Mastering Alloy

Alcino Cunha

Subset signatures

Subset signatures

- An arbitrary subset of a signature can be declared with keyword **`in`** instead of **`extends`**
- Subset signatures are not necessarily disjoint
- A signature can be declared as a subset of more than one signature
- Subset signatures cannot be extended
- Subset signatures can be used to simulate multiple-inheritance
- Atoms belonging to subset signatures are labelled in the visualizer

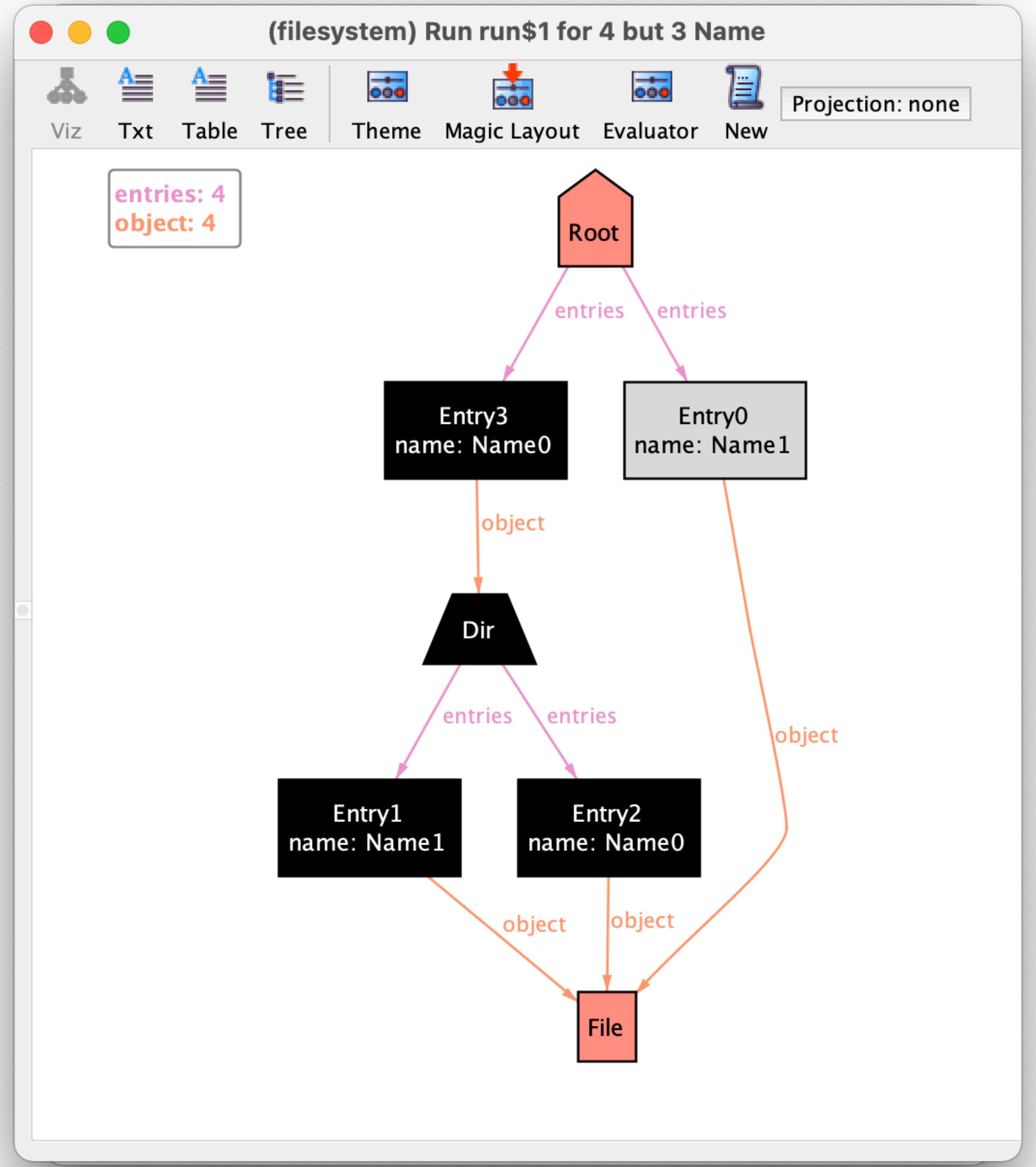
File-system

```
abstract sig Object {}
sig Dir extends Object {
  entries : set Entry
}
sig File extends Object {}
one sig Root extends Dir {}
sig Entry {
  object : one Object,
  name   : one Name
}
sig Name {}
```

Subset example

```
sig Trash in Object+Entry {}

fact {
  // The root cannot be trashed
  Root not in Trash
  // All other objects are trashed iff all entries
  // that point to them are trashed
  all o : Object-Root | o in Trash iff object.o in Trash
  // If a directory is trashed all its entries are trashed
  all d : Dir & Trash | d.entries in Trash
}
```

Visualizing subsets



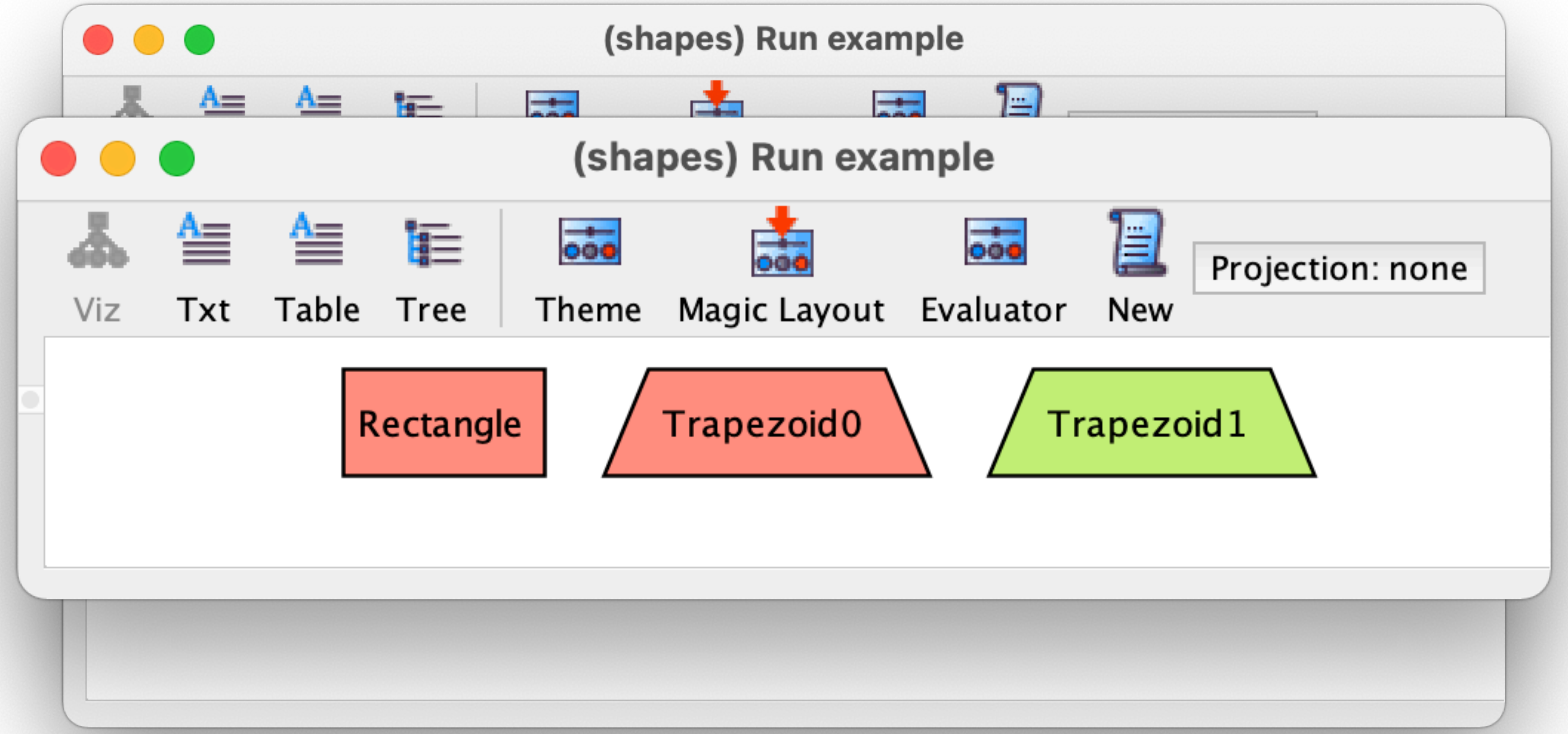
Subset example

```
abstract sig Shape {
  color : one Color
}
sig Rectangle, Trapezoid extends Shape {}

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

Subset example

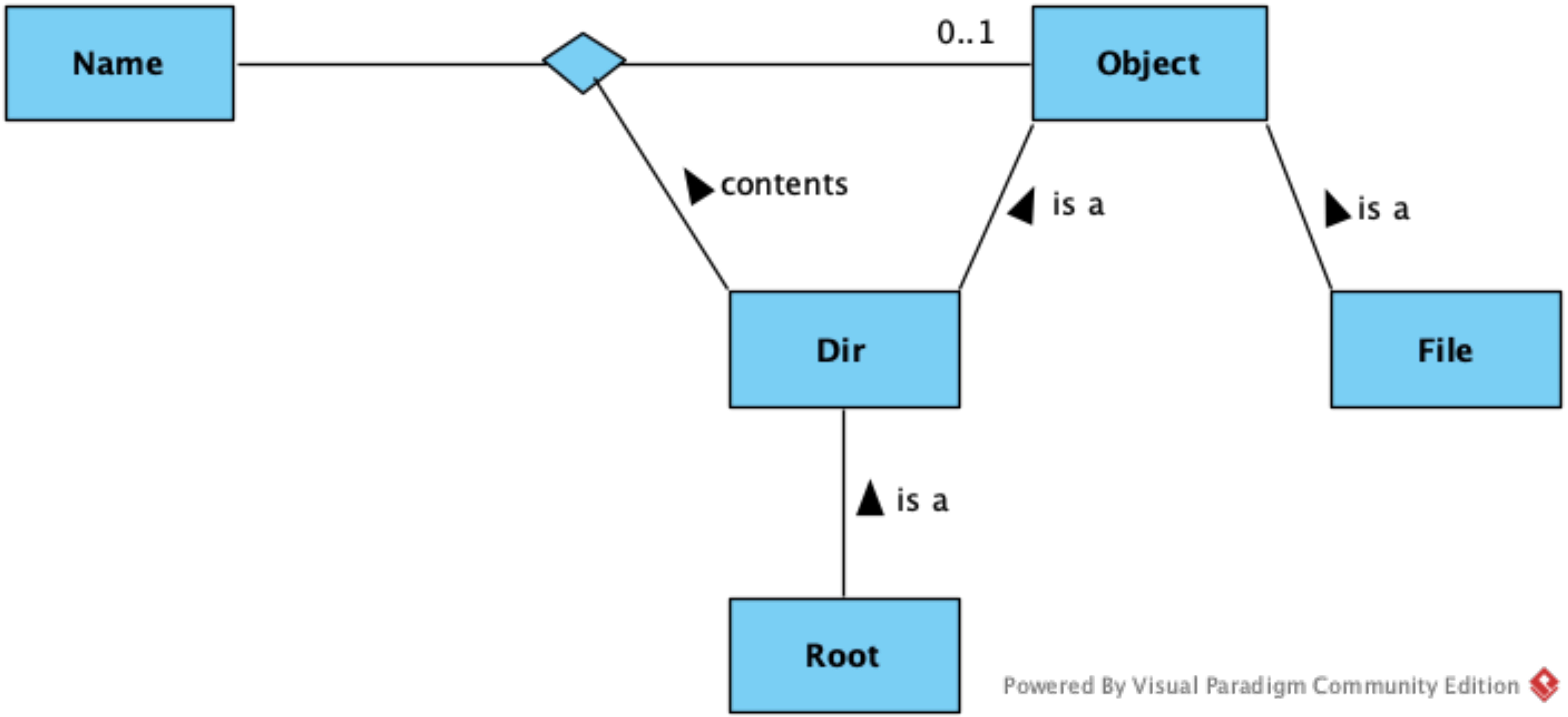
```
abstract sig Shape {}
sig Rectangle, Trapezoid extends Shape {}

sig Green, Red, Yellow in Shape {}

fact {
  Shape = Green+Red+Yellow
  no Green & Red
  no Red & Yellow
  no Green & Yellow
}
```

Visualizing subsets



N-ary relations

N-ary relationships *a la* UML



Ternary relation example

```
abstract sig Object {}
sig Dir extends Object {
  contents : Name -> lone Object
}
sig File extends Object {}
one sig Root extends Dir {}
sig Name {}
```

Composition

Root
Root

contents		
Root	Name0	Dir
Root	Name1	File
Dir	Name0	File
Dir	Name1	File

Root . contents	
Name0	Dir
Name1	File

Ternary relation example

```
fact {
   // All objects except the root are contained in at least one directory
   all o : Object - Root | some contents.o
   no contents.Root

   // All directories are contained in at most one directory
   all d : Dir | lone contents.d

   // A directory cannot be contained in itself
   all d : Dir | d not in d.^(???)
}
```

Comprehension

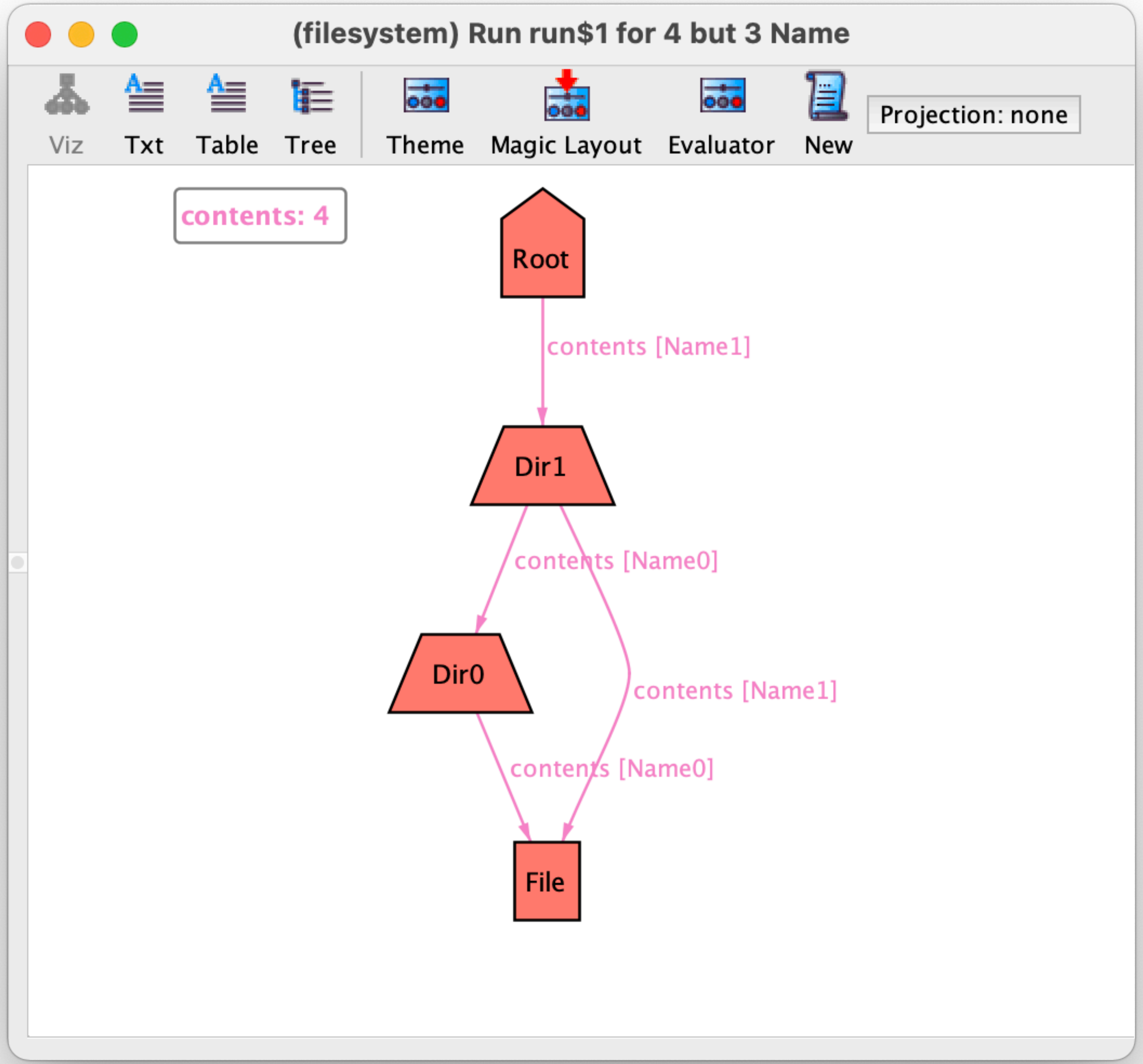
$$\{\ x_1 \mathbin{:} A_1 , \ldots , x_n \mathbin{:} A_n \ \mid \ \phi \ \}$$

$$\{\ x_1 \mathbin{:} A_1 , \ldots , x_n \mathbin{:} A_n \ \mid \ \phi \ \}\ (y_1, \ldots, y_n)$$
$$\leftrightarrow$$
$$A_1(y_1) \wedge \ldots \wedge A_n(y_n) \ \wedge \ \phi[x_1 \leftarrow y_1, \ldots, x_n \leftarrow y_n]$$

Ternary relation example

```
fact {
   // All objects except the root are contained in at least one directory
   all o : Object - Root | some contents.o
   no contents.Root

   // All directories are contained in at most one directory
   all d : Dir | lone contents.d

   // A directory cannot be contained in itself
   all d : Dir | d not in d.^({d : Dir, o : Object | some d.contents.o})
}
```

Visualizing N-ary relations



Overloading

Overloading

- Fields in disjoint signatures can be *overloaded* (have the same name)
- *Ambiguity* errors may occur

Overloading example

```
abstract sig Object {}
sig Dir extends Object {
   contents : set Entry
}
sig File extends Object {}
one sig Root extends Dir {}
sig Entry {
   contents : one Object,
   name     : one Name
}
sig Name {}
```

Overloading example

```
fact {
   // All objects except the root are contained in at least one entry
   all o : Object - Root | some contents.o
   no contents.Root

   // All directories are contained in at most one entry
   all d : Dir | lone contents.d

   // Different entries in a directory must have different names
   all d : Dir, n : Name | lone (d.contents & name.n)

   // A directory cannot be contained in itself
   all d : Dir | d not in d.^(contents.contents)
}
```

Ambiguity errors

```
run { some contents }
```

```
A type error has occurred:
This name is ambiguous due to multiple matches:
field this/Dir <: contents
field this/Entry <: contents
```

Resolving ambiguities

```
run { some d : Dir | some d.contents }

run { some contents & Dir->Entry }

run { some Dir <: contents }
```

Predicates and functions

Predicates

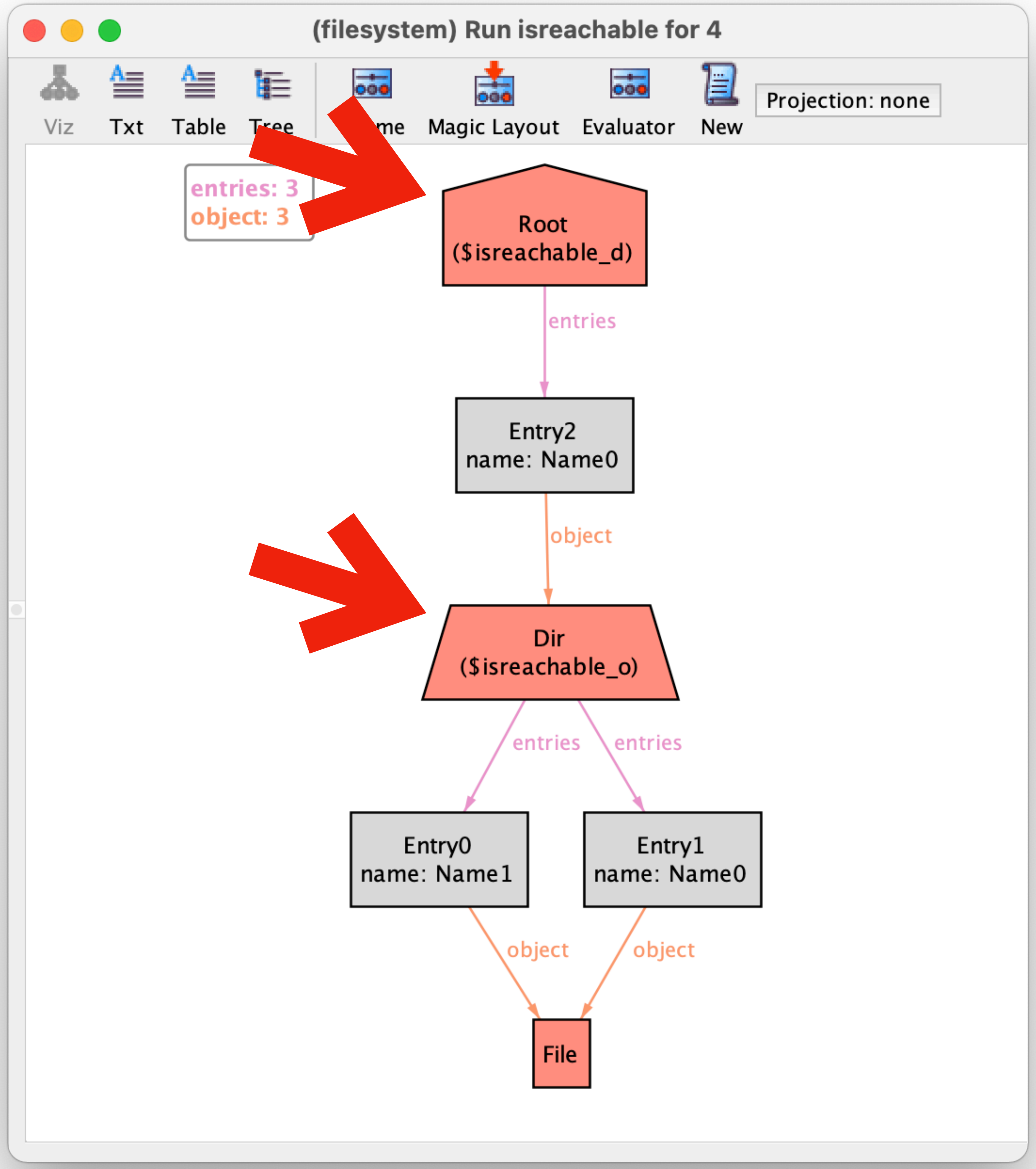
- *Predicates* are parametrized reusable constraints
  - Can also be derived propositions (without arguments)
  - Parameters can be arbitrary relations
- Only hold when invoked in a fact, command, or other predicates
- Recursive definitions are not allowed
- Run commands can directly ask for an instance satisfying a predicate
  - Atoms instantiating the parameters are shown in the visualizer

Predicate example

```
pred isreachable [d : Dir, o : Object] {
  o in d.^(entries.object)
}

fact {
  // A directory cannot be contained in itself
  all d : Dir | not isreachable[d,d]
}
```

Running a predicate

```
run isreachable for 4
```



Higher-order predicate example

```
pred acyclic [r : univ -> univ] {
  no ^r & iden
}

fact {
  // A directory cannot be contained in itself
  acyclic[entries.object]
}
```

Functions

- *Functions* are parametrized reusable expressions
  - Parameters can be arbitrary relations
- Functions without parameters can be used to define derived relations
  - These show up in the visualizer
- Recursive definitions are not allowed

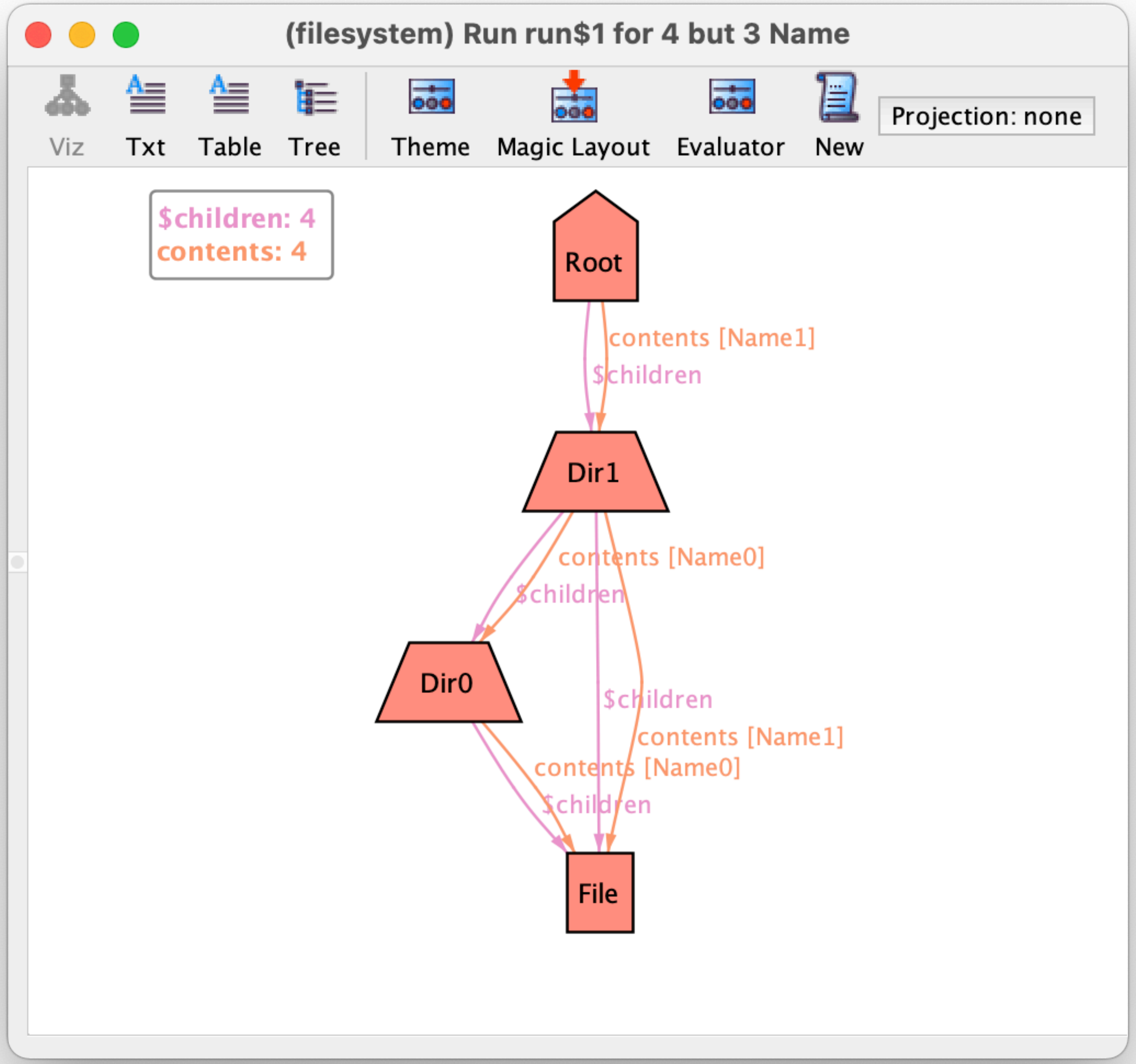
Function example

```
fun descendants [d : Dir] : set Object {
  d.^(entries.object)
}

fact {
  // A directory cannot be contained in itself
  all d : Dir | d not in descendants[d]
}
```

Derived relation example

```
sig Dir extends Object {
   contents : Name -> lone Object
}

fun children : Dir -> Object {
   { d : Dir, o : Object | some d.contents.o }
}

fact {
   // A directory cannot be contained in itself
   all d : Dir | d not in d.^children
}
```

Visualizing derived relations



Modules

Modules

- A model can be split into *modules*
- A module name is declared in the first line with keyword **`module`**
- A module can be imported with an **`open`** statement
- A module name must match the path of the corresponding file
- To disambiguate a call to an entity, the module name can be prepended
- An alias to a module name can be given with the **`as`** keyword in an **`open`** statement
- A module can be parametrised by one or more signatures

Module example

```
module relation

pred acyclic [r : univ -> univ] {
  no ^r & iden
}
```

Module example

```
open relation

fact {
  // A directory cannot be contained in itself
  acyclic[entries.object]
}
```

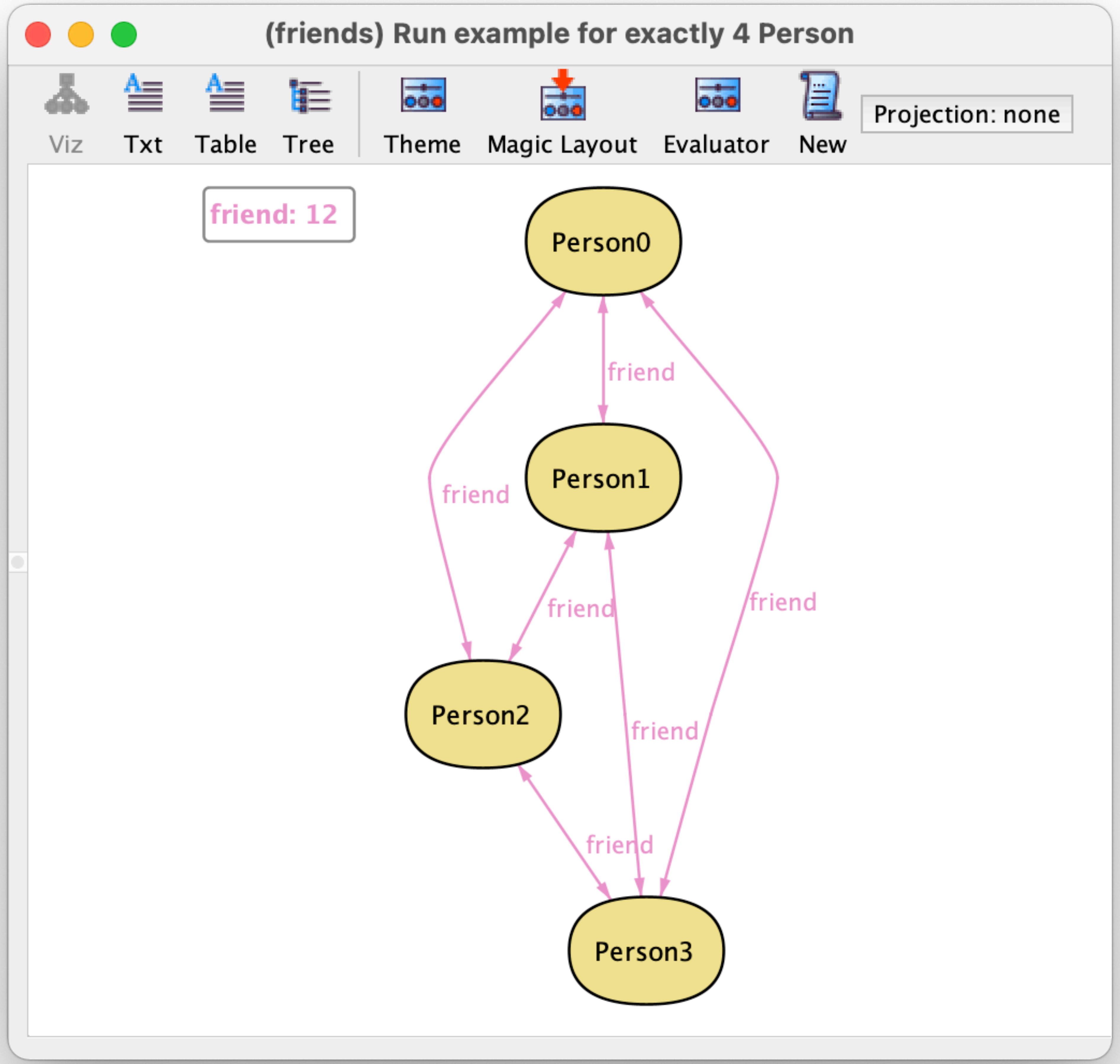
Parametrized module example

```
module graph[node]

pred complete[adj : node -> node] {
  all n : node | n.adj = node-n
}
```

Parametrized module example

```
open graph[Person]

sig Person {
  friend : set Person
}

fact { complete[friend] }
```

We are all friends



Predefined modules

`util/relation`	Useful functions and predicates for binary relations
`util/ternary`	Useful functions and predicates for ternary relations
`util/graph[A]`	Useful functions and predicates for graphs with nodes from signature *A*
`util/natural`	Natural numbers, including some arithmetic operations
`util/boolean`	Boolean type, including common logical connectives
`util/ordering[A]`	Imposes a total order on signature *A*

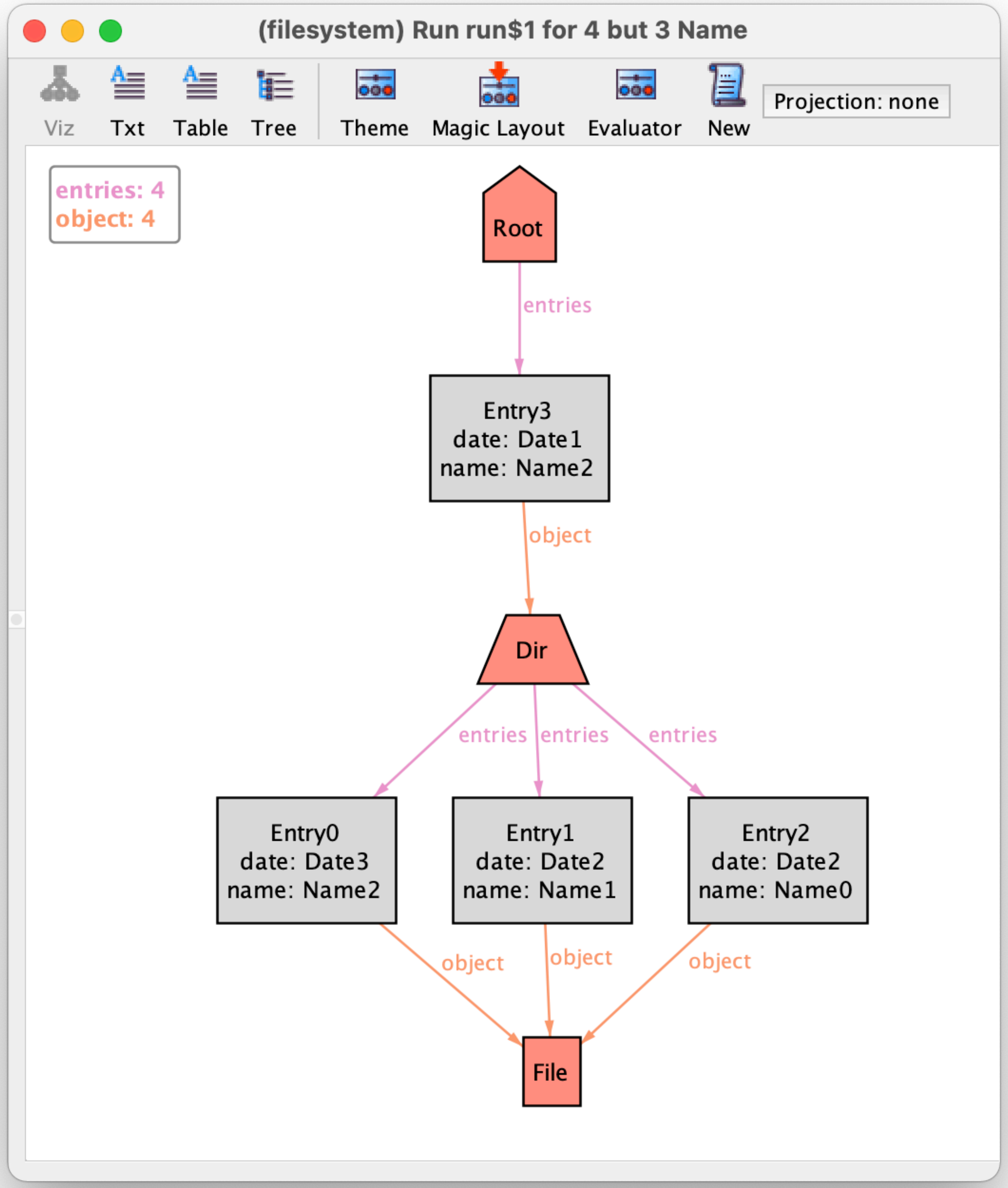
`util/ordering`

- Imposes a total order on the parameter signature
- For efficiency reasons the scope on that signature becomes exact
- Visualiser attempts to name atoms according to the order
- Many useful functions and relations, including
  - `next` and `prev` binary relations
  - `first` and `last` singleton sets
  - `lt`, `lte`, `gt`, and `gte` comparison predicates

`util/ordering` example

```
open util/ordering[Date]

sig Date {}
sig Entry {
   object : one Object,
   name   : one Name,
   date   : one Date
}

fact {
   // Entries inside a directory must have been created later
   all e : object.Dir, c : e.object.entries | lt[e.date, c.date]
}
```

util/ordering vizualisation



Type system

File-system

```
abstract sig Object {}
sig Dir extends Object {
  contents : set Entry
}
sig File extends Object {}
one sig Root extends Dir {}
sig Entry {
  contents : one Object,
  name    : one Name
}
sig Name {}
```

Arity error

```
run { some name & File }
```

```
A type error has occurred:
& can be used only between 2 expressions of the same arity.
Left type = {this/Entry->this/Name}
Right type = {this/File}
```

Ambiguity error

```
run { some contents }
```

```
A type error has occurred:
This name is ambiguous due to multiple matches:
field this/Dir <: contents
field this/Entry <: contents
```

Irrelevance warning

```
run { some Dir.name }
```

```
Warning #1
The join operation here always yields an empty set.
Left type = {this/Dir}
Right type = {this/Entry->this/Name}
```

Type system

- The main goal of Alloy's type system is to detect *irrelevant* expressions
- An expression is irrelevant if it can be replaced by **`none`**
- The same type system can be used to resolve overloading
  - An overloaded name is treated as the union of all respective relations
  - Only one of the overloaded relations must be relevant

Types

- The type of an expression is a set of tuples of *atomic* types
  - An atomic type is a signature that is not further extended
- For non abstract signatures we need a *reminder* type
  - The reminder contains all atoms not contained in one of the extensions
  - The reminder type of signature A is denoted as \$$A$
- The type of an expression is an upper-bound on its value
  - If the type of an expression is empty, the expression is irrelevant

Type inference

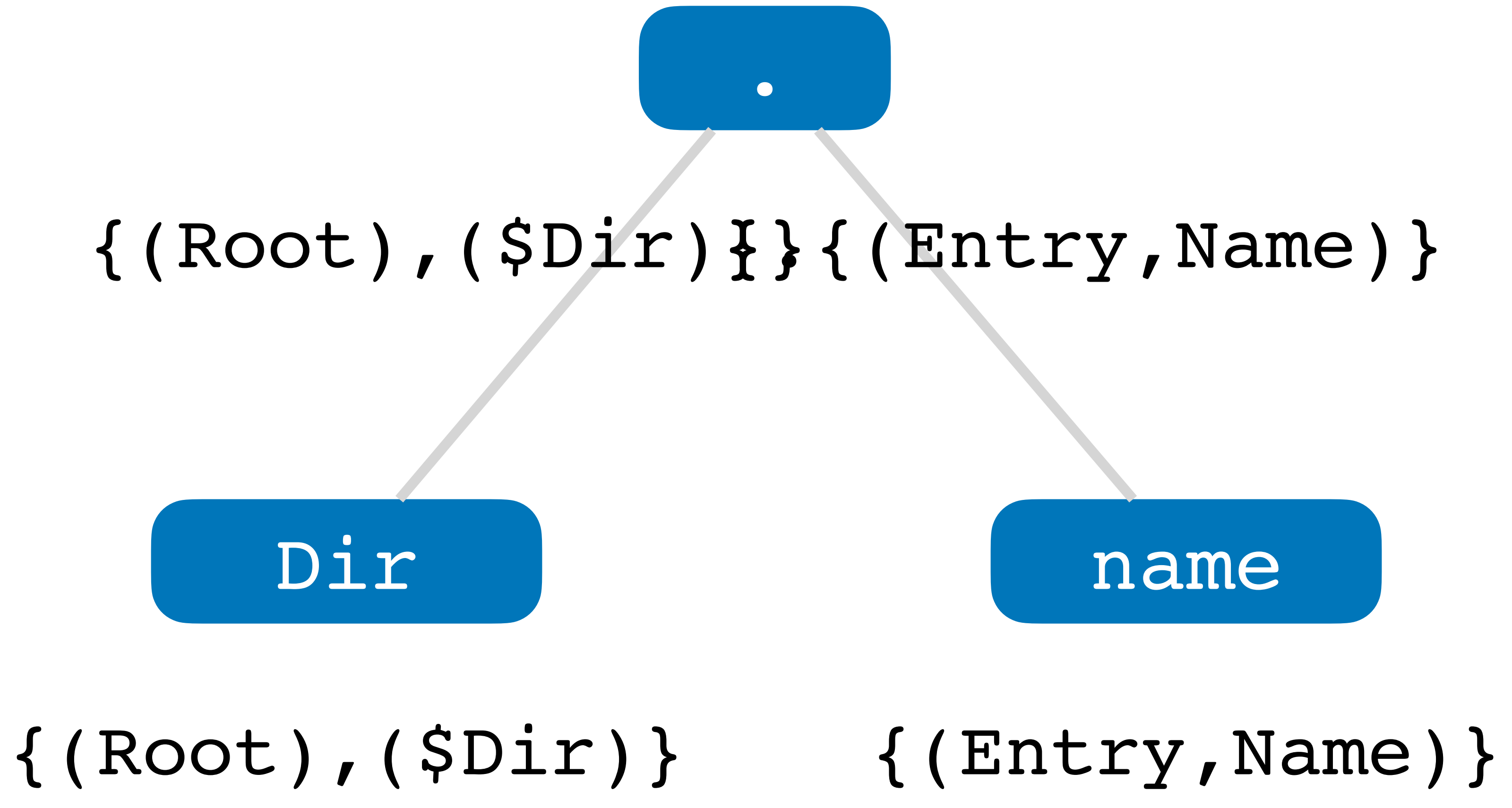
- Type inference is guided by the abstract syntax and works in two phases
  - A bottom-up phase computes the *bounding* types
  - A top-down phase refines these and computes the *relevance* types
- Unlike bounding types, relevance types depend on the context
  - The same expression in different formulas may have different types

Bounding type inference

- The bounding type of a signature or field is inferred from the respective declarations
- The bounding type of an expression is computed using the same relational operator applied to the bounding types of sub-expressions
    - This is possible because types are also relations

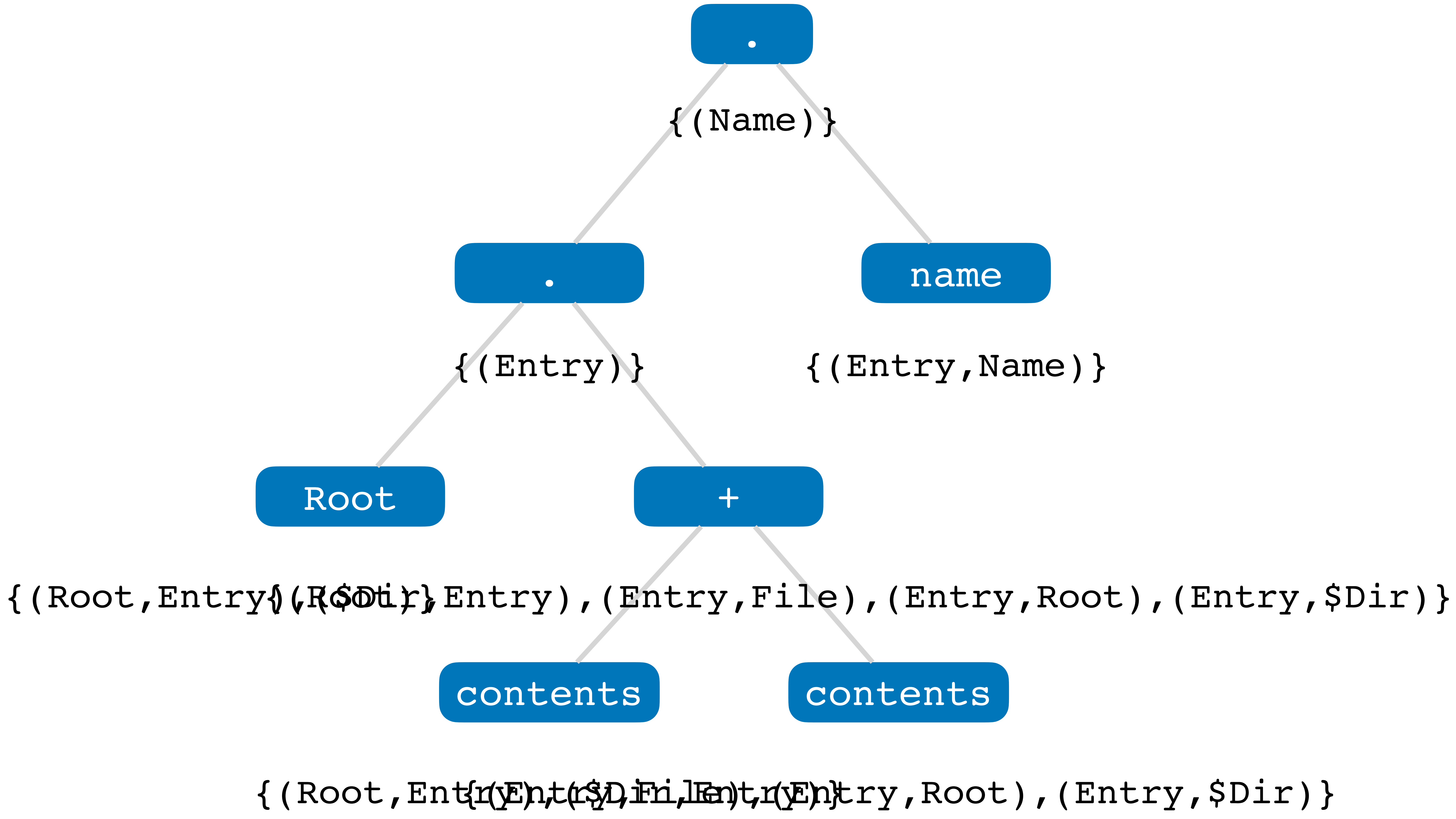
Bounding type inference

`Dir.name`



Bounding type inference

(Root.content).name

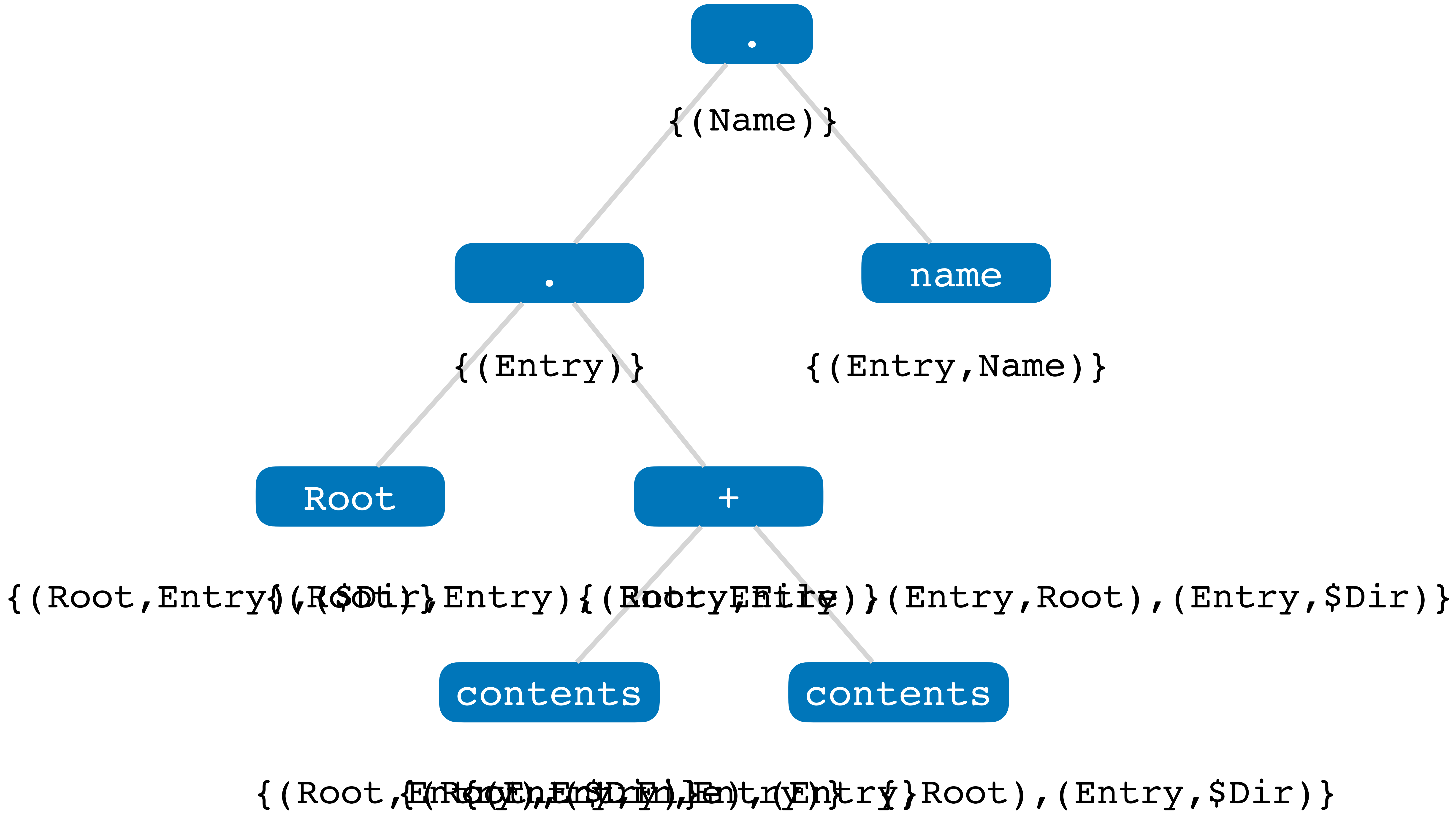


Relevance type inference

- The relevance type of the top expression is equal to its bounding type
- The relevance type of a sub-expression is computed by determining which tuples effectively contributed to the parent expression type

Relevance type inference

(Root.content).name



Relational model finding

Architecture

Kodkod

- A Kodkod problem consists of
  - A universe of atoms $\mathscr{U}$
  - A set of relation declarations of shape $r :_a \; r_L \, r_U$
    - a is the *arity* of r
    - r_L is the *lower-bound* of r, tuples that MUST be present in r
    - r_U is the *upper-bound* of r, tuples that MAY be present in r
  - A relational logic formula ϕ

Kodkod

- Kodkod is a *relational model finder*
- It finds a valuation (a model) for the relations such that
  - ϕ is true in that model
  - the valuation of each relation r complies with the *partial-knowledge* declared in the bounds

Alloy ⇄ Kodkod

- Alloy assertions to be checked are negated and conjoined with the facts in the Kodkod formula
  - An assertion is valid if its negation is unsatisfiable
- Alloy fields and atomic signatures are declared in the Kodkod problem
  - Non-atomic signatures are aliased to a disjunction of atomic ones
- Appropriate bounds are inferred from scopes
  - Upper-bounds can be shared between related atomic signatures
    - Further constraints must be added to ensure a sound structural semantics
  - Kodod atom names are meaningless
    - When building an Alloy instance from a Kodkod instance atoms are renamed

Alloy example

```
abstract sig Object {}
sig Dir extends Object {
  entries : set Entry
}
sig File extends Object {}
one sig Root extends Dir {}
sig Entry {}

run { some entries.Entry } for 3 but 2 Entry
```

Kodkod translation

```
{A,B,C,D,E}

$Dir   :1 {} {(A),(B)}
File   :1 {} {(A),(B)}
Root   :1 {(C)} {(C)}
Entry  :1 {} {(D),(E)}
entries :2 {} {(A,D),(A,E),(B,D),(B,E),(C,D),(C,E)}

no File & $Dir
all x : $Dir+Root | x.entries in Entry
entries.univ in $Dir+Root
some entries.Entry
```

Kodkod $\rightleftarrows$ SAT

- A relation r of arity a can be represented by a matrix of boolean variables of size $|\mathscr{U}|^a$

$$r[i_1, \ldots, i_a] = \begin{cases} \top & \text{if } (\mathscr{U}_{i_1}, \ldots, \mathscr{U}_{i_a}) \in r_L \\ r_{i_1, \ldots, i_a} & \text{if } (\mathscr{U}_{i_1}, \ldots, \mathscr{U}_{i_a}) \in r_U \backslash r_L \\ \bot & \text{otherwise} \end{cases}$$

Kodkod $\rightleftarrows$ SAT

$$\texttt{\$Dir} = \begin{bmatrix} d_A \\ d_B \\ \bot \\ \bot \\ \bot \end{bmatrix} \quad \texttt{File} = \begin{bmatrix} f_A \\ f_B \\ \bot \\ \bot \\ \bot \end{bmatrix} \quad \texttt{Root} = \begin{bmatrix} \bot \\ \bot \\ \top \\ \bot \\ \bot \end{bmatrix} \quad \texttt{Entry} = \begin{bmatrix} \bot \\ \bot \\ \bot \\ e_D \\ e_E \end{bmatrix}$$

$$\texttt{entries} = \begin{bmatrix} \bot & \bot & \bot & r_{A,D} & r_{A,E} \\ \bot & \bot & \bot & r_{B,D} & r_{B,E} \\ \bot & \bot & \bot & r_{C,D} & r_{C,E} \\ \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot \end{bmatrix}$$

Kodkod ⇄ SAT

- Relational operators are implemented by matrix operations (in the boolean semiring)

`.`	Multiplication
`+`	Addition
`&`	Hadamard product
…	

- Atomic formulas originate propositional formulas

`in`	Conjunction of point-wise implication
`some`	Disjunction
…	

Kodkod $\rightleftarrows$ SAT

some `entries.Entry`

$$\textbf{some} \begin{bmatrix} \bot & \bot & \bot & r_{A,D} & r_{A,E} \\ \bot & \bot & \bot & r_{B,D} & r_{B,E} \\ \bot & \bot & \bot & r_{C,D} & r_{C,E} \\ \bot & \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot & \bot \end{bmatrix} \cdot \begin{bmatrix} \bot \\ \bot \\ \bot \\ e_D \\ e_E \end{bmatrix}$$

$$\textbf{some} \begin{bmatrix} (r_{A,D} \wedge e_D) \vee (r_{A,E} \wedge e_E) \\ (r_{B,D} \wedge e_D) \vee (r_{B,E} \wedge e_E) \\ (r_{C,D} \wedge e_D) \vee (r_{C,E} \wedge e_E) \\ \bot \\ \bot \end{bmatrix}$$

$$(r_{A,D} \wedge e_D) \vee (r_{A,E} \wedge e_E) \vee (r_{B,D} \wedge e_D) \vee (r_{B,E} \wedge e_E) \vee (r_{C,D} \wedge e_D) \vee (r_{C,E} \wedge e_E)$$

Quantifiers

- Since the universe is finite quantifiers can be handled by expansion

```
all x : Entry | some entries.x
≡
some entries.{(D)} and some entries.{(E)}
```

- Unfortunately this yields no witnesses to existential quantifiers

```
some x : Entry | some entries.x
≡
some entries.{(D)} or some entries.{(E)}
```

Skolemization

- Skolemization replaces existentially quantified variables by free variables
  - Free variables are implicitly existentially quantified
  - Generates smaller but equisatisfiable formulas
  - Skolemized variables are witnesses that can be shown in the visualizer

some x : Entry | **some** entries.x

$\equiv$

\$x $:_1$ {} {(D),(E)}

one \$x **and some** entries.\$x

Symmetry breaking

- Kodkod performs several optimizations to decrease SAT complexity
- The most significant is symmetry breaking
  - Since atoms are uninterpreted isomorphic instances are equivalent
  - To avoid returning isomorphic instances a symmetry breaking formula is conjoined to the problem formula
  - For efficiency reasons the technique is not complete
- Besides increasing efficiency symmetry breaking is also useful for validation
  - Otherwise the user would be overwhelmed with isomorphic instances