α**Rby** : An Embedding of Alloy in Ruby

Aleksandar Milicevic, Ido Efrati, and Daniel Jackson
{aleks,idoe,dnj}@csail.mit.edu

What exactly is this embedding?

What exactly is this embedding?

- alloy API for ruby?

What exactly is this embedding?

- ~~alloy API for ruby?~~ ✗
- alloy-backed constraint solver for ruby?

What exactly is this embedding?

- ~~alloy API for ruby?~~ ✗
- ~~alloy-backed constraint solver for ruby?~~ ✗
- alloy-like syntax in ruby (embedded DSL)?

What exactly is this embedding?

- ~~alloy API for ruby?~~ ✗
- ~~alloy-backed constraint solver for ruby?~~ ✗
- alloy-like syntax in ruby (embedded DSL)? ✔

```
abstract sig Person {
  father: lone Man,
  mother: lone Woman
}
sig Man extends Person {
  wife: lone Woman
}
sig Woman extends Person {
  husband: lone Man
}
fact TerminologyBiology {
  wife = ~husband
  no p: Person |
    p in p.^(mother + father)
}
```

```
abstract sig Person [
  father: (lone Man),
  mother: (lone Woman)
]
sig Man extends Person [
  wife: (lone Woman)
]
sig Woman extends Person [
  husband: (lone Man)
]
fact terminology_biology {
  wife == ~husband and
  no(p: Person) {
    p.in? p.^(mother + father)
  }
}
```

What exactly is this embedding?

- alloy API for ruby? ✔
- alloy-backed constraint solver for ruby? ✔
- alloy-like syntax in ruby (embedded DSL)? ✔

```
abstract sig Person {
  father: lone Man,
  mother: lone Woman
}
sig Man extends Person {
  wife: lone Woman
}
sig Woman extends Person {
  husband: lone Man
}
fact TerminologyBiology {
  wife = ~husband
  no p: Person |
    p in p.^(mother + father)
}
```

```
abstract sig Person [
  father: (lone Man),
  mother: (lone Woman)
]
sig Man extends Person [
  wife: (lone Woman)
]
sig Woman extends Person [
  husband: (lone Man)
]
fact terminology_biology {
  wife == ~husband and
  no(p: Person) {
    p.in? p.^(mother + father)
  }
}
```

Why Embedding?

main goal: full-blown imperative shell around alloy

Why Embedding?

main goal: full-blown imperative shell around alloy

- retain the same alloy modeling environment
- write and analyze the same old alloy models

Why Embedding?

main goal: full-blown imperative shell around alloy

- retain the same alloy modeling environment
- write and analyze the same old alloy models
- add general-purpose scripting layer around it

Why Scripting?

Why Scripting?

practical reasons

- automate multiple model finding tasks
- pre-processing (e.g., prompt for analysis parameters)
- post-processing (e.g., display the results of the analysis)
- build tools more easily

Why Scripting?

practical reasons

- automate multiple model finding tasks
- pre-processing (e.g., prompt for analysis parameters)
- post-processing (e.g., display the results of the analysis)
- build tools more easily

```
s = SudokuModel::Sudoku.parse("0,0,1; 0,3,4; 3,1,1; 2,2,3")
s.solve    # invokes Alloy to solve the sudoku embodied in ‘s‘
s.display  # draws some fancy graphical grid displaying the solution
```

Why Scripting?

practical reasons

- automate multiple model finding tasks
- pre-processing (e.g., prompt for analysis parameters)
- post-processing (e.g., display the results of the analysis)
- build tools more easily

```
s = SudokuModel::Sudoku.parse("0,0,1; 0,3,4; 3,1,1; 2,2,3")
s.solve    # invokes Alloy to solve the sudoku embodied in ‘s‘
s.display  # draws some fancy graphical grid displaying the solution
```

fundamental reasons

- quest for a synergy between imperative and declarative

Why Scripting?

practical reasons

- automate multiple model finding tasks
- pre-processing (e.g., prompt for analysis parameters)
- post-processing (e.g., display the results of the analysis)
- build tools more easily

```
s = SudokuModel::Sudoku.parse("0,0,1; 0,3,4; 3,1,1; 2,2,3")
s.solve    # invokes Alloy to solve the sudoku embodied in ‘s‘
s.display  # draws some fancy graphical grid displaying the solution
```

fundamental reasons

- quest for a synergy between imperative and declarative
- imperative generation of declarative specifications
  - → can this change the way we write specifications?
  - → can this simplify specification languages?

Why Scripting?

practical reasons

- automate multiple model finding tasks
- pre-processing (e.g., prompt for analysis parameters)
- post-processing (e.g., display the results of the analysis)
- build tools more easily

```
s = SudokuModel::Sudoku.parse("0,0,1; 0,3,4; 3,1,1; 2,2,3")
s.solve    # invokes Alloy to solve the sudoku embodied in ‘s‘
s.display  # draws some fancy graphical grid displaying the solution
```

fundamental reasons

- quest for a synergy between imperative and declarative
- imperative generation of declarative specifications
  - → can this change the way we write specifications?
  - → can this simplify specification languages?

not studied as much

Implementation Choices

Implementation Choices

- extend alloy with a new programming language around it
  - → challenge: a lot of engineering
  - → potential drawbacks: generality, lack of existing libraries

Implementation Choices

- extend alloy with a new programming language around it
  - → challenge: a lot of engineering
  - → potential drawbacks: generality, lack of existing libraries
- recreate the alloy modeling environment in an existing language
  - → challenges:
    - achieving alloy’s relational semantics
    - achieving alloy’s “non-standard” operators
    - achieving alloy’s complex syntax
    - **reconcile two different paradigms**

Implementation Choices

- extend alloy with a new programming language around it
  - → challenge: a lot of engineering
  - → potential drawbacks: generality, lack of existing libraries
- recreate the alloy modeling environment in an existing language
  - → challenges:
    - achieving alloy's relational semantics
    - achieving alloy's "non-standard" operators
    - achieving alloy's complex syntax
    - **reconcile two different paradigms**

αRby

αRby by example: **Sudoku**

Example: Sudoku in α**Rby**

```
alloy :SudokuModel do


 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 # ...
end
```

Example: Sudoku in α**Rby**

```
alloy :SudokuModel do


 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # each row contains 1..N
  # each column contains 1..N
  # each matrix contains 1..N
 }
end
```

Example: Sudoku in α**Rby**

```
alloy :SudokuModel do


 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # each row contains 1..N
  # each column contains 1..N
  # each matrix contains 1..N
 }
end
```

1. translates ruby to classes/methods

```
module SudokuModel

  class Sudoku < Arby::Ast::Sig
    attr_accessor :grid
  end

  def self.solved(s)
    # exactly the same body in the
    # spec as on the left
  end
end
```

Example: Sudoku in α**Rby**

```
alloy :SudokuModel do


 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # each row contains 1..N
  # each column contains 1..N
  # each matrix contains 1..N
 }
end
```

2. can be used in regular OOP
   - monkey patch classes with utility methods

```
class SudokuModel::Sudoku
  def display
    puts grid # or draw fancy grid
  end

  def self.parse(str)
    Sudoku.new grid:
      str.split(/;\s*/).map{ |x|
        x.split(/,/).map(&:to_i) }
  end
end
```

   - create objects, get/set fields, call methods

```
s = SudokuModel::Sudoku.new
s.grid = [[0, 0, 1], [1, 3, 2]]
puts s.grid
s = SudokuModel::Sudoku.parse(
      "0,0,1; 0,3,4; 3,1,1; 2,2,3")
s.display
```

Sudoku in α**Rby**: Mixed Execution

```
alloy :SudokuModel do


 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # each row contains 1..N
  # each column contains 1..N
  # each matrix contains 1..N
 }
end
```

- ***goal***: parameterize the spec by sudoku size

Sudoku in α**Rby**: Mixed Execution

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # each row contains 1..N
  # each column contains 1..N
  # each matrix contains 1..N
 }
end
```

- ***goal***: parameterize the spec by sudoku size

3. specification parameterized by sudoku size

Sudoku in α**Rby**: Mixed Execution

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

- ***goal***: parameterize the spec by sudoku size

3. specification parameterized by sudoku size

4. mixed concrete and symbolic execution
   - the **spec** is the **return value** of the method
   - special αRby methods return symbolic values (e.g., **`all`**, overloaded operators, ...)
   - everything else executes concretely
   - executed lazily:

     this ruby code

     ```
     SudokuModel.N = 4
     puts SudokuModel.to_als
     ```

     and this code

     ```
     SudokuModel.N = 9
     puts SudokuModel.to_als
     ```

     produce different alloy specifications.

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

- ***goal***: shrink bounds to enforce the partial solution known upfront (the pre-filled Sudoku cells)

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]         = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

create empty bounds

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]         = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

compute indexes of empty cells

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]          = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

exact bound for `Sudoku`: exactly `self`

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]            = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

lower bound for `grid`: must include the filled cells

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]         = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

upper bound for `grid`:
may include `(1..N)` for all empty cells

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]           = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]          = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

only ints from 0 to N

Sudoku in α**Rby**: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

if SAT, automatically updates all “sig class” objects used as part of the partial instance →

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]           = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

Sudoku in αRby: Partial Instance

```
alloy :SudokuModel do
 SudokuModel::N = 9

 sig Sudoku [
  # cell coordinate -> cell value
  grid: Int ** Int ** (lone Int)
 ]

 pred solved[s: Sudoku] {
  # concrete
  m   = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N) {
   s.grid[r][Int] == (1..N) and
   s.grid[Int][r] == (1..N)
  } and
  all(c, r: 0...m) {
   s.grid[rng[c]][rng[r]] == (1..N)
  }
 }
end
```

5. solving with partial instance

```
class SudokuModel::Sudoku
 def pi
  b = Arby::Ast::Bounds.new
  inds = (0...N)**(0...N) -
         self.grid.project(0..1)
  b[Sudoku]         = self
  b.lo[Sudoku.grid] = self**self.grid
  b.hi[Sudoku.grid] = self**inds**(1..N)
  b.bound_int(0..N)
 end
 def solve
  # satisfy pred solved given partial inst
  SudokuModel.solve :solved, self.pi
 end
end
```

6. continue to use as regular OOP

```
s = SudokuModel::Sudoku.parse(
      "0,0,1; 0,3,4; 3,1,1; 2,2,3")
s.solve; s.display
```

Sudoku in α**Rby**: Staged Execution

goal: generate a minimal sudoku puzzle

Sudoku in α**Rby**: Staged Execution

goal: generate a minimal sudoku puzzle

```
def min(sudoku)
  # ...
end

s = Sudoku.new(); s.solve(); s = min(s);
puts "local minimum: #{s.grid.size}"
```

start with empty sudoku, solve it, then minimize it

Sudoku in α**Rby**: Staged Execution

goal: generate a minimal sudoku puzzle

```
def dec(s, order=Array(0...s.grid.size).shuffle)
 # ...
end









def min(sudoku)
 (s1 = dec(sudoku)) ? min(s1) : sudoku
end

s = Sudoku.new(); s.solve(); s = min(s);
puts "local minimum: #{s.grid.size}"
```

try to decrement;
if successful minimize the result,
otherwise return the input sudoku

Sudoku in *α***Rby**: Staged Execution

goal: generate a minimal sudoku puzzle

```
def dec(s, order=Array(0...s.grid.size).shuffle)
 return nil if order.empty?
 # remove a cell, then re-solve
 s_dec = Sudoku.new grid:
           s.grid.delete_at(order.first)
 sol   = s_dec.clone.solve()
 # check if unique
 if sol.satisfiable? && !sol.next.satisfiable?
  s_dec   # return decremented sudoku
 else   # try deleting some other cell
  dec(s, order[1..-1])
 end
end

def min(sudoku)
 (s1 = dec(sudoku)) ? min(s1) : sudoku
end

s = Sudoku.new(); s.solve(); s = min(s);
puts "local minimum: #{s.grid.size}"
```

← pick a cell to remove and check if the new sudoku has a unique solution;

keep trying until run out of cells;

Sudoku in α**Rby**: Staged Execution

goal: generate a minimal sudoku puzzle

```
def dec(s, order=Array(0...s.grid.size).shuffle)
 return nil if order.empty?
 # remove a cell, then re-solve
 s_dec = Sudoku.new grid:
           s.grid.delete_at(order.first)
 sol   = s_dec.clone.solve()
 # check if unique
 if sol.satisfiable? && !sol.next.satisfiable?
  s_dec   # return decremented sudoku
 else   # try deleting some other cell
  dec(s, order[1..-1])
 end
end

def min(sudoku)
 (s1 = dec(sudoku)) ? min(s1) : sudoku
end

s = Sudoku.new(); s.solve(); s = min(s);
puts "local minimum: #{s.grid.size}"
```

pick a cell to remove and check if the new sudoku has a unique solution;

keep trying until run out of cells;

solve next to check for uniqueness

Sudoku in α**Rby**: Staged Execution

goal: generate a minimal sudoku puzzle

```
def dec(s, order=Array(0...s.grid.size).shuffle)
 return nil if order.empty?
 # remove a cell, then re-solve
 s_dec = Sudoku.new grid:
           s.grid.delete_at(order.first)
 sol   = s_dec.clone.solve()
 # check if unique
 if sol.satisfiable? && !sol.next.satisfiable?
  s_dec   # return decremented sudoku
 else   # try deleting some other cell
  dec(s, order[1..-1])
 end
end

def min(sudoku)
 (s1 = dec(sudoku)) ? min(s1) : sudoku
end

s = Sudoku.new(); s.solve(); s = min(s);
puts "local minimum: #{s.grid.size}"
```

pick a cell to remove and check if the new sudoku has a unique solution;

keep trying until run out of cells;

solve next to check for uniqueness

uses the previous solution to search for a new (smaller) one

αRby Implementation Tricks

Reconciling Alloy and Ruby

- **alloy**: declarative, relational, based on FOL
- **ruby**: imperative, non-relational, object-oriented

Reconciling Alloy and Ruby

- **alloy**: declarative, relational, based on FOL
- **ruby**: imperative, non-relational, object-oriented

structures	modules sigs fields predicates	modules classes attributes methods

Reconciling Alloy and Ruby

- **alloy**: declarative, relational, based on FOL
- **ruby**: imperative, non-relational, object-oriented

structures	modules sigs fields predicates	modules classes attributes methods
syntax	“non-standard” operators	very liberal parser, can accommodate most cases

Reconciling Alloy and Ruby

- **alloy**: declarative, relational, based on FOL
- **ruby**: imperative, non-relational, object-oriented

structures	modules sigs fields predicates	modules classes attributes methods
syntax	“non-standard” operators	very liberal parser, can accommodate most cases
semantics	everything is a relation	“monkey patch” relevant ruby classes to make them look like relations

Reconciling Alloy and Ruby: Syntax

Reconciling Alloy and Ruby: Syntax

description	**Alloy**	**αRby**
equality	`x = y`	`x == y`

Reconciling Alloy and Ruby: Syntax

description	Alloy	αRby
equality	`x = y`	`x == y`
sigs and fields		

```
sig S {
  f: lone S -> Int
} {
  some f
}
```

```
sig S [
  f: lone(S) ** Int
] {
  some f
}
```

Reconciling Alloy and Ruby: Syntax

description	**Alloy**	**αRby**
equality	`x = y`	`x == y`
sigs and fields		

```
sig S {
  f: lone S -> Int
} {
  some f
}
```

```
sig S [
  f: lone(S) ** Int
] {
  some f
}
```

quantifiers		

```
all s: S {
  p1[s]
  p2[s]
}
```

```
all(s: S) {
  p1[s] and
  p2[s]
}
```

Reconciling Alloy and Ruby: Syntax

description	Alloy	αRby
equality	`x = y`	`x == y`

sigs and fields

```
sig S {
  f: lone S -> Int
} {
  some f
}
```

```
sig S [
  f: lone(S) ** Int
] {
  some f
}
```

quantifiers

```
all s: S {
  p1[s]
  p2[s]
}
```

```
all(s: S) {
  p1[s] and
  p2[s]
}
```

description	Alloy	αRby
fun return type declaration	`fun f[s: S]: set S {}`	`fun f[s: S][set S] {}`
set comprehension	`{s: S | p1[s]}`	`S.select{|s| p1(s)}`

illegal Ruby operators	`x in y, x !in y`
	`x !> y`
	`x -> y`
	`x . y`
	`#x`
	`x => y`
	`x => y else z`
	`S <: f, f >: Int`
		`x.in?(y), x.not_in?(y)`
		`not x > y`
		`x ** y`
		`x.(y)`
		`x.size`
		`y if x`
		`if x then y else z`
		`S.< f, f.> Int`

operator arity mismatch	`^x, *x`	`x.closure, x.rclosure`

Achieving Syntax: α**Rby** Builder Methods

how is this parsed by ruby?

```
abstract sig Person [ father: (lone Man), mother: (lone Woman) ] { <facts> }
```

Achieving Syntax: αRby Builder Methods

how is this parsed by ruby?

```
abstract sig Person [ father: (lone Man), mother: (lone Woman) ] { <facts> }
             ^
             Module#const_missing(:Person) ⟶ builder
```

legend:

- blue identifiers: method names implemented or overridden by αRby
- *red identifiers*: objects exchanged between methods

Achieving Syntax: *α***Rby** Builder Methods

how is this parsed by ruby?

```
abstract sig Person [ father: (lone Man), mother: (lone Woman) ] { <facts> }
             ^      |
             Module#|onst_missing(:Person) ⟶ builder
                    builder.send :[], {father: ...}, &proc{<facts>} ⟶ builder
```

legend:

- blue identifiers: method names implemented or overridden by *α*Rby
- *red identifiers*: objects exchanged between methods

Achieving Syntax: α**Rby** Builder Methods

how is this parsed by ruby?

```
abstract sig Person [ father: (lone Man), mother: (lone Woman) ] { <facts> }
         |   ^      |
         |   Module#const_missing(:Person) ⟶ builder
         |          builder.send :[], {father: ...}, &proc{<facts>} ⟶ builder
         sig(builder) ⟶ sigBuilder
```

legend:

- blue identifiers: method names implemented or overridden by αRby
- *red identifiers*: objects exchanged between methods

Achieving Syntax: αRby Builder Methods

how is this parsed by ruby?

```
abstract sig Person [ father: (lone Man), mother: (lone Woman) ] { <facts> }
```

```
|        |   ^      |
|        |   Module#const_missing(:Person) ⟶ builder
|        |          builder.send :[], {father: ...}, &proc{<facts>} ⟶ builder
|        sig(builder) ⟶ sigBuilder
abstract(sigBuilder) ⟶ sigBuilder
```

legend:

- blue identifiers: method names implemented or overridden by αRby
- *red identifiers*: objects exchanged between methods

Symbolic by Concrete Execution

goal

- translate αRby programs to (symbolic) alloy models

Symbolic by Concrete Execution

goal

- translate αRby programs to (symbolic) alloy models

approach

- run αRby programs using the standard ruby interpreter
- the return value is the symbolic result

```
pred solved[s: Sudoku] {
  # concrete
  m = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N)    { s.grid[r][Int] == (1..N) && s.grid[Int][r] == (1..N) } and
  all(c, r: 0...m) { s.grid[rng[c]][rng[r]] == (1..N) }
}
```

Symbolic by Concrete Execution

goal

- translate αRby programs to (symbolic) alloy models

approach

- run αRby programs using the standard ruby interpreter
- the return value is the symbolic result

```
pred solved[s: Sudoku] {
  # concrete
  m = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N)    { s.grid[r][Int] == (1..N) && s.grid[Int][r] == (1..N) } and
  all(c, r: 0...m) { s.grid[rng[c]][rng[r]] == (1..N) }
}
```

implicit in Alloy;
must be explicit in αRby

Symbolic by Concrete Execution

goal

- translate αRby programs to (symbolic) alloy models

approach

- run αRby programs using the standard ruby interpreter
- the return value is the symbolic result

```
pred solved[s: Sudoku] {
  # concrete
  m = Integer(Math.sqrt(N))
  rng = lambda{|i| m*i...m*(i+1)}

  # symbolic
  all(r: 0...N)    { s.grid[r][Int] == (1..N) && s.grid[Int][r] == (1..N) } and
  all(c, r: 0...m) { s.grid[rng[c]][rng[r]] == (1..N) }
}
```

implicit in Alloy;
must be explicit in αRby

benefits

- overload methods in sym classes instead of writing interpreter
- concrete values automatically evaluated

Online Source Instrumentation

challenge

- not all ruby operators can be overridden
  - → all logic operators: `&&`, `||`, `and`, `or`, ...
  - → all branching constructs: `if-then-else` (and all its variants)

Online Source Instrumentation

challenge

- not all ruby operators can be overridden
  - → all logic operators: `&&`, `||`, `and`, `or`, ...
  - → all branching constructs: `if-then-else` (and all its variants)

solution

- use an off-the-shelf ruby parser
- run a simple AST search-replace algorithm
- replace
  - → `x if c` with `BinExpr.new(IMPLIES, proc{c}, proc{x})`
  - → `a and b` with `BinExpr.new(AND, proc{a}, proc{b})`, etc.

Online Source Instrumentation

challenge

- not all ruby operators can be overridden
  - all logic operators: `&&`, `||`, `and`, `or`, ...
  - all branching constructs: `if-then-else` (and all its variants)

solution

- use an off-the-shelf ruby parser
- run a simple AST search-replace algorithm
- replace
  - `x if c` with `BinExpr.new(IMPLIES, proc{c}, proc{x})`
  - `a and b` with `BinExpr.new(AND, proc{a}, proc{b})`, etc.
- optional instrumentation (nicer syntax for some idioms)
  - `s.*f` ⟶ `s.join(f.closure)`
  - ...

Scripting for Alloy

Scripting for Alloy

mixed execution

- dynamically generate Alloy models (specifications)
- allows for parameterized and more flexible specifications
- *example*: generate Sudoku specification for a given size

Scripting for Alloy

mixed execution

- dynamically generate Alloy models (specifications)
- allows for parameterized and more flexible specifications
- *example*: generate Sudoku specification for a given size

partial instances

- shrink bounds to enforce partial solution (known upfront)
- *example*: pre-filled Sudoku cells

Scripting for Alloy

mixed execution

- dynamically generate Alloy models (specifications)
- allows for parameterized and more flexible specifications
- *example*: generate Sudoku specification for a given size

partial instances

- shrink bounds to enforce partial solution (known upfront)
- *example*: pre-filled Sudoku cells

staged model finding

- iteratively run Alloy (e.g., until some fixpoint)
  - → at each step use previous solutions as a guide
- *example*: generate a minimal Sudoku puzzle

Conclusions

the αRby approach

- addresses a collection of practical problems
- demonstrates an alternative to building classical APIs

Conclusions

the αRby approach

- addresses a collection of practical problems
- demonstrates an alternative to building classical APIs

but more broadly

- a new way to think about a modeling language
- microkernel modeling/specification language idea
  - → design a clean set of core modeling features
  - → build all idioms as functions in the outer shell

Conclusions

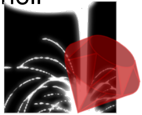
the αRby approach

- addresses a collection of practical problems
- demonstrates an alternative to building classical APIs

but more broadly

- a new way to think about a modeling language
- microkernel modeling/specification language idea
  - → design a clean set of core modeling features
  - → build all idioms as functions in the outer shell

Thank You!



αRby: http://people.csail.mit.edu/aleks/arby

Sudoku: More Reasons for Mixed Execution

Sudoku: More Reasons for Mixed Execution

- one of the top results for the "alloy sudoku" internet search [1]

[1] `https://gist.github.com/athos/1817230`

```
// Numbers
abstract sig Digit {}
one sig One, Two, Three, Four extends Digit {}

// cells
sig Cell { content: one One+Two+Three+Four }
one sig Ca0, Ca1, Ca2, Ca3,
        Cb0, Cb1, Cb2, Cb3,
        Cc0, Cc1, Cc2, Cc3,
        Cd0, Cd1, Cd2, Cd3 extends Cell {}

// groups
sig Group { cells: set Cell } {
  no disj c,c': cells | c.content=c'.content
}
sig Row, Column, Matrix extends Group {}
one sig Ra, Rb, Rc, Rd  extends Row {}
one sig C0, C1, C2, C3  extends Column {}
one sig M0, M1, M2, M3  extends Matrix {}
```

```
// assign cells to groups
fact {
  Ra.cells = Ca0+Ca1+Ca2+Ca3
  Rb.cells = Cb0+Cb1+Cb2+Cb3
  Rc.cells = Cc0+Cc1+Cc2+Cc3
  Rd.cells = Cd0+Cd1+Cd2+Cd3

  C0.cells = Ca0+Cb0+Cc0+Cd0
  C1.cells = Ca1+Cb1+Cc1+Cd1
  C2.cells = Ca2+Cb2+Cc2+Cd2
  C3.cells = Ca3+Cb3+Cc3+Cd3

  M0.cells = Ca0+Ca1+Cb0+Cb1
  M1.cells = Ca2+Ca3+Cb2+Cb3
  M2.cells = Cc0+Cc1+Cd0+Cd1
  M3.cells = Cc2+Cc3+Cd2+Cd3
}

run {} for 20 but 16 Cell
```

Sudoku: More Reasons for Mixed Execution

- one of the top results for the “alloy sudoku” internet search [1]

[1] `https://gist.github.com/athos/1817230`

```
// Numbers
abstract sig Digit {}
one sig One, Two, Three, Four extends Digit {}

// cells
sig Cell { content: one One+Two+Three+Four }
one sig Ca0, Ca1, Ca2, Ca3,
        Cb0, Cb1, Cb2, Cb3,
        Cc0, Cc1, Cc2, Cc3,
        Cd0, Cd1, Cd2, Cd3 extends Cell {}

// groups
sig Group { cells: set Cell } {
  no disj c,c’: cells | c.content=c’.content
}
sig Row, Column, Matrix extends Group {}
one sig Ra, Rb, Rc, Rd  extends Row {}
one sig C0, C1, C2, C3  extends Column {}
one sig M0, M1, M2, M3  extends Matrix {}
```

```
// assign cells to groups
fact {
  Ra.cells = Ca0+Ca1+Ca2+Ca3
  Rb.cells = Cb0+Cb1+Cb2+Cb3
  Rc.cells = Cc0+Cc1+Cc2+Cc3
  Rd.cells = Cd0+Cd1+Cd2+Cd3

  C0.cells = Ca0+Cb0+Cc0+Cd0
  C1.cells = Ca1+Cb1+Cc1+Cd1
  C2.cells = Ca2+Cb2+Cc2+Cd2
  C3.cells = Ca3+Cb3+Cc3+Cd3

  M0.cells = Ca0+Ca1+Cb0+Cb1
  M1.cells = Ca2+Ca3+Cb2+Cb3
  M2.cells = Cc0+Cc1+Cd0+Cd1
  M3.cells = Cc2+Cc3+Cd2+Cd3
}

run {} for 20 but 16 Cell
```

good

- elegant solution (very simple constraints)
- doesn’t use integer arithmetic
- possibly more efficient than with integers

→ the structure can be encoded as a partial instance

Sudoku: More Reasons for Mixed Execution

- one of the top results for the "alloy sudoku" internet search [1]

[1] https://gist.github.com/athos/1817230

```
// Numbers                                       // assign cells to groups
abstract sig Digit {}                            fact {
one sig One, Two, Three, Four extends Digit {}     Ra.cells = Ca0+Ca1+Ca2+Ca3
                                                   Rb.cells = Cb0+Cb1+Cb2+Cb3
// cells                                           Rc.cells = Cc0+Cc1+Cc2+Cc3
sig Cell { content: one One+Two+Three+Four }       Rd.cells = Cd0+Cd1+Cd2+Cd3
one sig Ca0, Ca1, Ca2, Ca3,
        Cb0, Cb1, Cb2, Cb3,                        C0.cells = Ca0+Cb0+Cc0+Cd0
        Cc0, Cc1, Cc2, Cc3,                        C1.cells = Ca1+Cb1+Cc1+Cd1
        Cd0, Cd1, Cd2, Cd3 extends Cell {}         C2.cells = Ca2+Cb2+Cc2+Cd2
                                                   C3.cells = Ca3+Cb3+Cc3+Cd3
// groups
sig Group { cells: set Cell }{                     M0.cells = Ca0+Ca1+Cb0+Cb1
  no disj c,c': cells | c.content=c'.content       M1.cells = Ca2+Ca3+Cb2+Cb3
}                                                  M2.cells = Cc0+Cc1+Cd0+Cd1
sig Row, Column, Matrix extends Group {}           M3.cells = Cc2+Cc3+Cd2+Cd3
one sig Ra, Rb, Rc, Rd  extends Row {}           }
one sig C0, C1, C2, C3  extends Column {}
one sig M0, M1, M2, M3  extends Matrix {}        run {} for 20 but 16 Cell
```

good

- elegant solution (very simple constraints)
- doesn't use integer arithmetic
- possibly more efficient than with integers

→ the structure can be encoded as a partial instance

bad

- hardcoded for size 4
- too much "copy-paste" repetition
- tedious to write for larger sizes
- partial instance encoded as a constraint

Sudoku: Mixed Execution in α**Rby**

```ruby
# Returns an Alloy model formally specifying the Sudoku puzzle for a given size.
# @param n: sudoku size
def self.gen_sudoku_spec(n)
  m = Math.sqrt(n).to_i # precompute sqrt(n) (used below to build the spec)

  # use the aRby DSL to specify Alloy model
  alloy :Sudoku do
     # ...
  end
end
```

Sudoku: Mixed Execution in α**Rby**

```ruby
# Returns an Alloy model formally specifying the Sudoku puzzle for a given size.
# @param n: sudoku size
def self.gen_sudoku_spec(n)
  m = Math.sqrt(n).to_i # precompute sqrt(n) (used below to build the spec)

  # use the aRby DSL to specify Alloy model
  alloy :Sudoku do
    self::N = n # save ‘n‘ as a constant in this Ruby module

    # declare base sigs (independent of sudoku size)
    abstract sig Digit
    abstract sig Cell  [ content: (one Digit) ]
    abstract sig Group [ cells: (set Cell) ] {
      no(c1, c2: cells) { c1 != c2 and c1.content == c2.content }
    }
    # ...
  end
end
```

Sudoku: Mixed Execution in αRby

```
# Returns an Alloy model formally specifying the Sudoku puzzle for a given size.
# @param n: sudoku size
def self.gen_sudoku_spec(n)
  m = Math.sqrt(n).to_i # precompute sqrt(n) (used below to build the spec)

  # use the aRby DSL to specify Alloy model
  alloy :Sudoku do
    self::N = n # save ‘n‘ as a constant in this Ruby module

    # declare base sigs (independent of sudoku size)
    abstract sig Digit
    abstract sig Cell  [ content: (one Digit) ]
    abstract sig Group [ cells: (set Cell) ] {
      no(c1, c2: cells) { c1 != c2 and c1.content == c2.content }
    }
    # generate concrete sigs for the given size
    (0...n).each do |i|
      one sig "D#{i}" < Digit
      one sig "R#{i}", "C#{i}", "M#{i}" < Group
      (0...n).each{ |j| one sig "C#{i}#{j}" < Cell }
    end
  end
end
```

Reconciling Alloy and Ruby: Structures

```
alloy :Grandpa do
  abstract sig Person [
    father: (lone Man),
    mother: (lone Woman)
  ]
  sig Man extends Person [
    wife: (lone Woman)
  ]
  sig Woman extends Person [
    husband: (lone Man)
  ]
  fact terminology_biology {
    wife == ~husband and
    no(p: Person) {
      p.in? p.^(mother + father)
    }
  }
end
```

⇝

```
module Grandpa
  class Person < Arby::Ast::Sig
    attr_accessor :father
    attr_accessor :mother
  end
  class Man < Person
    attr_accessor :wife
  end
  class Woman < Person
    attr_accessor :husband
  end
  def fact_terminology_biology
    wife = ~husband and
    no(p: Person) {
      p.in? p.^(father + mother)
    }
  end
end
```

Reconciling Alloy and Ruby: Structures

```
alloy :Grandpa do
  abstract sig Person [
    father: (lone Man),
    mother: (lone Woman)
  ]
  sig Man extends Person [
    wife: (lone Woman)
  ]
  sig Woman extends Person [
    husband: (lone Man)
  ]
  fact terminology_biology {
    wife == ~husband and
    no(p: Person) {
      p.in? p.^(mother + father)
    }
  }
end
```

⤳

```
module Grandpa
  class Person < Arby::Ast::Sig
    attr_accessor :father
    attr_accessor :mother
  end
  class Man < Person
    attr_accessor :wife
  end
  class Woman < Person
    attr_accessor :husband
  end
  def fact_terminology_biology
    wife = ~husband and
    no(p: Person) {
      p.in? p.^(father + mother)
    }
  end
end
```

- generated on the fly, automatically and transparently
- type (and other) info saved in `Grandpa.meta`

document