Signatures

Parameterized modules use signatures as a type system for their parameters. There are three categories of signatures: **predicate signatures**, **type signatures**, and **module signatures**.

Predicate signatures

Predicate signatures declare module parameters that will be substituted with predicates when the module is instantiated.

The substitution of predicate signatures relies on structural typing. That is, predicates do not have to be explicitly defined as implementing a predicate signature - they just have to match the return and argument types.

Predicate signatures are defined much like predicates themselves, but they do not have a body. In detail, a predicate signature definition consists of:

1. The keyword `signature`.
2. The keyword `predicate` (allows substitution with a predicate without result), or the type of the result (allows substitution with a predicate with result).
3. The name of the predicate signature. This is an identifier starting with a lowercase letter.
4. The arguments to the predicate signature, if any, separated by commas. For each argument, specify the argument type and an identifier for the argument variable.
5. A semicolon `;`.

For example:

```
signature int operator(int lhs, int rhs);
```

Type signatures

Type signatures declare module parameters that will be substituted with types when the module is instantiated. Type signatures may specify supertypes and required member predicates (in addition to those member predicates that are implied by the supertypes).

The substitution of type signatures relies on structural typing. That is, types do not have to be explicitly defined as implementing a type signature - they just need to have the specified (transitive) supertypes and member predicates.

In detail, a type signature definition consists of:

1. The keyword `signature`.

2. The keyword `class`.
3. The name of the type signature. This is an identifier starting with a uppercase letter.
4. Optionally, the keyword `extends` followed by a list of types, separated by commas.
5. Either a semicolon `;` or a list of predicate signatures enclosed in braces. The `signature` keyword is omitted for these contained signatures.

For example:

```
signature class ExtendsInt extends int;

signature class CanBePrinted {
  string toString();
}
```

Module signatures

Module signatures declare module parameters that will be substituted with modules when the module is instantiated. Module signatures specify a collection of types and predicates that a module needs to contain under given names and matching given signatures.

Unlike type signatures and predicate signatures, the substitution of type signatures relies on nominal typing. That is, the definition of a module must declare the module signatures it implements.

In detail, a type signature definition consists of:

1. The keyword `signature`.
2. The keyword `module`.
3. The name of the module signature. This is an identifier starting with a uppercase letter.
4. Optionally, a list of parameters for parameterized module signatures.
5. The module signature body, consisting of type signatures, predicate signatures, and default predicates enclosed in braces. The `signature` keyword is omitted for these contained signatures.

Module signature default predicates are syntactically constructed like predicate signatures, but preceded by the `default` keyword, and with a predicate body instead of the concluding semicolon `;`. Default predicate bodies are restricted in that they may not use entities that in any way depend on other module signature members or parameters of the module signature or any existing enclosing modules.

For example:

```
signature module MSig {
  class T;
  predicate restriction(T t);
  default string descr(T t) { result = "default" }
}

module Module implements MSig {
  newtype T = A() or B();

  predicate restriction(T t) { t = A() }
}
```

Parameterized module signatures

Module signatures can themselves be parameterized in exactly the same way as parameterized modules. This is particularly useful in combination with the dependent typing of module parameters.

For example:

```
signature class NodeSig;

signature module EdgeSig<NodeSig Node> {
  predicate apply(Node src, Node dst);
}

module Reachability<NodeSig Node, EdgeSig<Node> Edge> {
  Node reachableFrom(Node src) {
    Edge::apply+(src, result)
  }
}
```