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
}
```

Once defined, predicates can be used as part of boolean expressions. The following is a valid spec:

```
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
```


Warning

Predicates and functions **cannot**, in the general case, be recursive. Limit recursion is possible, see here for more info.

Parameters

Predicates can also take arguments.

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

The predicate is called with `foo[x, y]`, **using brackets, not parens**. In the body of the predicate, `a` and `b` would have the corresponding values.

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 visualisation and the evaluator even if you haven't called them directly. This is very useful for some visualisations, although the supporting code can be disorienting when transitioning from "regular" programming languages.

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

Tip

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.

[*] 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]}
```

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}
```

It is unclear whether `first` applies to `A` or `B`. To fix this, use Namespaced imports:

```
open util/ordering[A] as u1
open util/ordering[B] as u2

sig A, B {}
run {some u2/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
}
```

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}
```

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
}
```

[*] 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
}
```

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]}
```

See here for more information.