دانشگاه آزاد اسلامی واحد تبریز

نام درس: روش های رسمی در مهندسی نرم افزار

بخش: زبان برنامه نویسی Alloy

نام استاد: دکتر مسعود کارگر

CONTENTS

Signatures

A signature expresses a new type in your spec. It can be anything you want. Here are some example signatures:

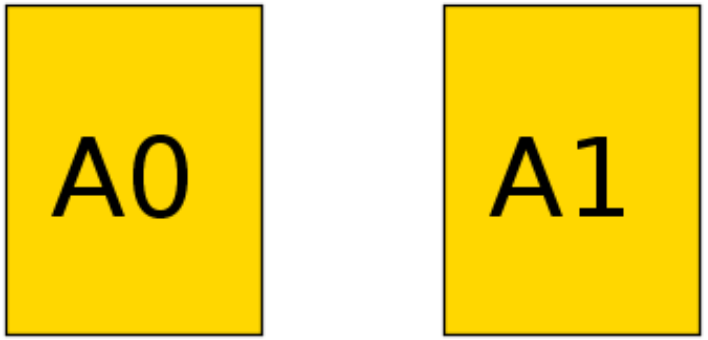
Time
State
File
Person
Msg
Pair

Alloy can generate models that have elements of each signature, called atoms. Take the following spec:

```
sig A {}
```

Signatures

The following would be an example generated model:



Signatures

Here we have two atoms A$1 and A$0. Both count as instances of the A signature. See visualizer for more on how to read the visualizations.

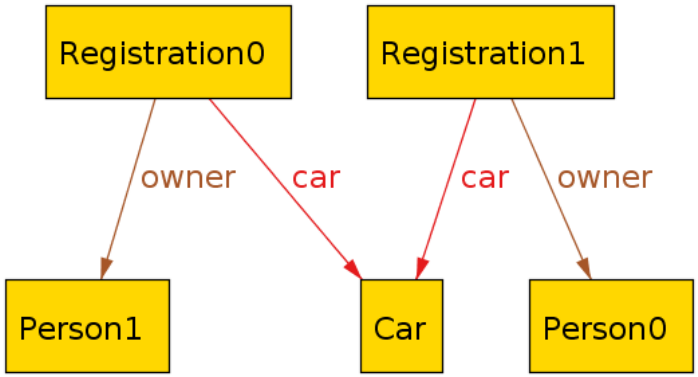
Usually we care about the relationships between the parts of our systems. We don't just care that there are people and cars, we care which people have which cars. We do this by adding relations inside of the signature body.

```
sig Person {}
sig Car {}

sig Registration {
  owner: Person,
  car: Car
}
```

Signatures

This defines a new Registration type, where each Registration has an owner, which is a Person, and a car, which is a Car. The comma is required.



Tip

Extra commas are ignored. So you can write Registration instead like this:

```
sig Registration {
   , owner: Person
   , car: Car
}
```

Relations

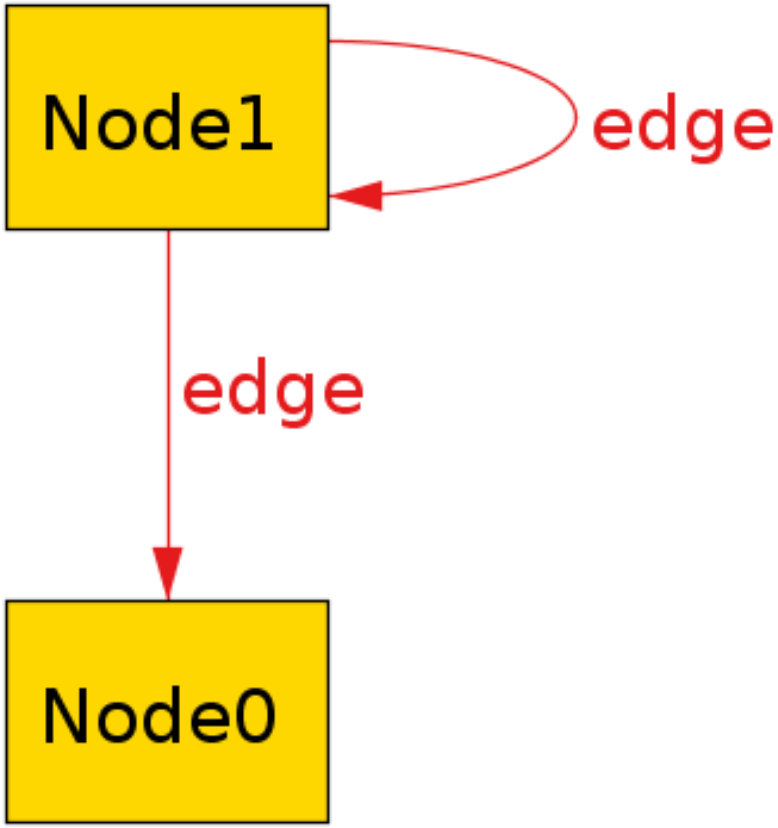
The body of a signature is a list of relations, which show how the signatures are connected to each other. **A relation in the body of a signature is also called a field.**

Relations are separated by a comma. The list can also start and end with a comma. Relations do not have to be on separate lines, as long as they are separated by commas.

Relations can refer to the same signature. This is valid:

```
sig Node {
  , edges: set Node
}
```

Relations



Relations

Alloy can generate models where a relation points from an atom to itself, aka a “self-loop”. For this reason we often want to add contraints to our model, such as Facts or Predicates.

Note
Each relation in the body of a signature actually represents a relation type. If we have:

sig A {r: one B}
Then r is set of relations in A -> B. See Sets and Relations for more information.

Different signatures may have relationships with the same name as long as the relationship is not ambiguous.

Multiplicity

Each relation has a multiplicity, which represents how many atoms it can include. If you do not include a multiplicity, it's assumed to be one for individual relations and set for Multi-relations.

one

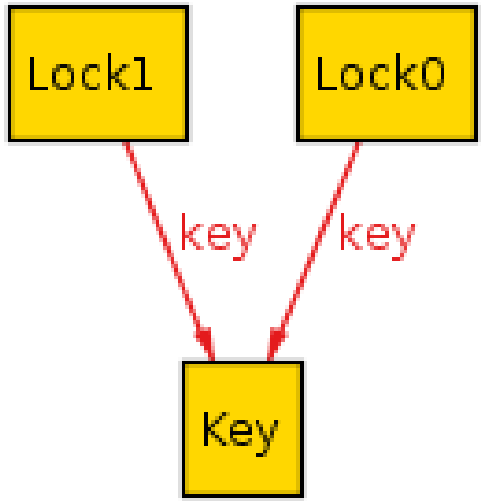
The default. r: one A states that there is exactly one A in the set.

```
sig Key {}

sig Lock {
  , key: one Key
}
```

Multiplicity

This says that every lock has exactly one Key. This does not guarantee a **1-1** correspondence! Two locks can share the same key.



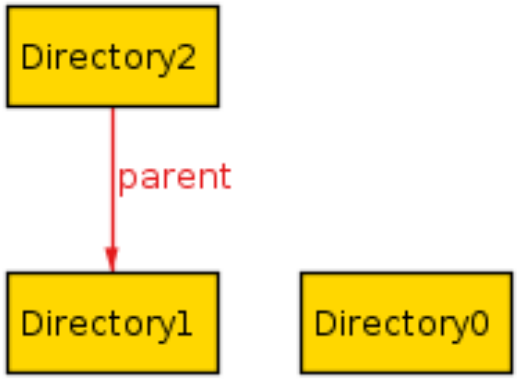
If no multiplicity is listed, Alloy assumes to be one. So the above relation can also be written as key: Key.

Multiplicity

lone
r: lone A states that either there is one A in the set, **or that the set is empty.**
You can also think of it as “optional”.

```
sig Directory {
  , parent: lone Directory
}
```

This says that every directory either has one parent, or it does not have a parent (it’s a root directory).

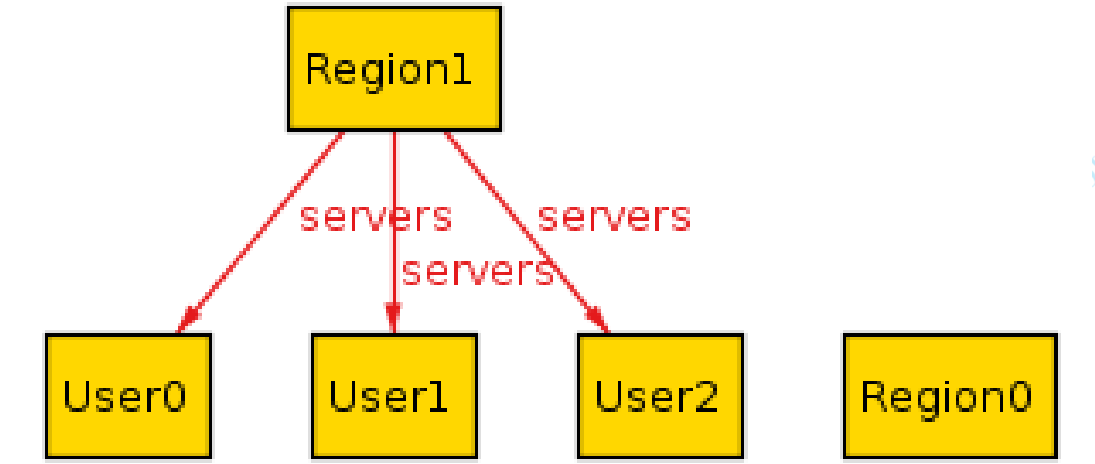


Multiplicity

set

r: set A states that there can be **any number** of A in the relation.

```
sig User {}
sig Region {
  servers: set User
}
```

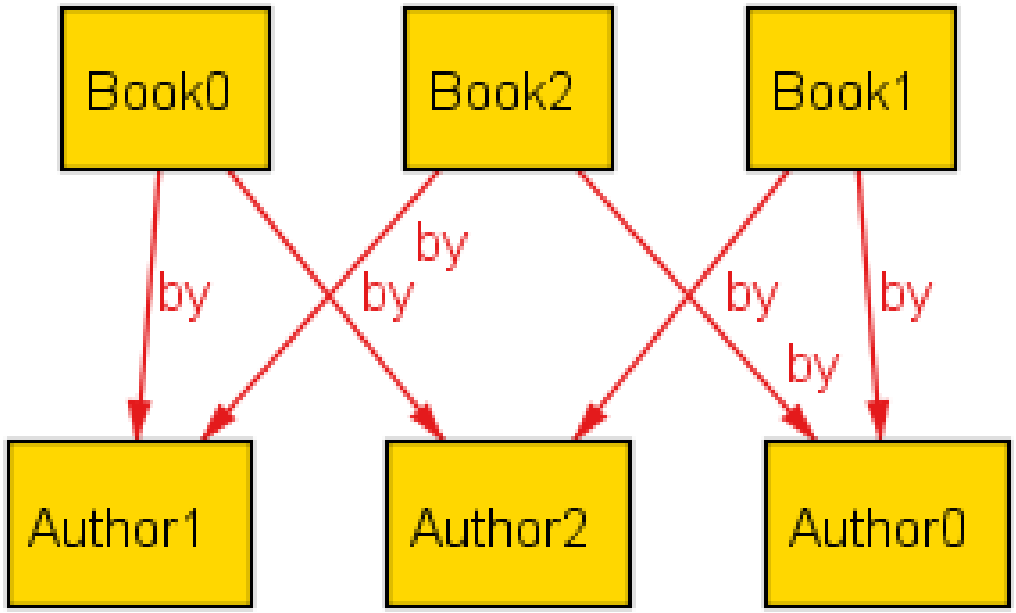


Multiplicity

some
r: some A states that there is at least one A in the relation.

```
sig Author {}

sig Book {
  by: some Author
}
```



Multiplicity

disj

disj can be prepended to any multiplicity to guarantee that it will be disjoint among all atoms. If we write

```
sig Lock {}
sig Key {
  lock: disj one Lock
}
```

Then every key will correspond to a different lock. If we instead write

```
sig Lock {}
sig Key {
  locks: disj some Lock
}
```

Then every key will correspond to one or more locks, but no two keys will share a lock.

Field Expressions

A field can be a simple expression over other signatures.

```
sig Resource {
  permissions: set (User + Group)
}
```

In addition to full signatures, the expression may contain this, which refers to the specific atom itself.

```
sig Node {
  -- no self loops
  , edges: set Node - this
}
```

Field Expressions

A dependent field is one where the expression depends on the values of other fields in the atom. The dependencies must be fields defined either in the signature or its supertype.

```
sig Item {}

sig Person {
  , favorite: Item
  , second: Item - favorite
}
```

Multi-relations

Signatures can have multi-relations as fields:

```
sig Door {}
sig Card {}

sig Person {
  access: Card -> Door
}
```

In this case access is a ternary relationship, where each element of access is a relation of form Person -> Card -> Door.

Multi-relations

Multi-relations have a special kind of multiplicity:

r: A m -> n B

This says that each member of A is mapped to n elements of B, and m elements of A map to each element of B. If not specified, the multiplicities are assumed to be set.

Multi-relations

As an aid, use the following table:

m	n	Meaning
set	set	No restrictions
set	some	Each A used at least once
set	one	Each A is mapped to exactly one B (total function)
set	lone	Each A is mapped to at most one B (partial function)
some	set	Each B mapped to at least once
some	some	Every A mapped from and every B mapped to
some	one	Each A used exactly once, each B used at least once
some	lone	Each A used at most once, each B used at least once
one	set	Each B used exactly once, no other restrictions (one A can map to two B atoms)
one	some	Each B used exactly once, each A used at least once
one	one	Only satisfiable if #A = #B, bijection
one	lone	At most #A arrows, exactly #B arrows, each A used at most once
lone	set	Each B used at most once
lone	some	Each A used at least once and each B used at most once
lone	one	Each A used exactly once, each B used at most once
lone	lone	Each A used at most once, each B used at most once

Multi-relations

Not all multiplicities will have valid models. For example,

```
sig A {}
sig B {}
one sig C {
  r: A one -> one B
}

run {} for exactly 3 A, exactly 2 B
```

Since r must be 1-1, and there's different numbers of A and B sigs, nothing satisfies this model.

Multi-relations can go higher than ternary using the same syntax, but this is generally not recommended.

Signature Multiplicity

In addition to having multiplicity relationships, we can put multiplicities on the signatures themselves.

```
one sig Foo {}
some sig Bar {}
//etc
```

By default, signatures have multiplicity set, and there may be zero or more in the model. By making the signature one, every model will have exactly one atom of that signature. By writing some, there will be at least one. By writing lone, there will be zero or one.

Subtypes

We can make some signatures subtypes of other signatures.

in
Writing sig Child in Parent creates an inclusive subtype: any Parent atoms may or may not also be a Child. This is also called a “subset subtype”.

```
sig Machine {}

sig Broken in Machine {}
sig Online in Machine {}
```

In this case, any Machine can also be Broken, Online, both, or neither.

Subtypes

+

A single inclusive subtype can be defined for many parent signatures. We can do this by using the **set union operator** on the parent signatures.

sig Bill, Client {}

sig Closed in Bill + Client {}

Subtypes

extends
Writing sig Child extends Parent creates a subtype, as with in. Unlike in, though, any Parent atom can only match up to one extension.

```
sig Machine {}

sig Server extends Machine {}
sig Client extends Machine {}
```

In this case, any Machine can also be a Server, a Client, or neither, but not both.

Subtypes

Something can belong to both extend and in subtypes.

```
sig Machine {}
sig Broken in Machine {}

sig Server extends Machine {}
sig Client extends Machine {}
```

A Machine can be both a Server and Broken, or a Client and Broken, or just one of the three, or none at all.

Subtypes

abstract
If you make a signature abstract, then all atoms of the signature will belong to extensions. There will be no atoms that are just the supertype and not any of the subtypes.

abstract sig Machine {}
sig Broken in Machine {}

sig Server extends Machine {}
sig Client extends Machine {}
Here any machine must be either a Server or a Client. They still may or may not be Broken.

Subtypes

Warning

If there is nothing extending an abstract signature, the abstract is ignored.

Tip

You can place multiple signatures on the same line.

sig Server, Client extends Machine {}

Subtypes and Relationships

All subtypes are also their parent type. So if we have

```
sig B {}
sig C in B {}

sig A {
  , b: B
  , c: C
}
```

Then the b relation can map to atoms of C, and c cannot map to elements of B that are not also in C.

Subtypes and Relationships

Tip

If you want to map to elements of B that are not also in C, you can write:

```
sig A {
  , b: B - C
}
```

Child Relations

Children automatically inherit all of their Parent fields, and also can define their own fields. We can have:

```
sig Person {}
sig Account {
  , person: Person
}

sig PremiumAccount in Account {
  , billing: Person
}
```

Then all Account atoms will have the person field, while all PremiumAccount atoms will have both a person field and a billing field.

Child Relations

Note

This also applies to Implicit Facts. If Account has an implicit fact, it automatically applies to PremiumAccount.

It is not possible redefine a relationship, only to add additional ones.

Enums

Enums are a special signature.

enum Time {Morning, Noon, Night}

The enum will always have the defined atoms in it. Additionally, the atom will have an ordering. In this case, Morning will be the first element, Noon the second, and Night will be the third. You can use enums in facts and predicates, but you cannot add additional properties to them.

Enums

Tip

If you want to use an enumeration with properties, you can emulate this by using one and signature extensions.

```
abstract sig Time {}

one sig Morning, Noon, Night extends Time {
   time: Time
}
```

You can also use this to make enumerations without a fixed number of elements, by using lone instead.

Enums

Warning

Each enum implicitly imports ordering. The following is invalid:

enum A {a}

enum B {b}

run {some first}

As it is ambiguous whether first should return a or b. If you need to use both an enum inside of a dynamic model, be sure to use a namespace when importing ordering.

Sets and Relations

Sets

A set is a collection of unique, unordered elements. All Alloy expressions use sets of atoms and Relations. All elements of a set must all be either atoms, relations, or multi relations of the same arity, but may be different types of each category.

In expressions, the name of the signature is equal to the set of all atoms in that signature. The same is true for signature fields. Given

```
sig Teacher {}
sig Student {
  teacher: Teacher
}
```

Sets

Then the spec recognizes Student as the set of all atoms of type Student, and likewise with the Teacher signature and the teacher relationship.

Everything in Alloy is a set. If S1 is a Student atom, then S1 is the set containing just S1 as an element.

There are also two special sets:

none is just the empty set. Saying no Set is the same as saying Set = none. See Expressions.
univ is the set of all atoms in the model. In this example, univ = Student + Teacher.
Note

By default, the analyzer also generates a set of integers for each model, which will appear in univ. This can almost always be ignored in specifications (but see # below).

Set Operators

Set operators can be used to construct new sets from existing ones, for use in expressions and predicates.

S1 + S2 is the set of all elements in either S1 or S2 (set **union**).
S1 - S2 is the set of all elements in S1 but not S2 (set **difference**).
S1 & S2 is the set of all elements in both S1 and S2 (set **intersection**).
S1 = {A, B}
S2 = {B, C}

S1 + S2 = {A, B, C}
S1 - S2 = {A}
S1 & S2 = {B}

-> used as an operator

Given two sets, Set1 -> Set2 is the Cartesian product of the two: the set of all relations that map any element of Set1 to any element of Set2.

Set1 = {A, B}
Set2 = {X, Y, Z}

Set1 -> Set2 = { A -> X, A -> Y, A -> Z, B -> X, B -> Y, B -> Z}

As with other operators, a standalone atom is the set containing that atom. So we can write A -> (X + Y) to get (A -> X + A -> Y).

Tip

univ -> univ is the set of all possible relations in your model.

Integers

Alloy has limited support for integers. To enforce bounded models, the numerical range is finite. By default, Alloy uses models with 4-bit signed integers: all integers between -8 and 7. If an arithmetic operation would cause this to overflow, then the predicate is automatically declared false. In the Evaluator, however, it will wrap the overflowed number.

Tip

The numerical range can be changed by placing a scope on Int. The number of the scope is the number of bits in the signed integers. For example, if the scope is 5 Int, the model will have all integers between -16 and 15.

Integers

All arithmetic operators are over the given model's numeric range. To avoid conflict with set and relation operators, the arithmetic operators are written as Functions:

add[1, 2]
sub[1, 2]
mul[1, 2]
div[3, 2] -- integer divison, drop remainder
rem[1, 2] -- remainder

#
#S is the number of elements in S.

Integers

You can use receiver syntax for this, and write add[1, 2] as 1.add[2]. There are also the following comparison predicates:

1 <= 2
1 < 2
1 > 2
1 >= 2
1 != 2
1 = 2
As there are no corresponding symbols for elements to overload, these operators are written as infixes.

Warning

Sets of integers have non-intuitive properties and should be used with care.

Sets of numbers

For set operations, a set of numbers are treated as a set. For arithmetic operations, however, a set of numbers is first summed before applying the operator. This is equivalent to using the sum[] function.

(1 + 2) >= 3 -- true
(1 + 2) <= 3 -- true
(1 + 2) = 3 -- false
(1 + 2).plus[0] = 3 -- true
(1 + 1).plus[0] = 2 -- false

Relations

Given the following spec

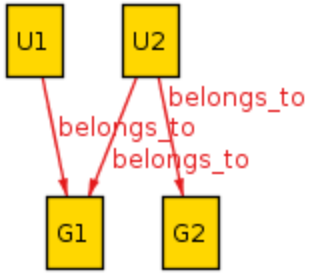
```
sig Group {}
sig User {
  belongs_to: set Group
}
```

belongs_to describes a relation between User and Group. Each individual relation consists of a pair of atoms, the first being User, the second being Group. We write an individual relation with ->. One possible model might have

```
belongs_to = {
  U1 -> G1 +
  U2 -> G1 +
  U2 -> G2
}
```

Relations

Relations do not need to be 1-1: here two users map to G1 and one user maps to both G1 and G2.



Relations in Alloy are first class objects, and can be manipulated and used in expressions. [This assumes you already know the set operations]. For example, we can reverse a relation by adding ~ before it:

```
~belongs_to = {
  G1 -> U1 +
  G1 -> U2 +
  G2 -> U2
}
```

The . Operator

The dot (.) operator is the most common relationship operator, and has several different uses. The dot operator is left-binding: a.b.c is parsed as (a.b).c, not a.(b.c).

Set.rel
If Set is an individual atom, this returns all elements that said atom maps to. If Set is more than one atom, this gets all elements they map to.

U1.belongs_to = G1
(U1 + U2).belongs_to = {G1, G2}

Tip

In this case, we can find all groups in the relation with User.belongs_to. However, some relations may mix different types of atoms. In that case univ.~rel is the domain of rel and univ.rel is the range of rel.

The . Operator

For Multirelations, this will return the “tail” of the relation.
Eg if rel = A -> B -> C, then A.rel = B -> C.

rel.Set

Writing rel.Set is equivalent to writing Set.~rel. See ~rel.

belongs_to.G1 = {U1, U2}
G1.~belongs_to = {U1, U2}

rel1.rel2

The . Operator

We can use the dot operator with two relations. It returns the inner product of the two relations. For example, given

rel1 = {A -> B, B -> A}
rel2 = {B -> C, B -> D, A -> E}

rel1.rel2 = { A -> C, A -> D, B -> E}

In our case with Users and Groups, belongs_to.~belongs_to maps every User to every other user that shares a group.

Note

The operator isn't overloaded; it's the same operator with the same semantics for both Set.rel and rel1.rel2.

[]

rel[elem] is equivalent to writing elem.(rel). It has a lower precedence than the . operator, which makes it useful for Multirelations. If we have

```
sig Light {
   state: Color -> Time
}
```

Then L.state[C] would be all of the times T where the light L was color C. The equivalent without [] would be C.(L.state).

iden

iden is the relationship mapping every element to itself. If we have an element a in our model, then (a -> a) in iden.

An example of iden’s usefulness: if we want to say that rel doesn’t have any cycles, we can say no iden & ^rel.

Additional Operators

Note

You cannot use ~, ^, or * with higher-arity relations.

~rel
As mentioned, ~rel is the reverse of rel.

^ and *
These are the transitive closure relationships. Take the following example:

```
sig Node {
  edge: set Node
}
```

Additional Operators

N.edge is the set of all nodes that N connects to.
N.edge.edge is the set of all nodes that an edge of N connects to.
N.edge.edge.edge is the set of all nodes that are an edge of an edge of N, ad infinitum.

If we want every node that is connected to N, this is called the transitive closure and is written as N.^edge.

^ does not include the original atom unless it's transitively reachable! In the above example, N in N.^edge iff the graph has a cycle containing N.

If we want to also include N, use N.*edge instead.

Additional Operators

^ operates on the relationship, so ^edge is also itself a relationship and can be manipulated like any other. We can write both ~^edge and ^~edge.

It also works on arbitrary relationships.
U1.^(belongs_to.~belongs_to) is the set of people that share a group with U1, or share a group with people who share a group with U1, ad infinitum.

Warning

By itself *edge will include iden! *edge = ^edge + iden. For best results only use * immediately before joining the closure with another set.

Advanced Operators

<: and :>
<: is domain restriction. Set <: rel is all of the elements in rel that start with an element in Set. :> is the range restriction, and works similarly: rel :> Set is all the elements of rel that end with an element in Set.

This is mostly useful for directly manipulating relations. For example, given a set S, we can map every element to itself by doing S <: iden. We can also use restrictions to disambiguate overloaded fields. If we have

```
abstract sig Node {
  , edges: set Node
}

some sig Red, Blue extends Node {}
```

Then Blue <: edges :> Red is the set of all edges from Blue nodes to Red ones.

Advanced Operators

++

rel1 ++ rel2 is the union of the two relations, with one exception: if any relations in rel1 that share a “key” with a relation in rel2 are dropped. Think of it like merging two dictionaries.

Formally speaking, we have

rel1 ++ rel2 = rel1 - (rel2.univ <: rel1) + rel2
Some examples of ++:

(A -> B + A -> C) ++ (A -> A) = (A -> A)
(A -> B + A -> C) ++ (A -> A + A -> C) = (A -> A + A -> C)
(A -> B + A -> C) ++ (C -> A) = (A -> B + A -> C + C -> A)
(A -> B + B -> C) ++ (A -> A) = (A -> A + B -> C)

It’s mostly useful for modeling Time.

Advanced Operators

Note

When using multirelations the two relations need the same arity, and it overrides based on only the first element in the relations.

Set Comprehensions

Set comprehensions are written as

{x: Set1 | expr[x]}
The expression evaluates to the set of all elements of Set1 where expr[x] is true. expr can be any expression and may be inline. Set comprehensions can be used anywhere a set or set expression is valid.

Set comprehensions can use multiple inputs.

{x: Set1, y: Set2, ... | expr[x,y]}
In this case this comprehension will return relations in Set1 -> Set2.

Expressions and Constraints

Expressions and Constraints

Expressions are anything that returns a number, Boolean, or set. Boolean expressions are also called Constraints.

Information about relational expressions are found in the Sets and Relations chapters. There are two additional constructs that can be used with both boolean and relational expressions.

Let

let defines a local value for the purposes of the subexpression.

let x = A + B, y = C + D | x + y

In the context of the let expression x + y = (A + B) + (C + D). let is mostly used to simplify complex expressions and give meaningful names to intermediate computations.

If writing a boolean expression, you may use a {} instead of |.

let bindings are not recursive. A let binding may not refer to itself or a future let binding.

Tip: As with predicate parameters, let can shadow a global value. You can use the @ operator to retrieve the global value.

implies - else

When used in conjunction with else, implies acts as a conditional. p implies A else B

returns A if p is true and B if p is false.

p must be a boolean expression.

If A and B are boolean expressions, then this acts as a constraint. The else can be left out if using implies as a constraint.

Constraints
Bar expressions

A bar expression is one of the form:

some x: Set | expr

In this context, the expression is true iff expr is true. The newline is optional.

Constraints
Paragraph expressions

If multiple constraints are surrounded with braces, they are all and-ed together. The following two are equivalent:

```
expr1 or {
  expr2
  expr3
    ...
}
```

```
expr1 or (expr2 and expr3 and ...)
```

Constraint Types

All constraints can be inverted with not or !. To say that A is not a subset of B, you can write

A !in B,
A not in B,
!(A in B), etc.

Relation Constraints

=

A = B means that both sets of atoms or relations have the exact same elements. = cannot be used to compare two booleans. Use iff instead.

in

A in B means that every element of A is also an element of B. This is also known as a “subset” relation.
x in A means that x is an element of the set A.

Note

The above two definitions are equivalent as all atoms are singleton sets: x is the set containing x, so x in A is “the set containing just x is a subset of A”.

size constraints

There are **four** constraints on the number of elements in a set:

no A means A is empty.
some A means A has at least one element.
one A means A has exactly one element.
lone A means A is either empty or has exactly one element.

In practice, no and some are considerably more useful than one and lone.

Note

Relations are each exactly one element, no matter the order of the relation. If a, b, and c are individual atoms, (a -> b -> c) is exactly one element, while (a -> b) + (a -> c) is two.

size constraints

disj[A, B]

disj[A, B] is the predicate “A and B share no elements in common”.

Any number of arguments can be used, in which case disj is pairwise-disjoint. This means that disj[A, B, C] is equivalent to disj[A, B] and disj[B, C] and disj[A, C].

Boolean Constraints

Boolean constraints operate on Booleans or predicates. They can be used to create more complex constraints.

All Boolean constraints have two different forms, a symbolic form and an English form. For example, A && B can also be written A and B.

word	**symbol**
and	&&
or	||
not	!
implies	=>
iff	<=>

Boolean Constraints

The first three are self-explanatory. The other two are covered below:

implies (=>)
P implies Q is true if Q is true whenever P is true. If P is true and Q is false, then P implies Q is false. If P is false, then P implies Q is automatically true. P implies Q else T is true if P and Q are true or if P is false and T is true.

(Consider the statement x > 5 implies x > 3.
If we pick x = 4, then we have **false implies true**).

iff (<=>)
P iff Q is true if P and Q are both true or both false. Use this for booleans instead of =.

Tip

xor[A, B] can be written as A <=> !B.

Quantifiers

A quantifier is an expression about the elements of a set. All of them have the form

some x: A | expr

This expression is true if expr is true for any element of the set of atoms A. As with let, x becomes a valid identifier in the body of the constraint.

Instead of using a pipe, you can also write it as

```
some x: Set {
  expr1
  ...
}
```

Quantifiers

In which case it is treated as a standard paragraph expression.

The following quantifiers are available:

some x: A | expr is true for at least one element in A.
all x: A | expr is true for every element in A.
no x: A | expr is false for every element of A.
[A] **one** x: A | expr is true for exactly one element of A.
[A] **lone** x: A is equivalent to (one x: A | expr) or (no x: A | expr).

As discussed below, one and lone can have some unintuitive consequences.

Tip

As with all constraints, A can be any set expression. So you can write
some x: (A + B - C).rel, etc.

Multiple Quantifiers

There are two syntaxes to quantify over multiple elements:

```
-- 1
some x, y, ...: A | expr

-- 2
some x: A, y: B, ... | expr
```

For case (1) all elements will be drawn from A. For case (2) the quantifier will be over all possible combinations of elements from A and B. The two forms can be combined, as in

```
all x, y: A, z: B, ... | expr.
```

Multiple Quantifiers

Elements drawn do not need to be distinct. This means, for example, that the following is automatically false if A has any elements:

```
all x, y: A |
  x.rel != y.rel
```

Multiple Quantifiers

As we can pick the same element for x and y. If this is not your intention, there are two ways to fix this:

```
-- 1
all x, y: A |   x != y => x.rel != y.rel

-- 2
all disj x, y: A |   x.rel != y.rel
```

For case (1) we can still select the same element for x and y; however, the x != y clause will be false, making the whole clause true. For case (2), using disj in a quantifier means we cannot select the same element for two variables.

Multiple Quantifiers

one and lone behave unintuitively when used in multiple quantifiers. The following two statements are different:

```
one f, g: S | P[f, g]                -- 1

one f: S | one g: S | P[f, g]        -- 2
```

Multiple Quantifiers

Constraint (1) is only true if there is exactly one pair f, g that satisfies predicate P. Constraint (2) says that there's exactly one f such that there's exactly one g. The following truth table will satisfy clause (2) but not (1):

f	g	P[f, g]
A	B	T
A	C	T
B	A	T
B	C	T
C	B	T
C	A	F

As C is the only one where there is exactly one g that satisfies P[C, g]. As a rule of thumb, use only some and all when writing multiple clauses.

Relational Quantifiers

When using a run command, you can define a **some** quantifier over a relation:

```
sig Node {
    edge: set Node
}

pred has_self_loop {
    some e: edge | e = ~e
}

run {
    has_self_loop
}
```

Relational Quantifiers

When using a check command, you can define **all** and **no** quantifiers over relations:

```
assert no_self_loops {
      no e: edge | e = ~e
   }

check no_self_loops
```

You cannot use all or no in a run command or use some in a check command. You cannot use higher-order quantifiers in the Evaluator regardless of the command.

Predicates and Functions

Predicates and Functions

Predicates

A predicate is like a programming function that returns a boolean. While they are a special case of Alloy functions, they are more fundamental to modeling and addressed first.

Predicates take the form

```
pred name {
    constraint
}
```

Note:Predicates and functions cannot, in the general case, be recursive. Limited recursion is possible,

Predicate sample

```
sig A {}

pred at_least_one_a {
  some A
}

pred more_than_one_a {
  at_least_one_a and not one A
}

run more_than_one_a
```

Predicate Parameters

Predicates can also take arguments.

```
pred foo[a: Set1, b: Set2...] {
  expr
}
```

The predicate is called with foo[x, y], using brackets, not parents.

In the body of the predicate, a and b would have the corresponding values.

Predicate Receiver Syntax

- The initial argument to a predicate can be passed in via a . join. The following two are equivalent:

- pred[x, y, z]
- x.pred[y, z]

Functions

Alloy functions have the same structure as predicates but also return a value.

Unlike functions in programming languages, they are always executed within an execution run, so their results are available in the visualization and the evaluator even if you haven't called them directly.

This is very useful for some visualizations, although the supporting code can be disorienting when transitioning from "regular" programming languages.

Functions

```
fun name[a: Set1, b: Set2]: output_type {
  expression
}
```

if a function is constant (does not take any parameters), the analyzer casts it to a constant set. This means if we have a function of parameter

```
fun foo: A -> B {
    expression
}
```

Then ^foo is a valid expression.

Predicates and functions Overloading

Predicates and functions may be overloaded, as long as it's unambiguous which function applies. The following is valid:

```
sig A {}

sig B {}

pred foo[a: A] {                    --1
  a in A
}

pred foo[b: B] {                    --2
  b in B
}

run {some a: A | foo[a]}
```

Predicates and functions Overloading

As when foo is called, it's unambiguous whether it means (1) or (2). If we instead replaced sig B with sig B extends A, then it's ambiguous and the call is invalid.

Overloading can happen if you import the same parameterized module twice.For example, given the following:

```
open util/ordering[A]
open util/ordering[B]

sig A, B {}
run {some first}
```

Parameter Overrides

The parameters of a function (or predicate) can shadow a global value. In this case, you can retrieve the original global value by using @val.

```
sig A {}

pred f[A: univ, b: univ] {
  b in A                    -- function param
  b in @A                   -- global signature
}
```

Facts

A fact has the same form as a global predicate:

```
fact name {
  constraint
}
```

Tip

For facts, the name is optional. In addition, the name can be a string. So this is a valid fact:

```
fact "no cycles" {
  all n: Node | n not in n.^edge
}
```

Facts

A fact is always considered true by the Analyzer. Any models that would violate the fact are discarded instead of checked.

This means that if a potential model both violates an assertion and a fact, it is not considered a counterexample.

```
sig A {}

-- This has a counterexample
check {no A}

-- Unless we add this fact
fact {no A}
```

Implicit Facts

You can write a fact as part of a signature. The implicit fact goes after the signature definition and relations.

Inside of an implicit fact, you can get the current atom with this. Fields are automatically expanded in the implicit fact to this.field.

```
sig Node {
  edge: set Node
} {
  this not in edge
}
```

Implicit Facts

This means you cannot apply the relation to another atom of the same signature inside the implicit fact. You can access the original relation by using the @ operator:

```
-- undirected graphs only
sig Node {
  , edge: set Node
}
{
  all link: edge | this in link.edge          -- invalid
  all link: edge | this in link.@edge         -- valid
}
```

Macros

A macro is a similar to a predicate or function, except it is expanded before runtime. For this reason, macros can be used as part of signature fields. Parameters to macros also don’t need to be given types, so can accept arbitrary signatures and even boolean constraints. Macros are defined with let in the top scope.

```
let selfrel[Sig] = { Sig -> Sig }
let many[Sig] = { some Sig and not one Sig }

sig A {
  rel: selfrel[A]
}

run {many[A]}
```

Commands

A command is what actually runs the analyzer. It can either find models that satisfy your specification, or counterexamples to given properties.

By default, the analyzer will run the top command in the file. A specific command can be run under the Execute menu option.

run
run tells the analyzer to find a matching example of the spec.

run pred
Find examples where pred is true. If no examples match, the analyzer will suggest the predicate is inconsistent (see unsat core). The predicate may be consistent if the scope is off.

Commands

```
sig Node {
   edge: set Node
}

pred self_loop[n: Node] {
  n in n.edge
}

pred all_self_loop {
  all n: Node | self_loop[n]
}

run all_self_loop
```

Commands

The analyzer will title the command as the predicate.

```
Executing "Run all_self_loop"
   Sig this/Node scope <= 3
   Solver=minisatprover(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   79 vars. 12 primary vars. 101 clauses. 4ms.
   Instance found. Predicate is consistent. 5ms.
```

Commands

run {constraint}
Finds an example satisfying the ad-hoc constraint in the braces.

// some node with a self loop
run {some n: Node | self_loop[n]}

Tip

The analyzer will title the command run${num}. You can give the **command a name by prepending the run with name**::

some_self_loop: run {some n: Node | self_loop[n]}

Commands

check

check tells the Analyzer to find a counterexample to a given constraint. You can use it to check that your specification behaves as you expect it to.

Unlike with run commands, check uses assertions:

```
assert no_self_loops {
    no n: Node | self_loop[n]
}

check no_self_loops
```

Commands

Asserts may be used in check commands but not run commands.
Assertions may not be called by other predicates or assertions.

You can also call check with an ad-hoc constraint:

```
check {no n: Node | self_loop[n]}
```

check can also be given a named command.

Commands

Scopes

All alloy models are bounded: they must have a maximum possible size. If not specified, the analyzer will assume that there may be up to three of each top-level signature and any number of relations. This is called the scope, and can be changed for each command.

Given the following spec:

```
sig A {}
sig B {}
```

Commands

We can write the following scopes:

run {} for 5: Analyzer will look for models with up to five instances of each A and B.

run {} for 5 but 2 A: Analyzer will look for models with up to two instances of A.
run {} for 5 but exactly 2 A: Analyzer will only look for models with exactly two A. The exact scope may be higher than the general scope.

run {} for 5 but 2 A, 3 B: Places scopes on A and B.

If you are placing scopes on all of the signatures, the for N except is unnecessary: the last command can be written as run {} for 2 A, 3 B.

Commands

Tip

When using Arithmetic Operators, you can specify Int like any other signature:

run foo for 3 Int

Note
You cannot place scopes on relations. Instead, use a predicate.

```
sig A {
  rel: A
}

run {#rel = 2}
```

Commands

Scopes on Subtypes

Special scopes may be placed on extensional subtypes. The following is valid:

```
sig Plant {}

sig Tree extends Plant {}
sig Grass extends Plant {}

run {} for 4 Plant, exactly 2 Tree
```

Grass does not need to be scoped, as it is considered part of Plant. The maximum number of atoms for a subtype is either it or its parent's scope, whichever is lower. The parent scope is shared across all children. In this command, there are a maximum of four Plants, exactly two of which will be Tree atoms. Therefore there may be at most two Grass atoms.

Commands

In contrast, special scopes may not be placed on subset types. The following is invalid:

```
sig Plant {}

sig Seedling in Plant {}

run {} for 4 Plant, exactly 2 Seedling
```

Since Seedling is a subset type, it may not have a scope. If you need to scope on a subtype, use a constraint:

```
run {#Seedling = 2} for 4 Plant
```

Modules

Modules

Alloy modules are similar to programming languages and act as the namespaces. Alloy comes with a standard library of utility modules.

Simple Modules

open util/relation as r

Imports must be at the top of the file. Modules may import new signatures into the spec.

Modules can be imported multiple times under different namespaces.

Modules

Namespaces
A module can be namespaced by importing as a name. Namespaces are accessed with /. This is also called a qualified import.

```
open util/relation as r


-- later

r/dom
```

Modules

Parameterized Modules

A parameterized module is “generic”: its functions and predicates are defined for some arbitrary signature. When you import a parameterized module, you must pass in a signature. Its functions and predicates are then specialized to be defined for that signature.

```
open util/ordering[A]

sig A {}

run {some first} -- returns an A atom
```

Modules

Normally ord/first returns an abstract elem. By parameterizing the module with A, the function now returns an A atom.

The input must be a full signature and not a subset of one.

A parameterized module can be imported multiple times using Namespaces.

Note

The following built-in modules are parameterized: ordering, time, graph, and sequence.

Modules

Creating Modules
The syntax for a module is

module name
At the beginning of the file.

Private
Any module predicate, function, and signature can be preceded by private, which means it will not be imported into other modules.

module name

private sig A {}

Modules

Creating Parameterized Modules

module name[sig]

-- predicates and functions should use sig

Modules

- boolean
  - Functions
- graph
  - Functions
  - Predicates
- integer
  - Functions
  - Predicates
- naturals
  - Functions
  - Predicates

- ordering
  - Functions
  - Predicates
- relation
  - Functions
  - Predicates
- ternary
- time
- Macros

Modules - boolean

boolean

Emulates boolean variables by creating True and False atoms.

```
-- module definition
module util/boolean

abstract sig Bool {}
one sig True, False extends Bool {}

pred isTrue[b: Bool] { b in True }

pred isFalse[b: Bool] { b in False }
```

Modules - boolean

In our code:

```
-- our code
open util/boolean

sig Account {
  premium: Bool
}
```

Booleans created in this matter are not “true” booleans and cannot be used as part of regular Constraints, IE you cannot do bool1 && bool2. Instead you must use the dedicated boolean predicates, below. As such, boolean should be considered a proof-of-concept and is generally not recommended for use in production specs. You should instead represent booleans using subtyping.

Modules - boolean

Functions

All of the following have expected logic, but return Bool atoms:

Not
And
Or
Xor
Nand
Nor

So to emulate bool1 && bool2, write bool1.And[bool2].is True.

Modules - graph

Graph provides predicates on relations over a parameterized signature.

```
open util/graph[Node]

sig Node {
  edge: set Node
}

run {
  dag[edge]
}
```

Modules - graph

Notice that graph is parameterized on the signature, but the predicate takes in a relation. This is so that you can apply multiple predicates to multiple different relations, or different subsets of the same relation. The graph module uses some specific terminology:

This means that in a completely unconnected graph, every node is both a root and a leaf.

Functions
funroots[r: node-> node]
Return type: set Node

Returns the set of nodes that are not connected to by any other node.

Modules - graph

Warning

this is not the same meaning of root as in the rootedAt predicate! For the predicate, a root is a node that transitively covers the whole graph. Interally, util/graph uses rootedAt and not roots.

funleaves[r: node-> node]
Return type: set Node

Returns the set of nodes that do not connect to any other node.

Modules - graph

Note

If r is empty, roots[r] = leaves[r] = Node. If r is undirected or contains enough self loops, roots[r] = leaves[r] = none.

fun innerNodes[r: node-> node]
Returns: All nodes that aren’t leaves
Return type: set Node

Modules- graph

Predicates
pred undirected [r: node->node]
r is symmetric.

pred noSelfLoops[r: node->node]
r is irreflexive.

pred weaklyConnected[r: node->node]

For any two nodes A and B, there is a path from A to B or a path from B to A. The path may not necessarily be bidirectional.

Modules- graph

pred stronglyConnected[r: node->node]
For any two nodes A and B, there is a path from A to B and a path from B to A.

pred rootedAt[r: node->node, root: node]
All nodes are reachable from root.

Warning

this is not the same meaning of root as in the roots function! For the function, a root is a node no node connects to. Interally, util/graph uses rootedAt and not roots.

Modules- graph

pred ring [r: node->node]
r forms a single cycle.

pred dag [r: node->node]
r is a dag: there are no self-loops in the transitive closure.

pred forest [r: node->node]
r is a dag and every node has at most one parent.

pred tree [r: node->node]
r is a forest with a single root node.

pred treeRootedAt[r: node->node, root: node]
r is a tree with node root.

Modules-integer

integer
Emulates integers.

A collection of utility functions for using Integers in Alloy. Note that integer overflows are silently truncated to the current bitwidth using the 2's complement arithmetic, unless the ''forbid overfows'' option is turned on, in which case only models that do not have any overflows are analyzed.

Warning

The main challenge with this module is the distinction between Int and int.

Modules-integer

Int is the set of integers that have been instantiated, whereas int returns the value of an Int. You have to explicitly write int i to be able to add, subtract, and compare `Int`s.

```
open util/integers

fact ThreeExists { // there is some integer whose value is 3
  some x: Int | int x = 3
}

fun add[a, b: Int]: Int {
  {i: Int | int i = int a + int b}
}

run add for 10 but 3 int expect 1
```

Modules-integer

To try this module out, in Alloy Analyzer's evaluator, you may also issue the following commands (suppose that allow generated a set with numbers ranging from -8 to 7):

-8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7

1 + 3
4

7 + 1
-8

Modules-integer

Functions

fun add [n1, n2: Int]
Return type: one Int
Returns n1 + n2.

fun plus [n1, n2: Int]
Return type: one Int
Returns n1 + n2.

fun sub [n1, n2: Int]
Return type: one Int
Returns n1 - n2.

fun minus [n1, n2: Int]
Return type: one Int
Returns n1 - n2.

fun mul [n1, n2: Int]
Return type: one Int
Returns n1 * n2.

fun div [n1, n2: Int]
Return type: one Int

Modules-integer

Returns the division with ‘’round to zero’’ semantics, except the following 3 cases:

if a is 0, then it returns 0
else if b is 0, then it returns 1 if a is negative and -1 if a is positive
else if a is the smallest negative integer, and b is -1, then it returns a

Modules-integer

fun rem [n1, n2: Int]
Return type: one Int
Returns the unique integer that satisfies a = ((a/b)*b) + remainder.

fun negate [n: Int]
Return type: one Int
Returns the negation of n.

fun signum [n: Int]
Return type: one Int

Returns the signum of n (aka sign or sgn). In particular, n < 0 => (0 - 1) else (n > 0 => 1 else 0).

Modules-integer

fun int2elem [i: Int, next: univ->univ, s: set univ]
Return type: lone s
Returns the ith element (zero-based) from the set s in the ordering of next, which is a linear ordering relation like that provided by util/ordering.

fun elem2int [e: univ, next: univ->univ]
Return type: lone Int
Returns the index of the element (zero-based) in the ordering of next, which is a linear ordering relation like that provided by util/ordering.

fun max
Return type: one Int
Returns the largest integer in the current bitwidth.

fun min
Return type: one Int
Returns the smallest integer in the current bitwidth.

Modules-integer

fun next
Return type: Int -> Int
Maps each integer (except max) to the integer after it.

fun prev
Return type: Int -> Int
Maps each integer (except min) to the integer before it.

fun max [es: set Int]
Return type: lone Int
Given a set of integers, return the largest element.

fun min [es: set Int]
Return type: lone Int
Given a set of integers, return the smallest element.

Modules-integer

fun prevs [e: Int]
Return type: set Int
Given an integer, return all integers prior to it.

fun nexts [e: Int]
Return type: set Int
Given an integer, return all integers following it.

fun larger [e1, e2: Int]
Return type: Int
Returns the larger of the two integers.

fun smaller [e1, e2: Int]
Return type: Int
Returns the smaller of the two integers.

Modules-integer

Predicates

pred eq [n1, n2: Int]
True iff n1 is equal to n2.

pred gt [n1, n2: Int]
True iff n1 is greater than n2.

pred gte [n1, n2: Int]
True iff n1 is greater than or equal to n2.

pred lt [n1, n2: Int]
True iff n1 is less than n2.

pred lte [n1, n2: Int]
True iff n1 is less than or equal to n2.

pred zero [n: Int]
True iff n is equal to 0.

pred pos [n: Int]
True iff n is positive.

pred neg [n: Int]
True iff n is negative.

pred nonpos [n: Int]
True iff n is non-positive.

pred nonneg [n: Int]
True iff n is non-negative.

Modules - naturals

Emulates natural (non-negative) numbers.

This is an utility with functions and predicates for using the set of nonnegative integers (0, 1, 2, . . .). The number of naturals present in an analysis will be equal to the scope on Natural. Specifically, if the scope on Natural is N, then the naturals 0 through N-1 will be present.

```
open util/natural

fun sum[a: Natural, b: Natural]: Natural {
  {x:Natural | x = natural/add[a,b]}
}

run show for 3
```

Modules - naturals

To try this module out, in Alloy Analyzer's evaluator, you may invoke the function defined above as follows:

```
sum [natural/Natural1, natural/Natural1]
  {natural/Natural$2}

sum [natural/Natural1, natural/Natural2]
  {}
```

Modules - naturals

Functions

fun inc [n: Natural]
Return type: one Natural
Returns n + 1.

fun dec [n: Natural]
Return type: one Natural
Returns n - 1.

fun add [n1, n2: Natural]
Return type: one Natural
Returns n1 + n2.

fun sub [n1, n2: Natural]
Return type: one Natural
Returns n1 - n2.

fun mul [n1, n2: Natural]
Return type: one Natural
Returns n1 * n2.

fun div [n1, n2: Natural]
Return type: one Natural
Returns n1 / n2.

fun max [ns: set Natural]
Return type: one Natural
Returns the maximum integer in ns.

fun min [ns: set Natural]
Return type: one Natural
Returns the minimum integer in ns.

Modules - naturals

Predicates

pred gt [n1, n2: Natural]
True iff n1 is greater than n2.

pred gte [n1, n2: Natural]
True iff n1 is greater than or equal to n2.

pred lt [n1, n2: Natural]
True iff n1 is less than n2.

pred lte [n1, n2: Natural]
True iff n1 is less than or equal to n2.

Modules - ordering

Ordering places an ordering on the parameterized signature.

```
open util/ordering[A]

sig A {}

run {
  some first -- first in ordering
  some last  -- last in ordering
  first.lt[last]
}
```

ordering can only be instantiated once per signature. You can, however, call it for two different signatures:

```
open util/module[Thing1] as u1
open util/module[Thing2] as u2

sig Thing1 {}
sig Thing2 {}
```

Modules - ordering

Warning

ordering forces the signature to be exact. This means that the following model has no instances:

```
open util/ordering[S]

sig S {}

run {#S = 2} for 3
```

In particular, be careful when using ordering as part of an assertion: the assertion may pass because of the implicit constraint!

Modules - ordering

See also

Module time

Adds additional convenience macros for the most common use case of ordering.
Sequences

For writing ordered relations vs placing top-level ordering on signatures.

Modules - ordering

Functions

fun first
Returns: The first element of the ordering
Return type: elem
See Also: last

fun prev
Return type: elem -> elem
See Also: next
Returns the relation mapping each element to its previous element.
This means it can be used as any other kind of relation:

```
fun is_first[e: elem] {
  no e.prev
}
```

Modules - ordering

fun prevs[e]
Returns: All elements before e, excluding e.
Return type: elem
See Also: nexts

fun smaller[e1, e2: elem]
Returns: the element that comes first in the ordering
See Also: larger

fun min[es: set elem]
Returns: The smallest element in es, or the empty set if es is empty
Return type: lone elem
See Also: max

Modules - ordering

Predicates

pred lt[e1, e2: elem]
See Also: gt, lte, gte
True iff e1 in prevs[e2].

Modules -relation

All functions and predicates in this module apply to any binary relation. univ is the set of all atoms in the model.

Functions

fun dom[r: univ->univ]
Return type: set univ
Returns the domain of r. Equivalent to univ.~r.

fun ran[r: univ->univ]
Return type: set univ
Returns the range of r. Equivalent to univ.r.

Modules -relation

Predicates

pred total[r: univ->univ, s: set]
True iff every element of s appears in dom[r].

pred functional[r: univ->univ, s: set univ]
True iff every element of s appears at most once in the left-relations of r.

pred function[r: univ->univ, s: set univ]
True iff every element of s appears exactly once in the left-relations of r.

pred surjective[r: univ->univ, s: set univ]
True iff s in ran[r].

pred injective[r: univ->univ, s: set univ]
True iff no two elements of dom[r] map to the same element in s.

Modules -relation

pred bijective[r: univ->univ, s: set univ]
True iff every element of s is mapped to by exactly one relation in r. This is equivalent to being both injective and surjective. There may be relations that map to elements outside of s.

pred bijection[r: univ->univ, d, c: set univ]
True iff exactly r bijects d to c.

pred reflexive[r: univ -> univ, s: set univ]
r maps every element of s to itself.

pred irreflexive[r: univ -> univ]
r does not map any element to itself.

pred symmetric[r: univ -> univ]
A -> B in r implies B -> A in r

Modules -relation

pred antisymmetric[r: univ -> univ]
A -> B in r implies B -> A notin r. This is stronger than not symmetric: no subset of r can be symmetric either.

pred transitive[r: univ -> univ]
A -> B in r and B - > C in r implies A -> C in r

pred acyclic[r: univ->univ, s: set univ]
r has no cycles that have elements of s.

pred complete[r: univ->univ, s: univ]
all x,y:s | (x!=y => x->y in (r + ~r))

Modules -relation

pred preorder[r: univ -> univ, s: set univ]
reflexive[r, s] and transitive[r]

pred equivalence[r: univ->univ, s: set univ]
r is reflexive, transitive, and symmetric over s.

pred partialOrder[r: univ -> univ, s: set univ]
r is a partial order over the set s.

pred totalOrder[r: univ -> univ, s: set univ]
r is a total order over the set s.

Modules -ternary

util/ternary provides utility functions for working with 3-arity Multirelations. All functions return either an element in the relation or a new transformed relation.

util/ternary

f	**f[a -> b -> c]**
dom	a
mid	b
ran	c
select12	a -> b
select13	a -> c
select23	b -> c
flip12	b -> a -> c
flip13	c -> b -> a
flip23	a -> c -> b

Modules -time

Automatically imports an ordered Time signature to your spec.

Warning

Time internally uses the ordering module. This means that the signature is forced to be exact.

See also

Module ordering

Modules -time

Macros

letdynamic[x]
Arguments:
x (sig) – any signature.
Expands to:
x one -> Time

dynamic can be used as part of a signature definition:

```
open util/time

abstract sig Color {}
        one sig Red, Green, Yellow extends Color {}

sig Light {
  , state: dynamic[Color]
}
```

Modules -time

At every Time, every Light will have exactly one color.

let dynamicSet[x]
Arguments:
x (sig) – any signature.
Expands to:
x -> Time

Equivalent to dynamic, except that any number of elements can belong to any given time:

```
open util/time

sig Keys {}

one sig Keyboard {
  pressed: dynamicSet[Keys]
}
```