Expressions and Constraints

Expressions

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.

See also

Integer expressions

Let

`let` defines a local value for the purposes of the subexpressic

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

In the context of the `let` expression `x + y = (A + B) + (C +` used to simplify complex expressions and give meaningful nam 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 `let` binding.

Warning

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. See below for details.

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.

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 (expr 2 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.

`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`	`<=>`

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
  ...
}
```

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`.

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
```

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.

`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
```

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
}
```

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.