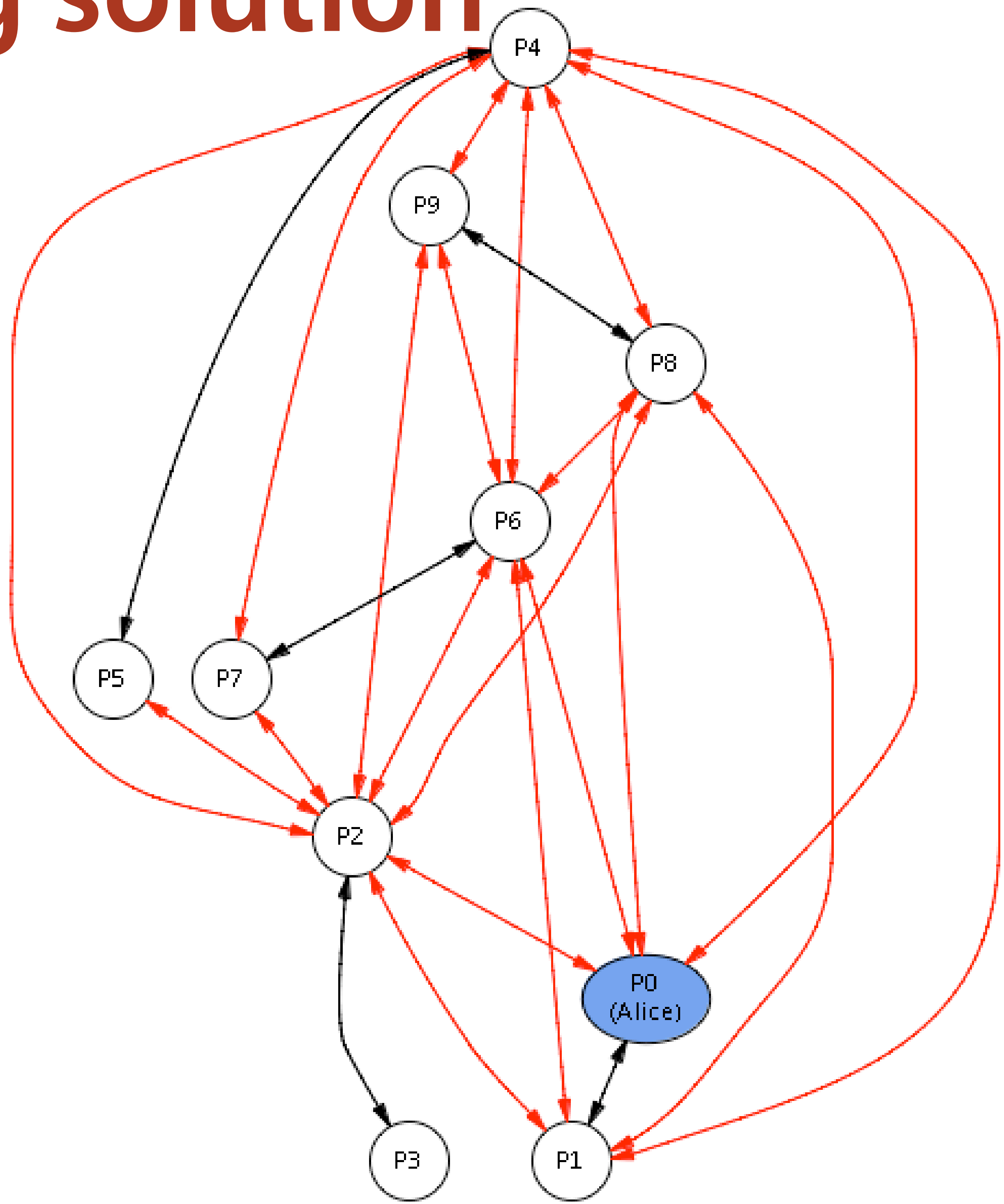
LOGIC & LANGUAGE

Daniel Jackson · Lipari Summer School · July 18-22, 2005

handshaking solution



what this lecture's about

a relational logic

› first-order logic + relational operators

Alloy language

› signatures & fields

› constraint packaging

mostly review? but …

› generalized join

› sets & scalars as relations

› first-order puns

introduction

why logic?

simplicity

› close to phenomena being described

› familiar syntax & semantics

one language

› for system & properties

› model checking research:
    focus on property language; machine language ignored

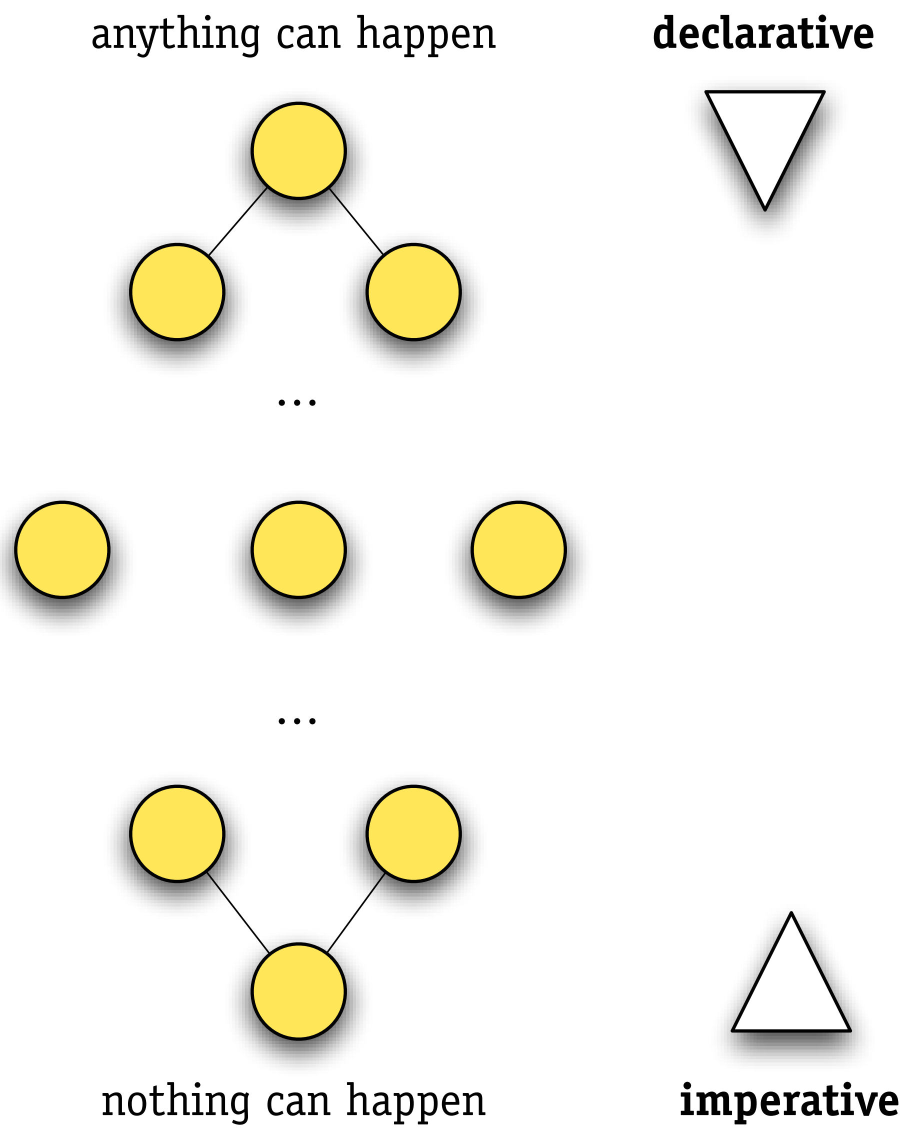
declarative

› the more you add, the less happens

› so good for partial descriptions (esp. environment)

› and good for incremental modelling

imperative vs. declarative



why relations?

simplest way to talk about structure?

› just like references in OOP

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

-- David Wheeler

Roger Needham & David Wheeler
explain Cambridge Ring to chancellor of Cambridge University

3 logics

"everybody loves a winner"

predicate logic

$\forall w \mid \text{Winner}(w) \Rightarrow \forall p \mid \text{Loves}(p,w)$

relational calculus

$\text{Person} \, ; \, \text{Winner}^{-1} \subseteq \text{loves}$

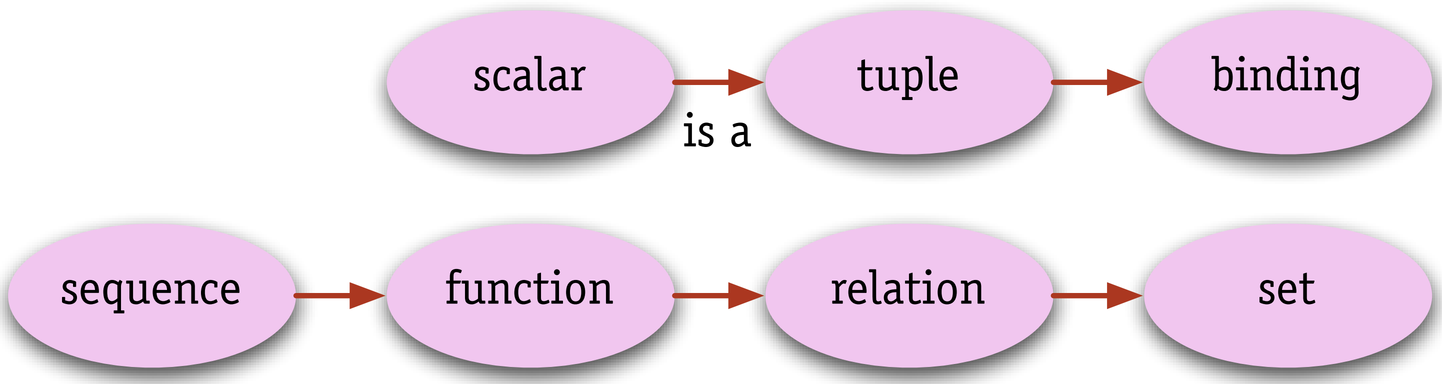
my relational logic

```
all p: Person, w: Winner | p->w in loves
Person -> Winner in loves
all p: Person | Winner in p.loves
```

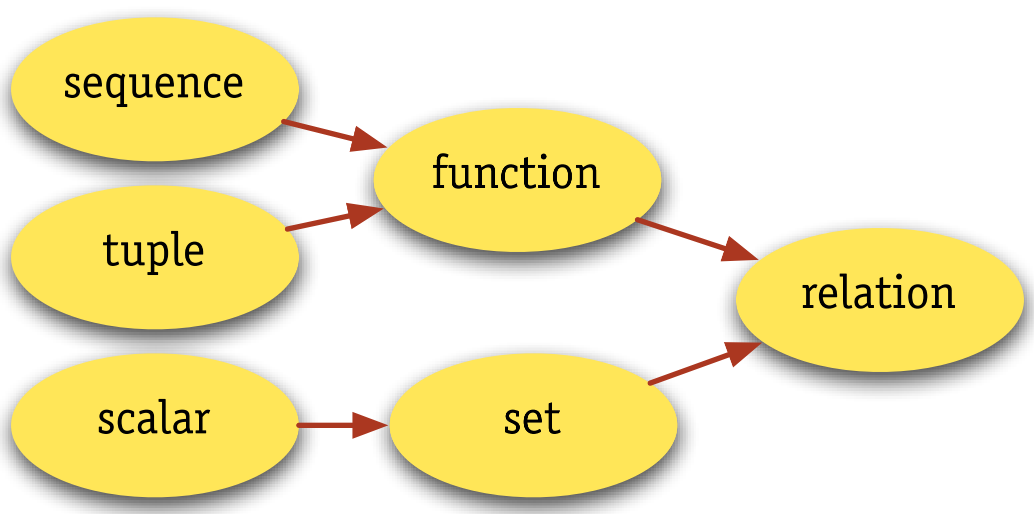
semantic basis: relations

everything's a relation

Z based on ZF set theory



Alloy based on relational calculus



atoms & relations

atoms are individuals that are

- indivisible

  can't be broken into smaller parts
- immutable

  don't change over time
- uninterpreted

  no built-in properties

a **relation** is a table

- set of tuples of atoms
- arity = number of columns, ≥ 1
- size = number of rows, ≥ 0

examples

Winner

alice

scalar

Person

alice
bob
carol

set

loves

alice	alice
bob	alice
carol	alice

relation

in standard set theory: scalar a, tuple (a), set {a}, relation {(a)}

in Alloy logic: scalar, tuple, set, relation {(a)}

typical kinds of relation

containment

msgs: Buffer -> Message

grouping

group: Graphic -> Group
sameGroup: Graphic -> Graphic

indirection

style: Paragraph -> Style

naming

addr: Alias -> Address

ordering

next: Time -> Time

constants, ops, quantifiers: a lightning tour

constants

universal	**univ**	{(x) | x is an atom}
identity	**iden**	{(x,x) | x ∈ **univ**}
empty	**none**	{}

set operators

union	p + q	$\{t \mid t \in p \lor t \in q\}$
difference	p - q	$\{t \mid t \in p \land t \notin q\}$
intersection	p & q	$\{t \mid t \in p \land t \in q\}$
subset	p in q, p : q	$\{(p_1, \ldots p_n) \in p\} \subseteq \{(q_1, \ldots q_n) \in q\}$
equality	p = q	$\{(p_1, \ldots p_n) \in p\} = \{(q_1, \ldots q_n) \in q\}$

File + Dir, Object - Dir, Open & File
File in Object
Object = File + Dir + Alias
brother + sister
sister in sibling
root: Dir

arrow product

$p \rightarrow q$ $\{(p_1, \ldots p_n, q_1, \ldots q_m) \mid (p_1, \ldots p_n) \in p \wedge (q_1, \ldots q_m) \in q\}$

```
alice -> bob
Person -> Winner
univ -> univ
alice -> bob in loves
f -> root in dir
dir: Object -> Dir
```

arrow idioms

when `s` and `t` are sets

- `s -> t` is their cartesian product
- `r: s -> t` says `r` maps atoms in `s` to atoms in `t`

when `x` and `y` are scalars

- `x -> y` is a tuple

dot & box join

$p . q \; \{(p_1, \ldots p_{n-1}, q_2, \ldots q_m) \mid (p_1, \ldots p_n) \in p \land (p_n, q_2, \ldots q_m) \in q\}$

alice	{(ALICE)}
loves	{(ALICE, BOB}, (ALICE, CAROL},(CAROL, ALICE)}
alice.loves	{(ALICE, CAROL})}
loves.alice	{(CAROL})}
loves.loves	{(ALICE, ALICE})}

p.expr [q] = q.(p.expr)

loves.loves[alice] {(ALICE})}

join idioms

when p and q are binary relations

- p.q is standard relational composition

when r is a binary relation and s is a set

- s.r is relational image of s under r ('navigation')
  univ.r is the range of r
- r.s is relational image of s under ~r ('backwards navigation')
  r.univ is the domain of r

when f is a function and x is a scalar

- x.f is application of f to x

what is x.f when x outside domain of f?

the partial function tarpit

Romeo's wife is Juliette or Romeo is unmarried

romeo.wife = juliette or romeo in Unmarried

› true if Romeo has no wife?

approaches

› 3-valued logic (eg, VDM, OCL) [complex, no congruence]

    maybe or true

› all functions total (eg, Larch) [function not just a relation]

    ? = juliette or true

› undefined values (eg, OCL) [strictness]

    undefined = juliette or true

› partial semantics (eg, Z) [complex, no congruence]

    ? or true ⇛ ?

› bad applications are false (eg, Parnas) [$x \neq y$, $\neg x = y$ differ]

    false or true

joins on multirelations

given a relation on books/aliases/addresses

addr	{(B0,A0,D0), (B0,A1,D1), (B1,A1,D2), (B1,A2,D3)}
b	{(B0)}
a	{(A0)}
d	{(D3)}

we have

b.addr	{(A0,D0), (A1,D1)}
b.addr[a]	{(D0)}
addr.d.univ	{(B1)}

playing with the analyzer

other handy operators

transpose	~p	$\{(p_n, \ldots p_1) \mid (p_1, \ldots p_n) \in p\}$
transitive closure	^p	smallest $q \mid q.q \subseteq q \land p \subseteq q$
restriction	s <: p	$\{(p_1, \ldots p_n) \mid (p_1, \ldots p_n) \in p \land p_1 \in s\}$
domain	dom p	$\{(p_1) \mid (p_1, \ldots p_n) \in p\}$
override	p ++ q	q + (p - dom q <: p)

quantifiers & cardinalities

quantifiers

all, **some**, **no**, **one**, **lone**

quantified formulas

all x: e | F $\bigwedge_{v \in x}$ F [{(v)}/x]

cardinality expressions

#e	size of relation e
no e	#e = 0
some e	#e > 0
lone e	#e ≤ 1
one e	#e = 1

declarations & multiplicity

multiplicity keywords: **some**, **one**, **lone**, **set**

set declarations

s: ***m*** e   s ⊆ e ∧ ***m*** e

s: e   s: **one** e

relation declarations

r: e ***m*** -> ***n*** e′   r ⊆ e × e′ ∧ ∀x: e | ***n*** x.r ∧ ∀x: e′ | ***m*** r.x

examples

alice: Winner

Winner: **set** Person

loves: Person **some** -> **some** Person

root: Dir

dir: (Object - root) -> **one** Dir

puns

to support familiar declaration syntax

- Alloy declaration `r: A -> B`
- has traditional reading $r \in 2^{(A \times B)}$
- has Alloy reading $r \subseteq A \times B$

to support ‘navigation expressions’

- Alloy expression `x.f.g`
- has traditional reading g(f(x)) unless f(x) undefined or a set
- has Alloy reading image (image({(x)}, f), g)

summary of features

simple syntax

- same operators for sets, scalars, relations
- no lifting `{x}`
- conventional, but puns

simple semantics

- no undefined expressions: all operators total
- two-valued logic

why does this work?

- first order: no sets of sets needed
- no boolean *expressions*

language

structure of an alloy model

signatures & fields
› introduces sets and relations
› 'extends' hierarchy for classification & subtypes

constraints paragraphs
› facts: assumed to hold
› predicates: reusable constraints
› functions: reusable expressions
› assertions: conjectures to check

commands
› run: generate instances of a predicate
› check: generate counterexamples to an assertion

instances

alloy analyzer is a **model finder**

- finds solutions to constraints
- run predicate: solution is instance
- check assertion: solution is counterexample

solution

- assignment of relational values to variables
- variables are
  - sets (signatures)
  - relations (fields)
  - skolem constants (witnesses, predicate arguments)

a model

```
module examples/addressBook/addLocal

abstract sig Target {}
sig Addr extends Target {}
sig Name extends Target {}
sig Book {addr: Name -> Target}

fact Acyclic {all b: Book | no n: Name | n in n.^(b.addr)}
fun lookup (b: Book, n: Name): set Addr {n.^(b.addr) & Addr}
pred add (b, b': Book, n: Name, t: Target) {b'.addr = b.addr + n->t}
run add for 3 but 2 Book

assert addLocal {
  all b,b': Book, n,n': Name, a: Addr |
     add (b,b',n,a) and n != n' => lookup (b,n') = lookup (b',n') }
check addLocal for 3 but 2 Book
```

declarations

```
module examples/addressBook/addLocal

abstract sig Target {}
sig Addr extends Target {}
sig Name extends Target {}
abstract: Target in Addr + Name
extends: Addr in Target and Name in Target and no Addr & Name

sig Book {addr: Name -> Target}
addr: Book -> Name -> Target
```

variables

```
module examples/addressBook/addLocal

abstract sig Target {}
sig Addr extends Target {}
sig Name extends Target {}
sig Book {addr: Name -> Target}

fact Acyclic {all b: Book | no n: Name | n in n.^(b.addr)}
fun lookup (b: Book, n: Name): set Addr {n.^(b.addr) & Addr}
pred add (b, b': Book, n: Name, t: Target) {b'.addr = b.addr + n->t}

run add for 3 but 2 Book
```

negating an assertion

```
assert addLocal {
   all b,b': Book, n,n': Name, a: Addr |
      add (b,b',n,a) and n != n' => lookup (b,n') = lookup (b',n') }
check addLocal for 3 but 2 Book
```

for analysis, equivalent to these:

```
pred addLocal () {
   some b,b': Book, n,n': Name, a: Addr |
      add (b,b',n,a) and n != n' and not lookup (b,n') = lookup (b',n') }
run addLocal for 3 but 2 Book
```

```
pred addLocal (b,b': Book, n,n': Name, a: Addr) {
      add (b,b',n,a) and n != n' and not lookup (b,n') = lookup (b',n') }
run addLocal for 3 but 2 Book
```

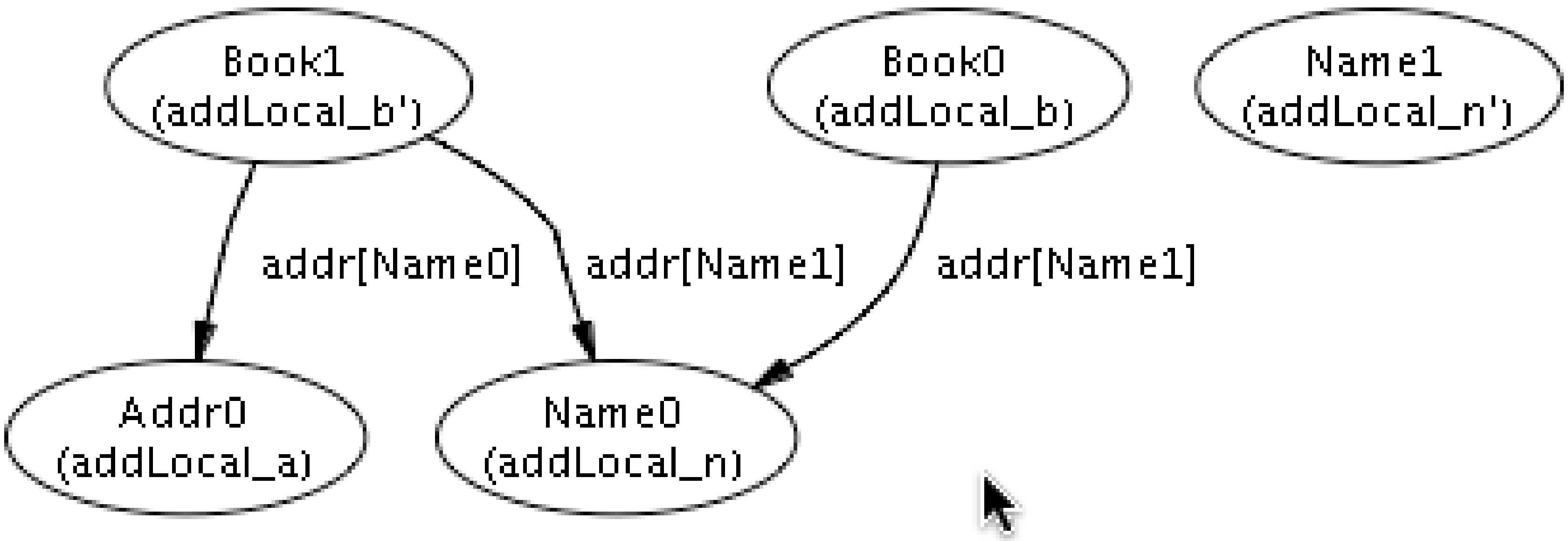
counterexample: textual

```
module examples/addressBook/addLocal
sig Target extends univ = {Addr_0, Name_0, Name_1}
sig Addr extends Target = {Addr_0}
sig Name extends Target = {Name_0, Name_1}
sig Book extends univ = {Book_0, Book_1}
 addr: {Book_0 -> Name_1 -> Name_0,
    Book_1 -> {Name_0 -> Addr_0, Name_1 -> Name_0}}

skolem constants
addLocal_b = {Book_0}
addLocal_b' = {Book_1}
addLocal_n = {Name_0}
addLocal_n' = {Name_1}
addLocal_a = {Addr_0}
```

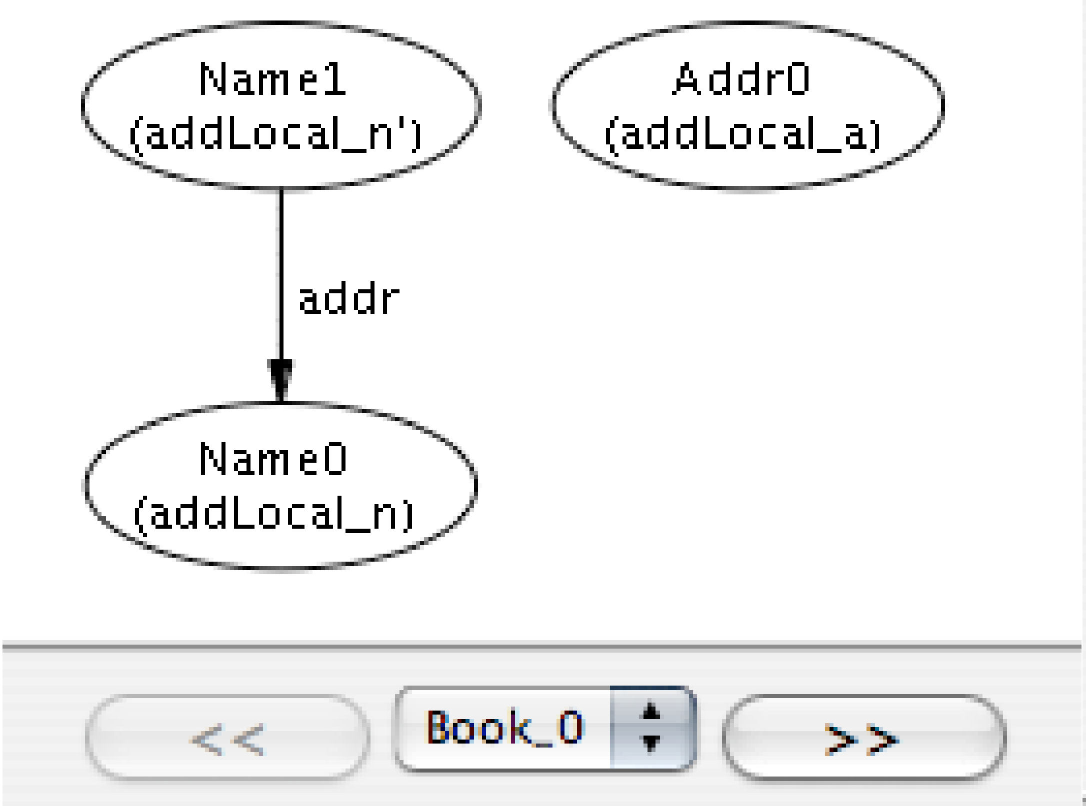
counterexample: graphical

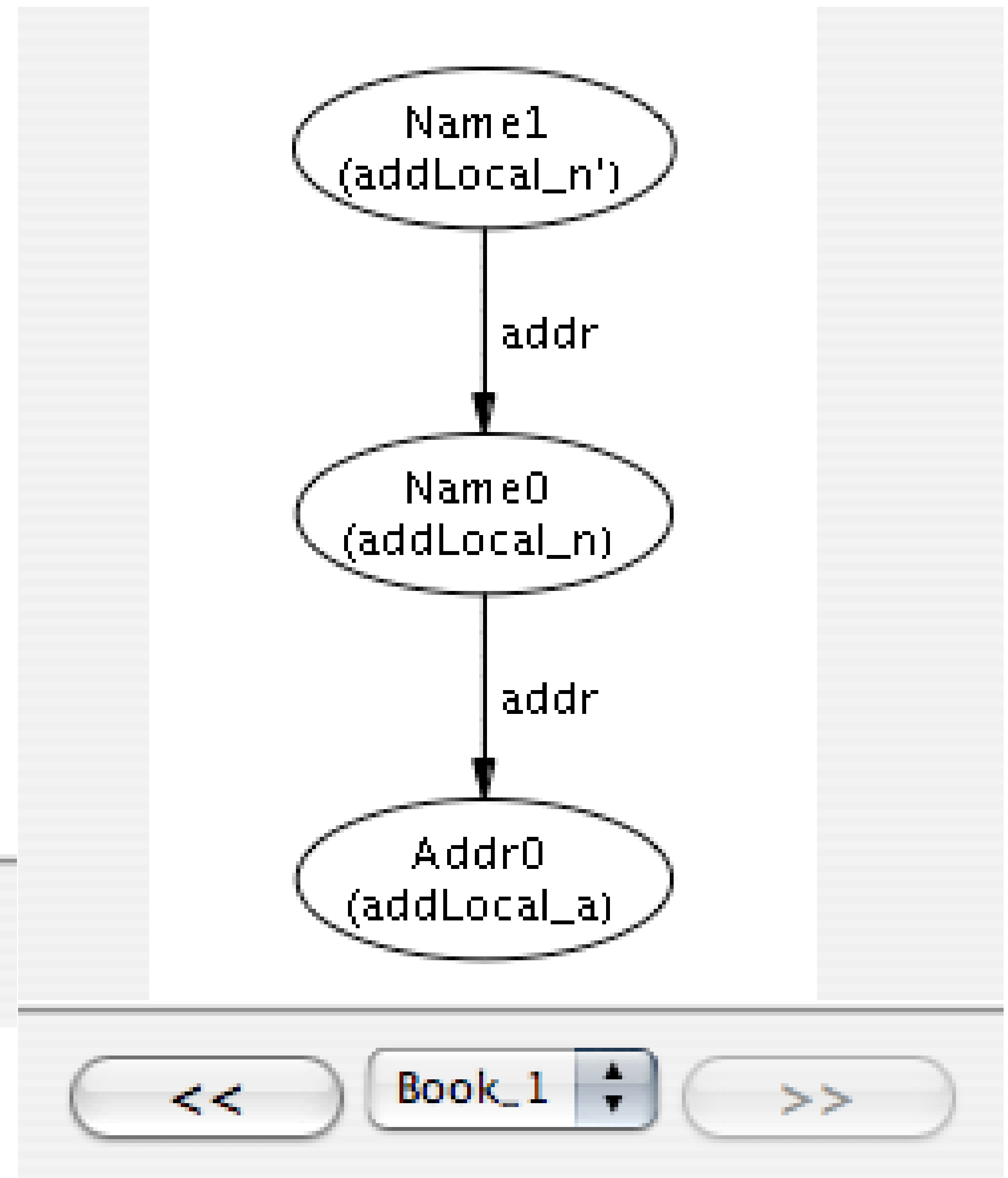
without customization



counterexample: graphical

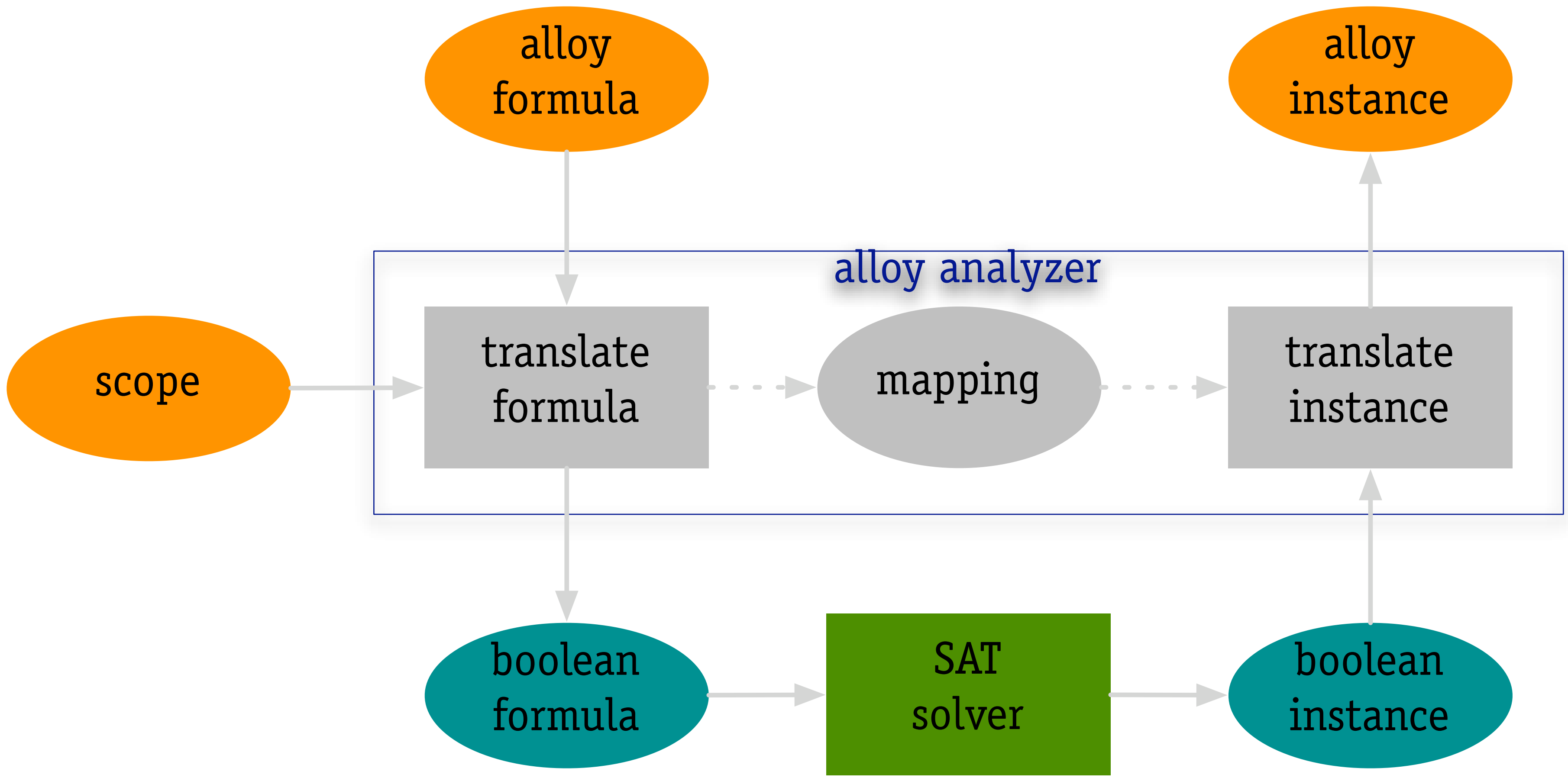
with Book projected





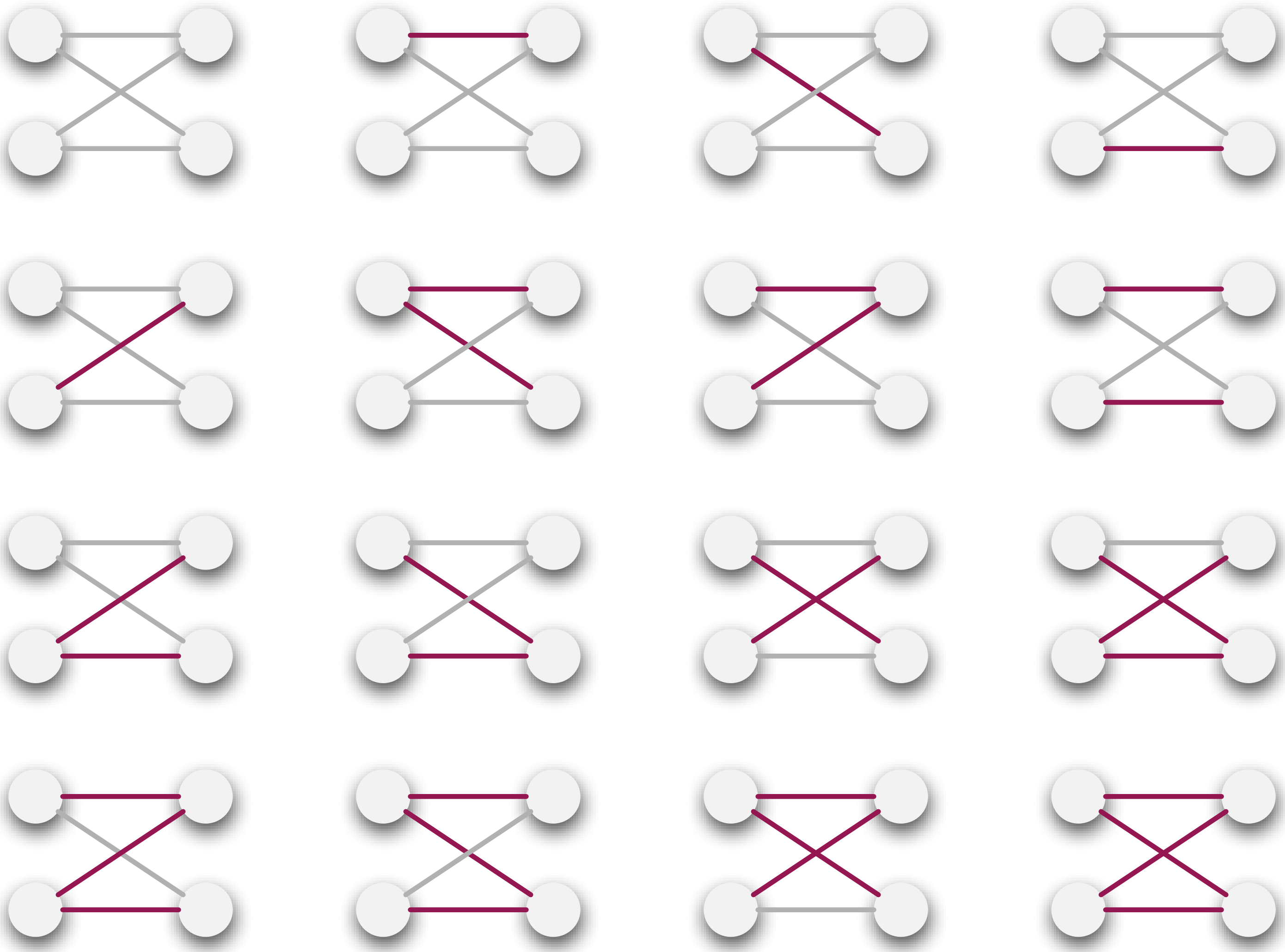
how the analysis works

alloy analyzer architecture

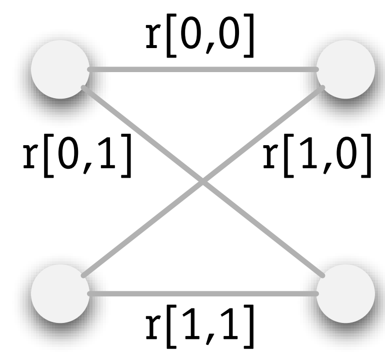


relational values

space of values for relation in scope 2



space of values as
4 boolean vars



sample translation

example

b, b′: Book

n: Name

names: Book -> Name

b′.names = b.names + n

compositional translation

b [i] : true if ith element of Book is in the set b

names [i, j] : true if names maps ith element of Book to jth element of Name

$b.names + n\ [i] = (\exists k.\ b\ [k] \wedge names\ [k, i]) \vee n\ [i]$

$b'.names = b.names + n =$

$$\forall i.\ (\exists k.\ b'\ [k] \wedge names\ [k, i]) \Leftrightarrow (\exists k.\ b\ [k] \wedge names\ [k, i]) \vee n\ [i]$$

quantification

grounding out

all x: t | F

becomes: F [x0/x] **and** F [x0/x] **and** ...

skolemization

some x: t | F

becomes: F [X/x] where X is a fresh free variable

all x: s | **some** y: t | F

becomes: **all** x: s | F [x.Y/y] where Y: s-> t is a free (function) var

optimizations

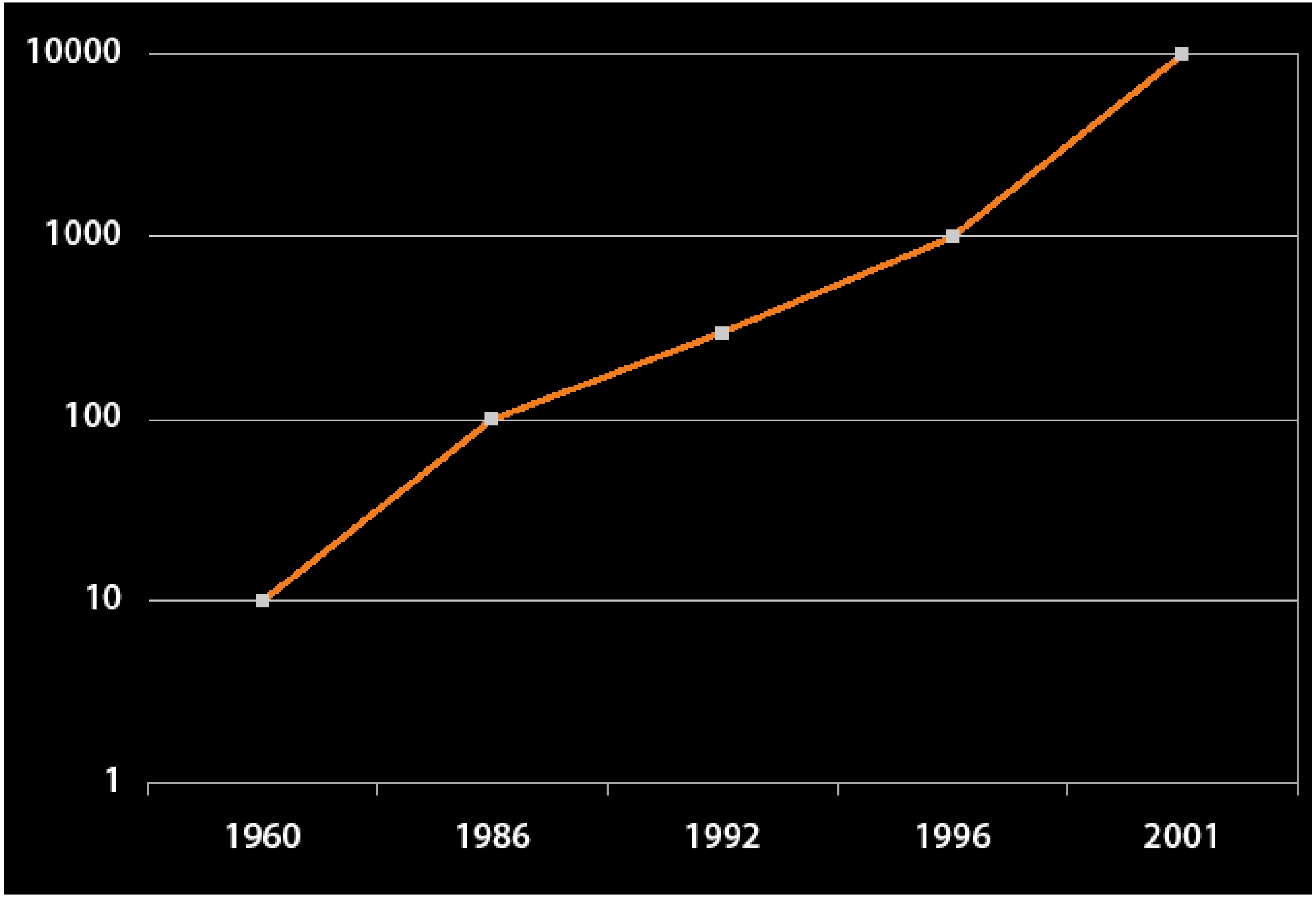
symmetry

- atoms of a signature are interchangeable
- so adds symmetry breaking predicates automatically
- for `util/ordering`, ordering is fixed `A1, A2, A3, ...`

other optimizations

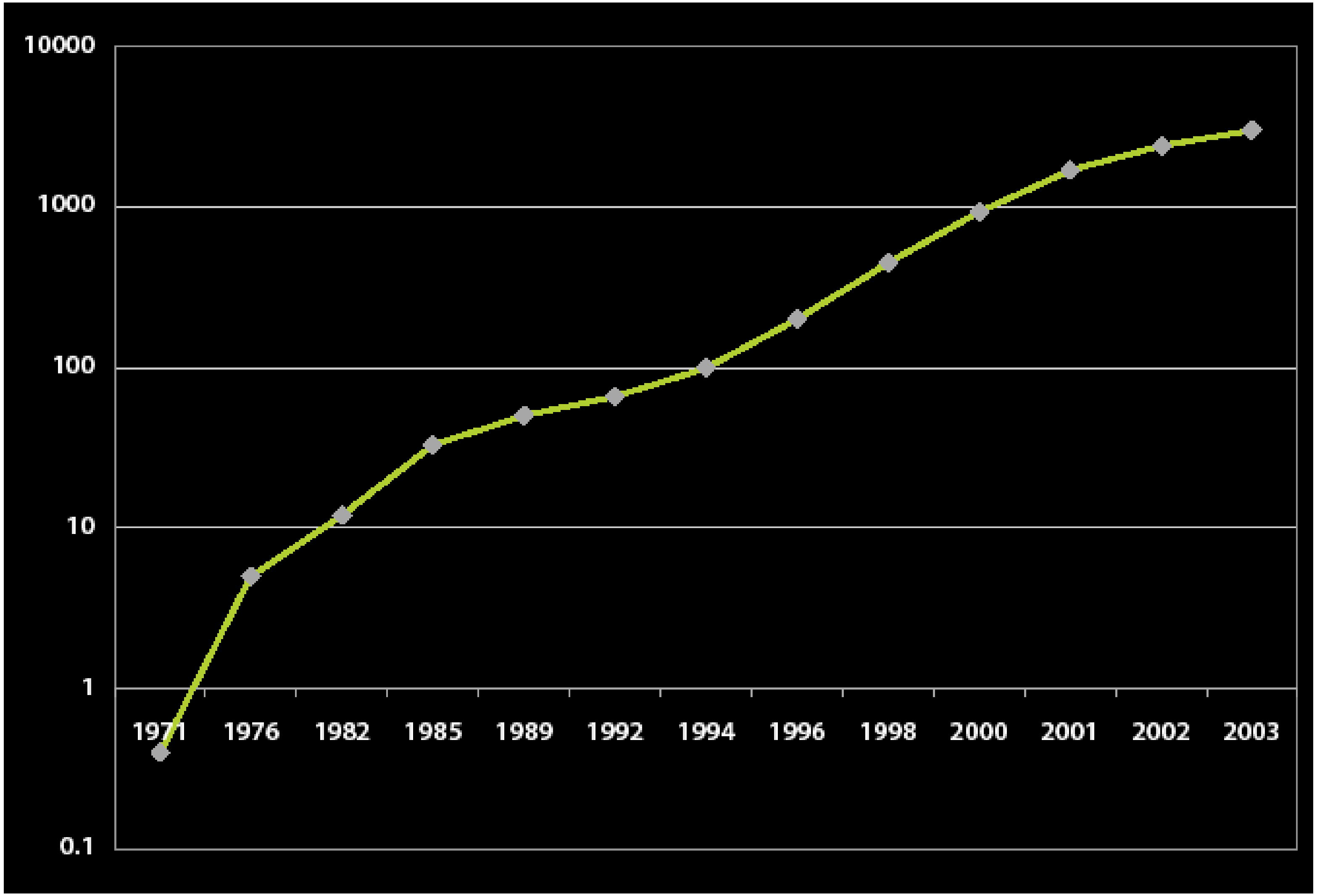
- sharing: detecting shared formulas before they appear during grounding out
- boolean simplifications
- careful conversion to CNF
- 'atomization' using subtypes

performance: SAT solvers



size of solvable constraint in #boolean variables
from Sharad Malik

performance: moore's law



speed of main processor offering in MHz

from intel.com

homework

homework: self-grandpa

I'M MY OWN GRANDPA (Dwight Latham & Moe Jaffe)

Many, many years ago when I was twenty-three
I was married to a widow who was pretty as could be …

… Now if my wife is my grandmother, then I'm her grandchild
And every time I think of it, it nearly drives me wild
‘Cause now I have become the strangest case you ever saw
As husband of my grandmother, I am my own grandpa

I’m my own grandpa,
I’m my own grandpa,
It sounds funny, I know
But it really is so
I’m my own grandpa

self-grandpa in alloy

```
module examples/grandpa/grandpa1

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
   }

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

pred ownGrandpa (p: Person) {p in grandpas (p)}
run ownGrandpa for 4 Person
```

your task

find a solution to the song by

- changing the expression in the function `grandpa`
- adding any new constraints that seem necessary

free upgrades available!

new memory sticks for old!

ask me if you'd like:

› yesterday's address book examples
› today's updated lecture and examples