Alloy Modules
Alloy Modules

- Alloys has a module system that allows the modularization and reuse of models.
- A module defines a model that can be incorporated as a submodel into another one.
- To facilitate reuse, modules may be parametric in one or more signatures.
Examples

```plaintext
module util/relation
-- r is acyclic over the set s
pred acyclic [r: univ->univ, s: set univ] {
  all x: s | x !in x.^r
}

module family
open util/relation as rel
sig Person {
  parents: set Person
}
fact { acyclic [parents, Person] }
```
module util/relation
-- r is acyclic over the set s
pred acyclic [r: univ->univ, s: set univ] {
    all x: s | x !in x.^r
}

module fileSystem
open util/relation as rel
sig Object {}
sig Folder extends Object {
    inside: set Folder
}
fact { acyclic [inside, Folder] }
Module Declarations

- The first line of every module is a module header

  \texttt{module modulePathName}

- The module can import another module with an open statement immediately following the header

  \texttt{open modulePathName}
Module Definition

- A module A can import a module B that can in turn import a module C, and so on
- You can understand open statements informally as textual inclusion
- No cycles in the import structure are permitted
Every module has a path name that must match the path of its corresponding file in the file system.

The module’s path name can range:
- from just the name of the file (without the .als extension)
- to the whole path from the root

The root of the path in the importing module header is the root of the path of every import.
ModulePathName definition

- Example:
  - A module: module relation
  - and a use in the family model
    module family
    open lib/people
  - If the path of family.als is p in the file system then the Alloy Analizer will search people.als in p/lib/
ModulePathName definition

- Example:
  - A module: `module graph`
  - and a use in the family model
    ```
    module myProject/family
    open lib/people
    ```
  - If the path of `myProject` is `p` in the file system then AA will search `people.als` in `p/lib/`
The modulePathName in the module header just specifies the root directory for every imported file.
Predefined Modules

- Alloy 4 comes with a library of predefined modules.
- Any imported module will actually be searched first among those modules.
  - Examples:
    - book/chapter2/addressBook1a
    - util/relation
    - examples/puzzles/farmer
- Failing that, the rules in the previous slides apply.
As

- When the path name of an import `include / (i.e. it is not just the name of a file but also a path)
- Then you **must** give a shorter name to the module with `as`

```
open util/relation as rel
```
Name Clashes

- Modules have their own namespaces
- If there is a name clash between components of different modules
- Then we use qualified names

module family
don open util/relation as rel
sig Person { parents: set Person }
fact { rel/acyclic [parents] }
Parametric Modules

- A model $m$ can be parametrized by one or more signature parameters $[x_1, \ldots, x_n]$
- Any importing module must instantiate each parameter with a signature name
- The effect of opening $m[S_1, \ldots, S_n]$ is that of importing a copy of $m$ with each signature parameter $x_1$ replaced by the signature name $S_1$
Parametric Modules Example

module graph [node] // 1 signature param
open util/relation as rel

pred dag [r: node -> node] {
  acyclic [r, node]
}

module family
open util/graph [Person] as g
sig Person { parents: set Person }
fact { dag [parents] }
The Predefined Module **Ordering**

- Creates a single linear ordering over the atoms in `elem`
  
  ```
  module util/ordering[elem]
  ```

- It also constrains all the atoms to exist that are permitted by the scope on `elem`. If the scope on a signature `S` is 5, opening `ordering[S]` will force `S` to have 5 elements and create a linear ordering over those five elements.
module util/ordering[elem]
private one sig Ord { 
    First, Last: elem,
    Next, Prev: elem -> lone elem
}

fact {
    // all elements of elem are totally
    // ordered
    elem in Ord.First.*Next
    ...
}
The Module Ordering

// constraints that actually define the // total order
Ord.Prev = ~(Ord.Next)
one Ord.First
one Ord.Last
no Ord.First.Prev
no Ord.Last.Next
The Module Ordering

{  
    // either elem has exactly one atom,
    // which has no predecessors or successors ...
    (one elem and no elem.(Ord.Prev) and
    no elem.(Ord.Next)) or
    // or ...
    all e: elem |
        // ... each element except the first has one 
        // predecessor, and ...
        (e = Ord.First or one e.(Ord.Prev)) and
        // ... each element except the last has one 
        // successor, and ...
        (e = Ord.Last or one e.(Ord.Next)) and
        // ... there are no cycles
        (e \in e.(Ord.Next))
    }

The Module Ordering

// first
fun first: one elem { Ord.First }

// last
fun last: one elem { Ord.Last }

// return the predecessor of e, or empty set if e is
// the first element
fun prev [e: elem]: lone elem { e.(Ord.Prev) }

// return the successor of e, or empty set of e is
// the last element
fun next [e: elem]: lone elem { e.(Ord.Next) }

// return elements prior to e in the ordering
fun prevs [e: elem]: set elem { e.^(Ord.Prev) }

// return elements following e in the ordering
fun nexts [e: elem]: set elem { e.^^(Ord.Next) }
The Module Ordering

// e1 is before e2 in the ordering
pred lt [e1, e2: elem] { e1 in prevs[e2] }

// e1 is after than e2 in the ordering
pred gt [e1, e2: elem] { e1 in nexts[e2] }

// e1 is before or equal to e2 in the ordering
pred lte [e1, e2: elem] { e1=e2 || lt [e1,e2] }

// e1 is after or equal to e2 in the ordering
pred gte [e1, e2: elem] { e1=e2 || gt [e1,e2] }
The Module Ordering

// returns the larger of the two elements in the // ordering
fun larger [e1, e2: elem]: elem
    { lt[e1,e2] => e2 else e1 }  

// returns the smaller of the two elements in the // ordering
fun smaller [e1, e2: elem]: elem
    { lt[e1,e2] => e1 else e2 }  

// returns the largest element in es // or the empty set if es is empty
fun max [es: set elem]: lone elem
    { es - es.^(Ord.Prev) }  

// returns the smallest element in es // or the empty set if es is empty
fun min [es: set elem]: lone elem
    { es - es.^(Ord.Next) }