Unifying Execution of Imperative and Declarative Code




Aleksandar Milicevic Derek Rayside Kuat Yessenov Daniel Jackson

Massachusetts Institute of Technology
Cambridge, MA

33rd International Conference on Software Engineering
May 27, 2011

Solving Sudoku

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Sudoku puzzle: fill in the empty cells s.t.:

1. all rows contain all values from 1 to 9
2. all columns contain all values from 1 to 9
3. all sub-grids contain all values from 1 to 9

Solving Sudoku

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Sudoku puzzle: fill in the empty cells s.t.:

1. all rows contain all values from 1 to 9
2. all columns contain all values from 1 to 9
3. all sub-grids contain all values from 1 to 9

Approaches:

- write a custom (heuristic-based) algorithm [imperative]
- write a set of constraints and use a constraint solver [declarative]

Solving Sudoku

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Sudoku puzzle: fill in the empty cells s.t.:

1. all rows contain all values from 1 to 9
2. all columns contain all values from 1 to 9
3. all sub-grids contain all values from 1 to 9

Approaches:

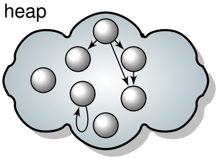
- write a custom (heuristic-based) algorithm [imperative]
- write a set of constraints and use a constraint solver [declarative]

Sudoku with **Squander**

```
public class Sudoku {
  private int[][] grid = new int[9][9];



  public void solve() { ??? }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

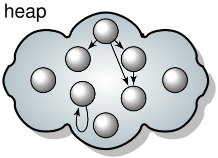
Sudoku with **Squander**




```java
public class Sudoku {
  private int[][] grid = new int[9][9];



  public void solve() { ??? }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Sudoku with **Squander**

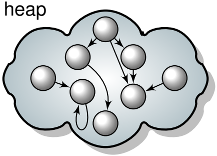



```java
public class Sudoku {
  private int[][] grid = new int[9][9];



  public void solve() { ??? }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Sudoku with **Squander**

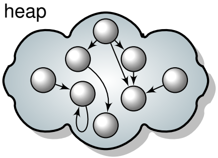
heap



```java
public class Sudoku {
  private int[][] grid = new int[9][9];



  public void solve() { ??? }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Sudoku with **Squander**

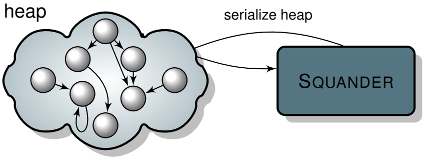


1. all rows contain all values from 1 to 9
2. all columns contain all values from 1 to 9
3. all sub-grids contain all values from 1 to 9

```
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all col in {0 ... 8} | this.grid[int][col] = {1 ... 9}",
    "all r, c in {0, 1, 2} | this.grid[{r*3 ... r*3+2}][{c*3 ... c*3+2}] = {1 ... 9}"})
  @Modifies("this.grid[int].elems | _<2> = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

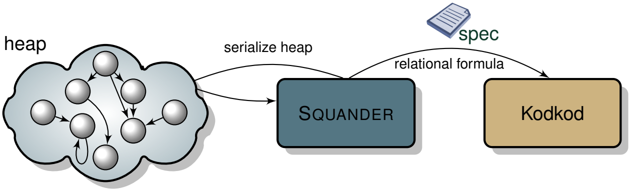
Sudoku with **Squander**




```
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all col in {0 ... 8} | this.grid[int][col] = {1 ... 9}",
    "all r, c in {0, 1, 2} | this.grid[{r*3 ... r*3+2}][{c*3 ... c*3+2}] = {1 ... 9}"})
  @Modifies("this.grid[int].elems | _<2> = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

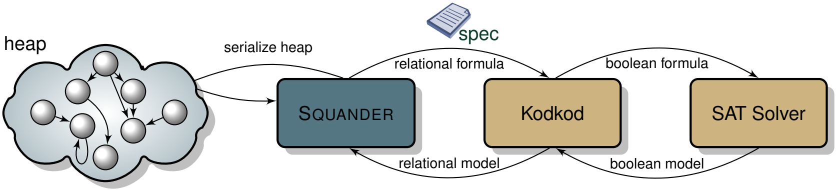
Sudoku with **Squander**




```
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all col in {0 ... 8} | this.grid[int][col] = {1 ... 9}",
    "all r, c in {0, 1, 2} | this.grid[{r*3 ... r*3+2}][{c*3 ... c*3+2}] = {1 ... 9}"})
  @Modifies("this.grid[int].elems | _<2> = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

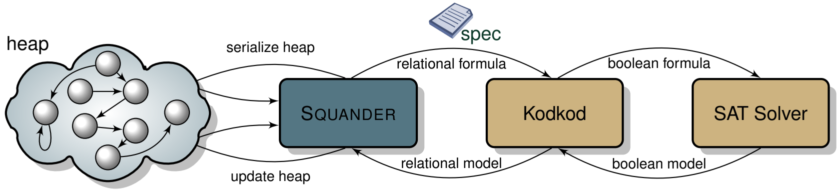
Sudoku with **Squander**




```java
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all col in {0 ... 8} | this.grid[int][col] = {1 ... 9}",
    "all r, c in {0, 1, 2} | this.grid[{r*3 ... r*3+2}][{c*3 ... c*3+2}] = {1 ... 9}"})
  @Modifies("this.grid[int].elems | _<2> = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

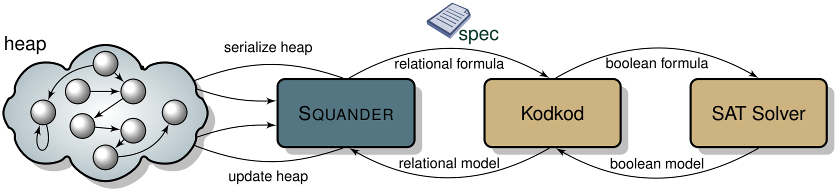
Sudoku with **Squander**




```
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all col in {0 ... 8} | this.grid[int][col] = {1 ... 9}",
    "all r, c in {0, 1, 2} | this.grid[{r*3 ... r*3+2}][{c*3 ... c*3+2}] = {1 ... 9}"})
  @Modifies("this.grid[int].elems | _<2> = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

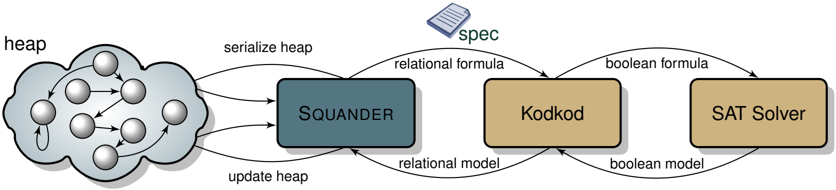
Sudoku with **Squander**




```
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all col in {0 ... 8} | this.grid[int][col] = {1 ... 9}",
    "all r, c in {0, 1, 2} | this.grid[{r*3 ... r*3+2}][{c*3 ... c*3+2}] = {1 ... 9}"})
  @Modifies("this.grid[int].elems | _<2> = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

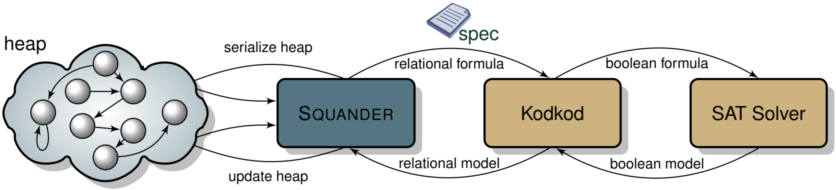
Sudoku with **Squander**




```
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all c
    "all r,                                                   c*3+2}] = {1 ... 9}"})
  @Modifies
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

executable first-order relational specifications for Java

Sudoku with **Squander**

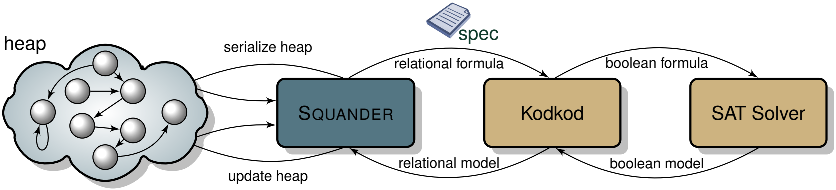



```
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all col in {0 ... 8} | this.grid[int][col] = {1 ... 9}",
    "all r, ... c*3+2}] = {1 ... 9}"})
  @Modifies(...)
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

specify and solve constraint problems in place

executable first-order relational specifications for Java

Sudoku with **Squander**




```
public class Sudoku {
  private int[][] grid = new int[9][9];

  @Ensures({
    "all row in {0 ... 8} | this.grid[row][int] = {1 ... 9}",
    "all col in {0 ... 8} | this.grid[int][col] = {1 ... 9}",
    "all r ... c*3+2}] = {1 ... 9}"})
  @Modifies("this.grid[int].elems | <25 = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.grid[0][3] = 1; ...; s.grid[8][5] = 1;
    s.solve();
    System.out.println(s);
  }
}
```

specify and solve constraint problems in place

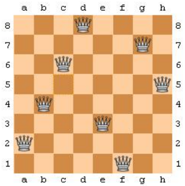
executable first-order relational specifications for Java

no manual translation to/from an external solver

SQUANDER vs Manual Search

N-Queens

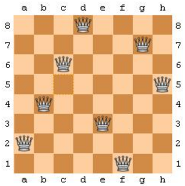
- *place N queens on an N×N chess board such that no two queens attack each other*



SQUANDER vs Manual Search

N-Queens

- *place N queens on an N×N chess board such that no two queens attack each other*



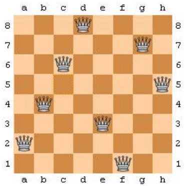
A backtracking with pruning solution

```
static boolean solveNQueens(int n, int col, int[] queenCols,
                            boolean[] bRow, boolean[] bD45, boolean[] bD135) {
  if (col >= n)
    return true;

  for (int row = 0; row < n; row++) {
    if (bRow[row] || bD45[row + col] || bD135[col - row + n - 1])
      continue;
    queenCols[col] = row;
    bRow[row]              = true;
    bD45[row + col]        = true;
    bD135[col - row + n - 1] = true;
    if (solveNQueens(n, col+1, queenCols, bRow, bD45, bD135))
      return true;
    bRow[row]              = false;
    bD45[row + col]        = false;
    bD135[col - row + n - 1] = false;
  }

  return false;
}
```

SQUANDER vs Manual Search

N-Queens

- *place N queens on an N×N chess board such that no two queens attack each other*



A backtracking with pruning solution

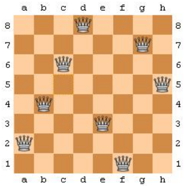
```
static boolean solveNQueens(int n, int col, int[] queenCols,
                            boolean[] bRow, boolean[] bD45, boolean[] bD135) {
  if (col >= n)
    return true;

  for (int row = 0; row < n; row++) {
    if (bRow[row] || bD45[row + col] || bD135[col - row + n - 1])
        continue;
    queenCols[col] = row;
    bRow[row]               = true;
    bD45[row + col]         = true;
    bD135[col - row + n - 1] = true;
    if (solveNQueens(n, col+1, queenCols, bRow, bD45, bD135))
      return true;
    bRow[row]               = false;
    bD45[row + col]         = false;
    bD135[col - row + n - 1] = false;
  }

  return false;
}
```

doesn't look terribly bad, but fairly complicated

SQUANDER vs Manual Search

N-Queens

- *place N queens on an $N \times N$ chess board such that no two queens attack each other*



A backtracking with pruning solution

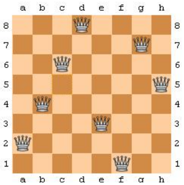
```
static boolean solveNQueens(int n, int col, int[] queenCols,
                            boolean[] bRow, boolean[] bD45, boolean[] bD135) {
  if (col >= n)
    return true;

  for (int row = 0; row < n; row++) {
    if (bRow[row] || bD45[row + col] || bD135[col - row + n - 1])
        continue;
    queenCols[col] = row;
    bRow[row]              = true;
    bD45[row + col]        = true;
    bD135[col - row + n - 1] = true;
    if (solveNQueens(n, col+1, queenCols, bRow, bD45, bD135))
      return true;
    bRow[row]              = false;
    bD45[row + col]        = false;
    bD135[col - row + n - 1] = false;
  }

  return false;
}
```

doesn't look terribly bad, but fairly complicated

how do you argue that it is correct?

N-Queens

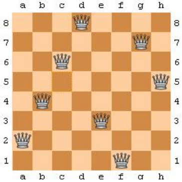
- *place N queens on an N×N chess board such that no two queens attack each other*



A solution with SQUANDER

```
@Ensures({
   "all disj q, r: result.elts |  " +   // for every two different queens q and r ensure that they are
   "   q.i          != r.i        && " +   //    not in the same row
   "   q.j          != r.j        && " +   //    not in the same column
   "   q.i − q.j != r.i − r.j  && " +   //    not in the same ↖↘ diagonal
   "   q.i + q.j != r.i + r.j" })          //    not in the same ↙↗ diagonal
@Modifies({
   "result.elts.i from {0 ... n−1}",      // modify fields i and j of all elements of
   "result.elts.j from {0 ... n−1}" })   // the result set, but only assign values from {0, ..., n−1}
static void solveNQueens(int n, Set<Queen> result) {
    Squander.exe(null, n, result);
}
```

SQUANDER vs Manual Search



N-Queens

- *place N queens on an N×N chess board such that no two queens attack each other*

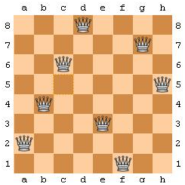
A solution with SQUANDER

```
@Ensures({
  "all disj q, r: result.elts |   " +   // for every two different queens q and r ensure that they are
  "   q.i        != r.i        && " +   //   not in the same row
  "   q.j        != r.j        && " +   //   not in the same column
  "   q.i − q.j  != r.i − r.j  && " +   //   not in the same ↘ diagonal
  "   q.i + q.j  != r.i + r.j" })       //   not in the same ↗ diagonal
@Modifies({
  "result.elts.i from {0 ... n−1}",     // modify fields i and j of all elements of
  "result.elts.j from {0 ... n−1}" })   // the result set, but only assign values from {0, ..., n−1}
static void solveNQueens(int n, Set<Queen> result) {
    Squander.exe(null, n, result);
}
```

says what, not how

N-Queens

- *place N queens on an N×N chess board such that no two queens attack each other*



A solution with SQUANDER

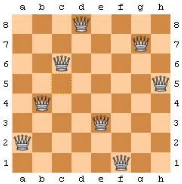
```
@Ensures({
  "all disj q, r: result.elts |  " +   // for every two different queens q and r ensure that they are
  "   q.i        != r.i        && " +   //    not in the same row
  "   q.j        != r.j        && " +   //    not in the same column
  "   q.i - q.j != r.i - r.j   && " +   //    not in the same ↖ diagonal
  "   q.i + q.j != r.i + r.j" })        //    not in the same ↗ diagonal
@Modifies({
  "result.elts.i from {0 ... n-1}",     // modify fields i and j of all elements of
  "result.elts.j from {0 ... n-1}" })  // the result set, but only assign values from {0, ..., n-1}
static void solveNQueens(int n, Set<Queen> result) {
    Squander.exe(null, n, result);
}
```

says what, not how

(almost) correct by construction!

SQUANDER vs Manual Search

N-Queens



- *place N queens on an N×N chess board such that no two queens attack each other*

A solution with SQUANDER

```
@Ensures({
   "all disj q, r: result.elts | " +   // for every two different queens q and r ensure that they are
   "   q.i       != r.i        && " +   //    not in the same row
   "   q.j       != r.j        && " +   //    not in the same column
   "   q.i - q.j != r.i - r.j  && " +   //    not in the same ↖↘ diagonal
   "   q.i + q.j != r.i + r.j" })       //    not in the same ↙↗ diagonal
@Modifies({
   "result.elts.i from {0 ... n-1}",    // modify fields i and j of all elements of
   "result.elts.j from {0 ... n-1}" })  // the result set, but only assign values from {0, ..., n-1}
static void solveNQueens(int n, Set<Queen> result) {
    Squander.exe(null, n, result);
}
```

says what, not how

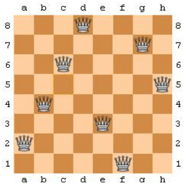
(almost) correct by construction!

What about performance?

SQUANDER vs Manual Search

N-Queens

- *place N queens on an N×N chess board such that no two queens attack each other*



A solution with SQUANDER

```
@Ensures({
   "all disj q, r: result.elts |  " +   // for every two different queens q and r ensure that they are
   "   q.i        != r.i        && " +   //    not in the same row
   "   q.j        != r.j        && " +   //    not in the same column
   "   q.i - q.j  != r.i - r.j  && " +   //    not in the same ↖ diagonal
   "   q.i + q.j  != r.i + r.j" })      //    not in the same ↗ diagonal
@Modifies({
   "result.elts.i from {0 ... n-1}",     // modify fields i and j of all elements of
   "result.elts.j from {0 ... n-1}" })   // the result set, but only assign values from {0, ..., n-1}
static void solveNQueens(int n, Set<Queen> result) {
    Squander.exe(null, n, result);
}
```

says what, not how

(almost) correct by construction!

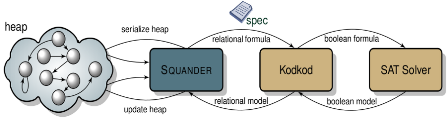
What about performance?

- It even outperforms the backtracking algorithm in this case!

Outline

Framework Overview

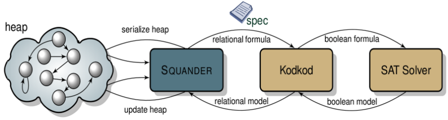
- specification language
- SQUANDER architecture



Outline

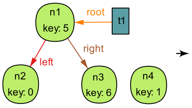
Framework Overview

- specification language
- SQUANDER architecture



Translation

- from Java heap + specs to Kodkod
- minimizing the universe size



BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

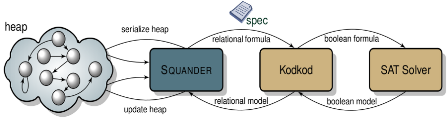
key_pre: $\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre: $\{(t_1 \to n_1)\}$
left_pre: $\{(n_1 \to n_2), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
right_pre: $\{(n_1 \to n_3), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$

root: $\{\}$, $\{t_1\} \times \{n_1, n_2, n_3, n_4\}$
left: $\{\}$, $\{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$
right: $\{\}$, $\{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$

Outline

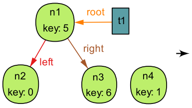
Framework Overview

- specification language
- SQUANDER architecture



Translation

- from Java heap + specs to Kodkod
- minimizing the universe size



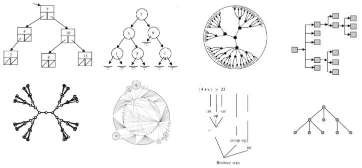
BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

key_pre: $\{(n_1 \rightarrow 5), (n_2 \rightarrow 0), (n_3 \rightarrow 6), (n_4 \rightarrow 1)\}$
root_pre: $\{(t_1 \rightarrow n_1)\}$
left_pre: $\{(n_1 \rightarrow n_2), (n_2 \rightarrow null), (n_3 \rightarrow null), (n_4 \rightarrow null)\}$
right_pre: $\{(n_1 \rightarrow n_3), (n_2 \rightarrow null), (n_3 \rightarrow null), (n_4 \rightarrow null)\}$

root: $\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$
left: $\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$
right: $\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$

Treatment of Data Abstractions

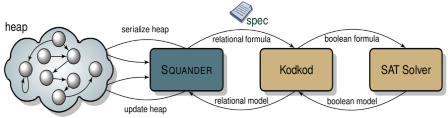
- support for third party library classes (e.g. Java collections)



Outline

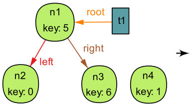
Framework Overview

- specification language
- SQUANDER architecture



Translation

- from Java heap + specs to Kodkod
- minimizing the universe size

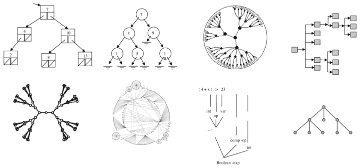


BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

key_pre:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre:	$\{(t_1 \to n_1)\}$
left_pre:	$\{(n_1 \to n_2), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
right_pre:	$\{(n_1 \to n_3), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
root:	$\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$
left:	$\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$
right:	$\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$

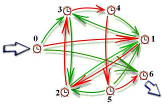
Treatment of Data Abstractions

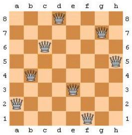
- support for third party library classes (e.g. Java collections)



Evaluation/Case Study

- performance advantages for some puzzles and graph algorithms
- case study: *MIT course scheduler*

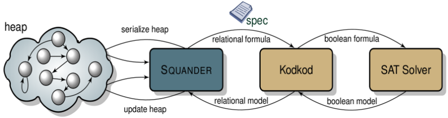




Framework Overview

Framework Overview

- specification language
- SQUANDER architecture



Translation

- from Java heap + specs to Kodkod
- minimizing the universe size

Treatment of Data Abstractions

- support for third party library classes (e.g. Java collections)

Evaluation/Case Study

- performance advantages for some puzzles and graph algorithms
- case study: *MIT course scheduler*

Specification Language

Example - Binary Search Tree

```java
public class Tree {
  private Node root;
}
```

```java
public class Node {
  private Node left, right;
  private int key;
}
```

Specification Language

Example - Binary Search Tree

```
public class Tree {
  private Node root;
}
```

```
public class Node {
  private Node left, right;
  private int key;
}
```

Annotations

class specification field **@SpecField** ("<fld_decl> | <abs_func>")

Specification Language

Example - Binary Search Tree

```
public class Tree {
  private Node root;
}
```

```
public class Node {
  private Node left, right;
  private int key;
}
```

Annotations

class specification field **@SpecField** ("<fld_decl> | <abs_func>")

```
@SpecField("this.nodes: set Node | this.nodes = this.root.*(left+right) − null")
public class Tree {
```

Specification Language

Example - Binary Search Tree

```
public class Tree {
  private Node root;
}
```

```
public class Node {
  private Node left, right;
  private int key;
}
```

Annotations

class specification field **@SpecField** ("<fld_decl> | <abs_func>")

```
@SpecField("this.nodes: set Node | this.nodes = this.root.*(left+right) − null")
public class Tree {
```

class invariant **@Invariant** ("<expr>")

Specification Language

Example - Binary Search Tree

```
public class Tree {
  private Node root;
}
```

```
public class Node {
  private Node left, right;
  private int key;
}
```

Annotations

class specification field **@SpecField** ("<fld_decl> | <abs_func>")

```
@SpecField("this.nodes: set Node | this.nodes = this.root.*(left+right) − null")
public class Tree {
```

class invariant **@Invariant** ("<expr>")

```
@Invariant({
  /* left sorted  */ "all x: this.left.*(left+right) − null | x.key < this.key",
  /* right sorted */ "all x: this.right.*(left+right) − null | x.key > this.key"})
public class Node {
```

Specification Language

Example - Binary Search Tree

```
public class Tree {
  private Node root;
}
```

```
public class Node {
  private Node left, right;
  private int key;
}
```

Annotations

class specification field **@SpecField** ("<fld_decl> | <abs_func>")

```
@SpecField("this.nodes: set Node | this.nodes = this.root.*(left+right) − null")
public class Tree {
```

class invariant **@Invariant** ("<expr>")

```
@Invariant({
  /* left sorted  */ "all x: this.left.*(left+right) − null | x.key < this.key",
  /* right sorted */ "all x: this.right.*(left+right) − null | x.key > this.key"})
public class Node {
```

method pre-condition **@Requires** ("<expr>")
method post-condition **@Ensures** ("<expr>")
method frame condition **@Modifies** ("<fld> | < filter > **from** <domain>")

Specification Language

Example - Binary Search Tree

```
public class Tree {
  private Node root;
}
```

```
public class Node {
  private Node left, right;
  private int key;
}
```

Annotations

class specification field **@SpecField** ("<fld_decl> | <abs_func>")

```
@SpecField("this.nodes: set Node | this.nodes = this.root.*(left+right) − null")
public class Tree {
```

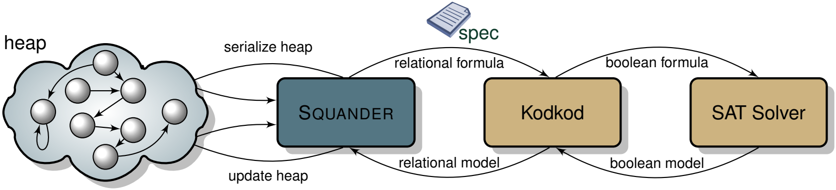
class invariant **@Invariant** ("<expr>")

```
@Invariant({
  /* left sorted */ "all x: this.left.*(left+right) − null | x.key < this.key",
  /* right sorted */ "all x: this.right.*(left+right) − null | x.key > this.key"})
public class Node {
```

method pre-condition **@Requires** ("<expr>")
method post-condition **@Ensures** ("<expr>")
method frame condition **@Modifies** ("<fld> | < filter > **from** <domain>")

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```

Framework Overview



Execution steps

- traverse the heap and assemble the relevant constraints
- translate to Kodkod
  - translate the heap to relations and bounds
  - collect all the specs and assemble a single relational formula
- if a solution is found, update the heap to reflect the solution

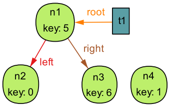
Translation

Framework Overview

- specification language
- SQUANDER architecture

Translation

- from Java heap + specs to Kodkod
- minimizing the universe size



BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

key_pre:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre:	$\{(t_1 \to n_1)\}$
left_pre:	$\{(n_1 \to n_2), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
right_pre:	$\{(n_1 \to n_3), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
root:	$\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$
left:	$\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$
right:	$\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$

Treatment of Data Abstractions

- support for third party library classes (e.g. Java collections)

Evaluation/Case Study

- performance advantages for some puzzles and graph algorithms
- case study: *MIT course scheduler*

From Objects to Relations

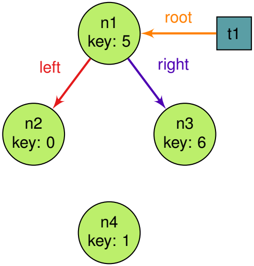
The back-end solver — Kodkod

- constraint solver for first-order logic with relations
- SAT-based finite relational model finder
  - finite bounds must be provided for all relations
- designed to be efficient for partial models
  - partial instances are encoded using bounds

From Objects to Relations

Translation of the `BST.insert` method

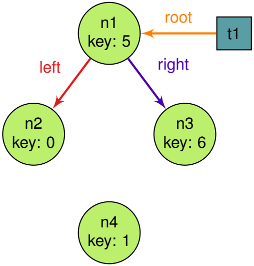
```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```



From Objects to Relations

Translation of the `BST.insert` method

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```



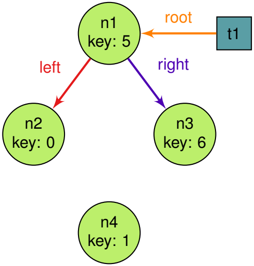
BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

reachable objects

From Objects to Relations

Translation of the `BST.insert` method

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```



BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

reachable objects

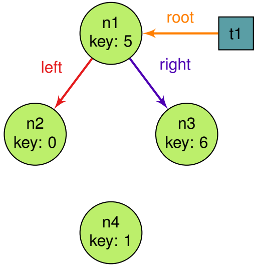
key_pre:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre:	$\{(t_1 \to n_1)\}$
left_pre:	$\{(n_1 \to n_2), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
right_pre:	$\{(n_1 \to n_3), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$

pre-state

From Objects to Relations

Translation of the `BST.insert` method

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```



BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

reachable objects

key_pre:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre:	$\{(t_1 \to n_1)\}$
left_pre:	$\{(n_1 \to n_2), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
right_pre:	$\{(n_1 \to n_3), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$

pre-state

root:	$\{\}$,	$\{t_1\} \times \{n_1, n_2, n_3, n_4, null\}$
left:	$\{n_1 \to n_2\}$,	$\{n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4, null\}$
right:	$\{n_1 \to n_3\}$,	$\{n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4, null\}$

post-state

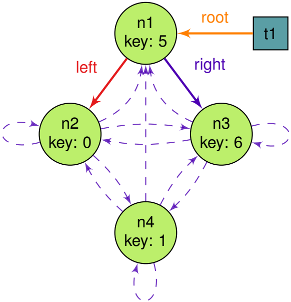
lower bound   upper bound

- **lower bound**: tuples that must be included
- **upper bound**: tuples that may be included
- shrinking the bounds (instead of adding more constraints) leads to more efficient solving

From Objects to Relations

Translation of the `BST.insert` method

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```



reachable objects

BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

pre-state

key_pre:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre:	$\{(t_1 \to n_1)\}$
left_pre:	$\{(n_1 \to n_2), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
right_pre:	$\{(n_1 \to n_3), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$

post-state

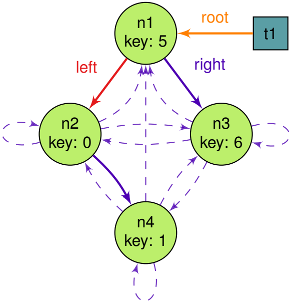
	lower bound	upper bound
root:	$\{\}$,	$\{t_1\} \times \{n_1, n_2, n_3, n_4, null\}$
left:	$\{n_1 \to n_2\}$,	$\{n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4, null\}$
right:	$\{n_1 \to n_3\}$,	$\{n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4, null\}$

- **lower bound**: tuples that must be included
- **upper bound**: tuples that may be included
- shrinking the bounds (instead of adding more constraints) leads to more efficient solving

From Objects to Relations

Translation of the `BST.insert` method

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```



reachable objects

BST1:	$\{t_1\}$	**N3**:	$\{n_3\}$	**BST_this**:	$\{t_1\}$
N1:	$\{n_1\}$	**N4**:	$\{n_4\}$	**z**:	$\{n_4\}$
N2:	$\{n_2\}$	**null**:	$\{null\}$	**ints**:	$\{0, 1, 5, 6\}$

pre-state

key_pre:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre:	$\{(t_1 \to n_1)\}$
left_pre:	$\{(n_1 \to n_2), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$
right_pre:	$\{(n_1 \to n_3), (n_2 \to null), (n_3 \to null), (n_4 \to null)\}$

post-state

	lower bound	upper bound
root:	$\{\}$,	$\{t_1\} \times \{n_1, n_2, n_3, n_4, null\}$
left:	$\{n_1 \to n_2\}$,	$\{n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4, null\}$
right:	$\{n_1 \to n_3\}$,	$\{n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4, null\}$

- **lower bound**: tuples that must be included
- **upper bound**: tuples that may be included
- shrinking the bounds (instead of adding more constraints) leads to more efficient solving

Performance of `Tree.insertNode`

What about performance now?

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```

Performance of `Tree.insertNode`

What about performance now?

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```

- can only handle trees up to about 100 nodes
- reason: tree insertion is algorithmically simple

  → imperative algorithm scales better than NP-complete SAT solving

Performance of `Tree.insertNode`

What about performance now?

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```

- can only handle trees up to about 100 nodes
- reason: tree insertion is algorithmically simple

  $\rightarrow$ imperative algorithm scales better than NP-complete SAT solving

"Squander": *wasting CPU cycles for programmer's cycles*

Performance of `Tree.insertNode`

What about performance now?

```
@Requires("z.key !in this.nodes.key")
@Ensures ("this.nodes = @old(this.nodes) + z")
@Modifies("this.root, this.nodes.left | _<1> = null, this.nodes.right | _<1> = null")
public void insertNode(Node z) { Squander.exe(this, z); }
```

- can only handle trees up to about 100 nodes
- reason: tree insertion is algorithmically simple
  → imperative algorithm scales better than NP-complete SAT solving

"Squander": *wasting CPU cycles for programmer's cycles*

Saving programmer's cycles

- **fast prototyping**: get a correct working solution early on
- **differential testing**: compare the results of imperative and declarative implementations
- **test input generation**: use SQUANDER to generate some binary trees

Generating Binary Search Trees with SQUANDER

```
@Ensures("#this.nodes = size")
@Modifies("this.root, Node.left, Node.right, Node.key")
@FreshObjects(cls=Node.class, num = size),
@Options(solveAll = true)
public void gen(int size) { Squander.exe(this); }
```

- to generate many different trees
  - the caller can use the SQUANDER API to request a different solution for the same specification

Treatment of Data Abstractions

Framework Overview

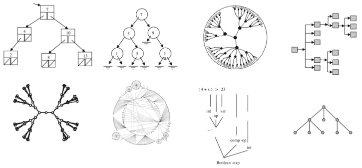
- specification language
- SQUANDER architecture

Translation

- from Java heap + specs to Kodkod
- minimizing the universe size

Treatment of Data Abstractions

- support for third party library classes (e.g. Java collections)



Evaluation/Case Study

- performance advantages for some puzzles and graph algorithms
- case study: *MIT course scheduler*

User-Defined Abstractions for Library Types

Why is it important to be able to specify library types?

- library classes are ubiquitous
- specs need to be able to talk about them

```
class Graph {
  class Node { public int key; }
  class Edge { public Node src, dest; }

  private Set<Node> nodes = new LinkedHashSet<Node>();
  private Set<Edge> edges = new LinkedHashSet<Edge>();

  // how to write a spec for the k-Coloring
  // problem for a graph like this?
  public Map<Node, Integer> color(int k) {
    return Squander.exe(this, k);
  }
}
```

User-Defined Abstractions for Library Types

Why is it important to be able to specify library types?

- library classes are ubiquitous
- specs need to be able to talk about them

```
class Graph {
  class Node { public int key; }
  class Edge { public Node src, dest; }

  private Set<Node> nodes = new LinkedHashSet<Node>();
  private Set<Edge> edges = new LinkedHashSet<Edge>();

  // how to write a spec for the k-Coloring
  // problem for a graph like this?
  public Map<Node, Integer> color(int k) {
    return Squander.exe(this, k);
  }
}
```

- **solution**:
  - use `@SpecField` to specify abstract data types

User-Defined Abstractions for Library Types

How to support a third party class?

- write a spec file

```
interface Map<K,V> {
  @SpecField("elts: K -> V")

  @SpecField("size: one int | this.size = #this.elts")
  @SpecField("keys: set K | this.keys = this.elts.(V)")
  @SpecField("vals: set V | this.vals = this.elts[K]")

  @Invariant({"all k: K | k in this.elts.V => one this.elts[k]"})}
```

User-Defined Abstractions for Library Types

How to support a third party class?

- write a spec file

```
interface Map<K,V> {
  @SpecField("elts: K -> V")

  @SpecField("size: one int | this.size = #this.elts")
  @SpecField("keys: set K | this.keys = this.elts.(V)")
  @SpecField("vals: set V | this.vals = this.elts[K]")

  @Invariant({"all k: K | k in this.elts.V => one this.elts[k]"})}
```

- write an **abstraction** and a **concretization** function

```
public class MapSer implements IObjSer {
  public List<FieldValue> absFunc(JavaScene javaScene, Object obj) {
    // return values for the field "elts": Map -> K -> V
  }

  public Object concrFunc(Object obj, FieldValue fieldValue) {
    // update and return the given object "obj" from
    // the given values of the given abstract field
}}
```

Using Collections: Example

Now we can specify the k-Coloring problem

```
class Graph {
  class Node { public int key; }
  class Edge { public Node src, dest; }

  private Set<Node> nodes = new LinkedHashSet<Node>();
  private Set<Edge> edges = new LinkedHashSet<Edge>();

  @Ensures({
    "return.keys = this.nodes.elts",
    "return.vals in {1 ... k}",
    "all e: this.edges.elts | return.elts[e.src] != return.elts[e.dst]"})
  @Modifies("return.elts")
  @FreshObjects(cls = Map.class, num = 1)
  public Map<Node, Integer> color(int k) {return Squander.exe(this, k);}
}
```

```
interface Set<K> {
  @SpecField("elts: set K")

  @SpecField("size: one int |
    this.size=#this.elts")
}
```

```
interface Map<K,V> {
  @SpecField("elts: K -> V")

  @SpecField("size: one int | this.size = #this.elts")
  @SpecField("keys: set K | this.keys = this.elts.(V)")
  @SpecField("vals: set V | this.vals = this.elts[K]")

  @Invariant({"all k: K | k in this.elts.V => one this.elts[k]"})}
```

Evaluation/Case Study

Framework Overview

- specification language
- SQUANDER architecture

Translation

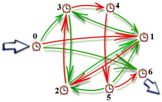
- from Java heap + specs to Kodkod
- minimizing the universe size

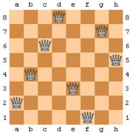
Treatment of Data Abstractions

- support for third party library classes (e.g. Java collections)

Evaluation/Case Study

- performance advantages for some puzzles and graph algorithms
- case study: *MIT course scheduler*

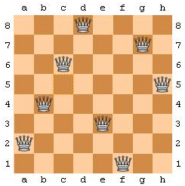


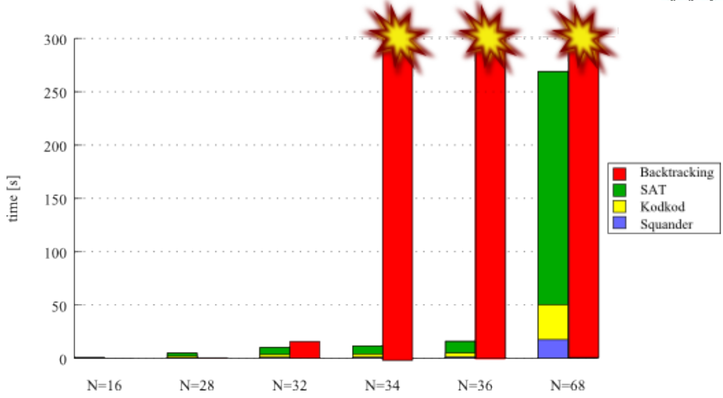


SQUANDER vs Manual Search

N-Queens

- place N queens on an N×N chess board such that no two queens attack each other

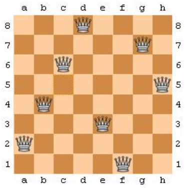


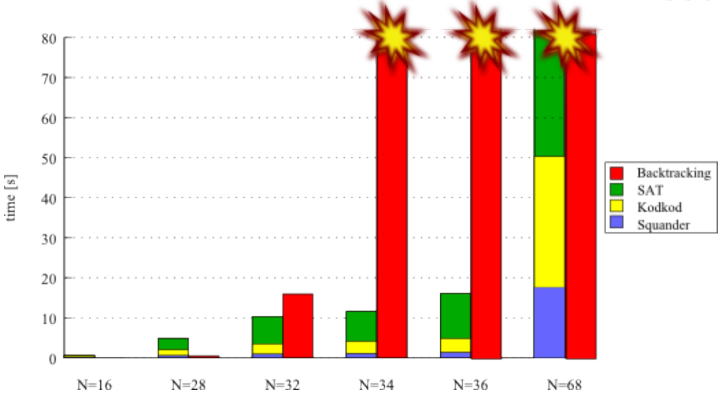


SQUANDER vs Manual Search

N-Queens

- place N queens on an N×N chess board such that no two queens attack each other

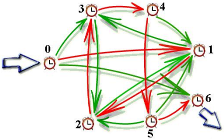




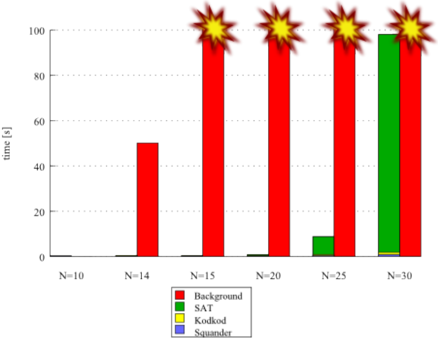
SQUANDER vs Manual Search

Hamiltonian Path

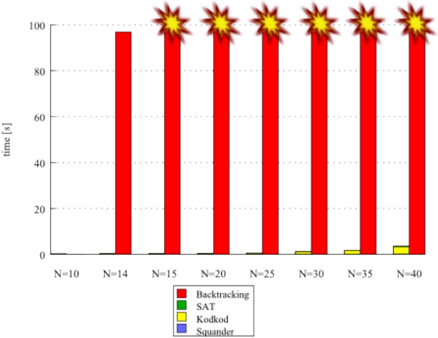
- find a path in a graph that visits all nodes exactly once



Graphs with Hamiltonian path



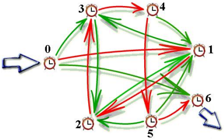
Graphs with no Hamiltonian path



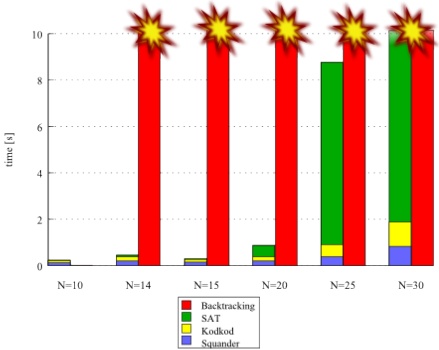
SQUANDER vs Manual Search

Hamiltonian Path

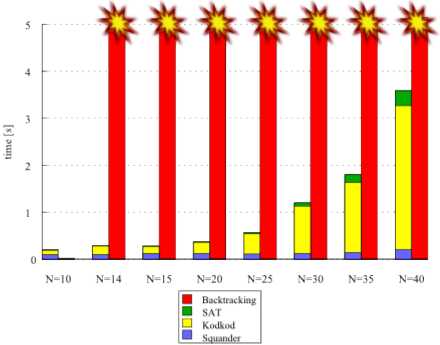
- find a path in a graph that visits all nodes exactly once



Graphs with Hamiltonian path



Graphs with no Hamiltonian path



SQUANDER vs Manual Search

So, is SQUANDER always better than backtracking?

- **of course not!**

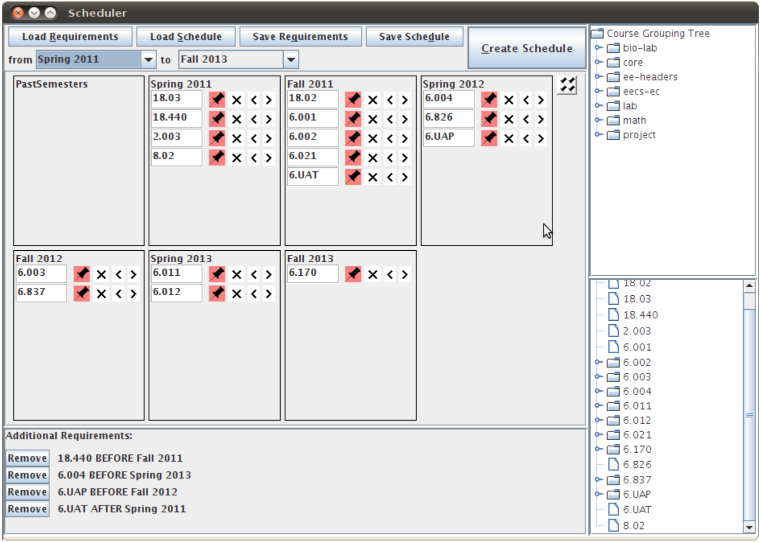
Rather, the takeaway point is

- if the problem is easy to specify, it makes sense to do that first
  1. you'll get a correct solution faster
  2. if the problem is algorithmically complex, the scalability might be satisfying as well

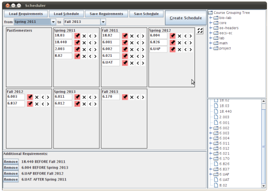
Other Evaluation Questions

- **usability on a real-world constraint problem**
- **annotation overhead**
- **ability to handle large program heaps**
- **efficiency**

Case Study – Course Scheduler

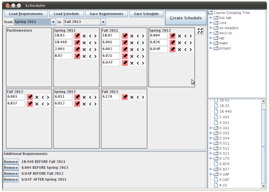


Other Evaluation Questions



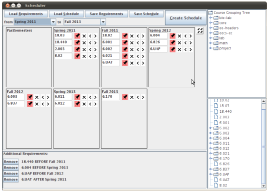
- **usability on a real-world constraint problem**
- **annotation overhead**
- **ability to handle large program heaps**
- **efficiency**

Other Evaluation Questions



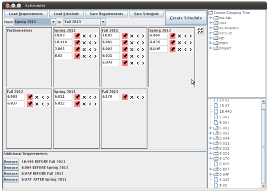
- **usability on a real-world constraint problem**
  - an existing implementation retrofitted with SQUANDER
  - didn't have to change the local structure, just annotate classes
  - ... thanks to the **treatment of data abstractions**
- **annotation overhead**
- **ability to handle large program heaps**
- **efficiency**

Other Evaluation Questions



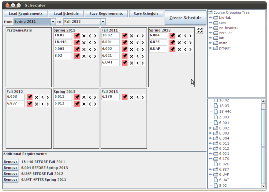
- **usability on a real-world constraint problem**
  - an existing implementation retrofitted with SQUANDER
  - didn't have to change the local structure, just annotate classes
  - ... thanks to the **treatment of data abstractions**
- **annotation overhead**
  - only about 30 lines of specs to replace 1500 lines of code
  - ... thanks to the **unified execution environment**
- **ability to handle large program heaps**
- **efficiency**

Other Evaluation Questions



- **usability on a real-world constraint problem**
  - an existing implementation retrofitted with SQUANDER
  - didn't have to change the local structure, just annotate classes
  - ... thanks to the **treatment of data abstractions**
- **annotation overhead**
  - only about 30 lines of specs to replace 1500 lines of code
  - ... thanks to the **unified execution environment**
- **ability to handle large program heaps**
  - the heap counted almost 2000 objects
  - ... thanks to the **clustering algorithm**
- **efficiency**

Other Evaluation Questions



- **usability on a real-world constraint problem**
  - an existing implementation retrofitted with SQUANDER
  - didn't have to change the local structure, just annotate classes
  - ... thanks to the **treatment of data abstractions**
- **annotation overhead**
  - only about 30 lines of specs to replace 1500 lines of code
  - ... thanks to the **unified execution environment**
- **ability to handle large program heaps**
  - the heap counted almost 2000 objects
  - ... thanks to the **clustering algorithm**
- **efficiency**
  - about 5s as opposed to 1s of the original implementation

Limitations

- **boundedness** – SQUANDER can't generate an arbitrary number of new objects; instead the maximum number of new objects must be explicitly specified by the user
- **integers** – integers must also be bounded to a small bitwidth
- **equality** – only referential equality can be used (except for strings)
- **no higher-order expressions** – e.g. can't specify *find the longest path in the graph*; instead must specify the minimum length k, i.e. *find a path in the graph of length at least k nodes*
- **debugging** – if a solution cannot be found, the user is not given any additional information as to why the specification wasn't satisfiable

Future Work

- **optimize translation** to Kodkod
  - use fewer relations to represent the heap
    (short-circuit some unmodifiable ones)
- **support debugging** better
  - when no solution can be found, explain why
    (with the help of unsat core)
- **synthesize** code from specifications
  - especially for methods that only traverse the heap
- **combine different solvers** in the back end
  - SMT solvers would be better at handling large integers

Summary

SQUANDER lets you

- execute first-order, relational specifications in Java

Summary

SQUANDER lets you

- execute first-order, relational specifications in Java

Why would you want to do that?

- conveniently express and solve algorithmically complicated problems using declarative constraints
- gain performance in certain cases (e.g. for NP-hard problems)
- during development:
  - fast prototyping (get a correct working solution fast)
  - generate test inputs
  - runtime assertion checking

CSAIL

Summary

SQUANDER lets you

- execute first-order, relational specifications in Java

Why would you want to do that?

- conveniently express and solve algorithmically complicated problems using declarative constraints
- gain performance in certain cases (e.g. for NP-hard problems)
- during development:
  - fast prototyping (get a correct working solution fast)
  - generate test inputs
  - runtime assertion checking

Thank You!

http://people.csail.mit.edu/aleks/squander

CSAIL

Solving Sudoku with **Alloy Analyzer**

```
abstract sig Number {}
one sig N1,N2,N3,N4,N5,N6,N7,N8,N9 extends Number {}

one sig Global {
    data: Number -> Number -> one Number
}

pred complete [rows:set Number, cols:set Number]{
   Number = Global.data[rows][cols]
}

pred rules {
    all row: Number { complete[row,Number] }
    all col: Number { complete[Number,col] }
    let r1=N1+N2+N3, r2=N4+N5+N6, r3=N7+N8+N9 |
      complete[r1,r1] and complete[r1,r2] and complete[r1,r3] and
      complete[r2,r1] and complete[r2,r2] and complete[r2,r3] and
      complete[r3,r1] and complete[r3,r2] and complete[r3,r3]
}

pred puzzle {
   N1->N4->N1 + N1->N8->N9 +
   ...
   N9->N2->N2 + N9->N6->N1 in Global.data
}

run { rules and puzzle }
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Solving Sudoku with **Kodkod**

```java
public class Sudoku {
  private Relation Number = Relation.unary("Number");
  private Relation data = Relation.ternary("data");
  private Relation[] regions = new Relation[] {
    Relation.unary("Region1"),
    Relation.unary("Region2"),
    Relation.unary("Region3") };

  public Formula complete(Expression rows, Expression cols) {
    // Number = data[rows][cols]
    return Number.eq(cols.join(rows.join(data))); }

  public Formula rules() {
    // all x,y: Number | lone data[x][y]
    Variable x = Variable.unary("x");
    Variable y = Variable.unary("y");
    Formula f1 = y.join(x.join(data)).lone().
      forAll(x.oneOf(Number).and(y.oneOf(Number)));
    // all row: Number | complete[row, Number]
    Variable row = Variable.unary("row");
    Formula f2 = complete(row, Number).
      forAll(row.oneOf(Number));
    // all col: Number | complete[Number, col]
    Variable col = Variable.unary("col");
    Formula f3 = complete(Number, col).
      forAll(col.oneOf(Number));
    // complete[r1,r1] and complete[r1,r2] and complete[r1,r3] and
    // complete[r2,r1] and complete[r2,r2] and complete[r2,r3] and
    // complete[r3,r1] and complete[r3,r2] and complete[r3,r3]
    Formula rules = f1.and(f2).and(f3);
    for(Relation rx: regions)
      for(Relation ry: regions)
        rules = rules.and(complete(rx,ry));
    return rules;
  }

  public Bounds puzzle() {
    Set<Integer> atoms = new LinkedHashSet<Integer>(9);
    for(int i = 1; i <= 9; i++) { atoms.add(i); }
    Universe u = new Universe(atoms);
    Bounds b = new Bounds(u);
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

```java
    TupleFactory f = u.factory();
    b.boundExactly(Number, f.allOf(1));
    b.boundExactly(regions[0], f.setOf(1, 2, 3));
    b.boundExactly(regions[1], f.setOf(4, 5, 6));
    b.boundExactly(regions[2], f.setOf(7, 8, 9));

    TupleSet givens = f.noneOf(3);
    givens.add(f.tuple(1, 4, 1));
    givens.add(f.tuple(1, 8, 9));
    ...
    givens.add(f.tuple(9, 6, 1));
    b.bound(data, givens, f.allOf(3));
    return b;
  }

  public static void main(String[] args) {
    Solver solver = new Solver();
    solver.options().setSolver(SATFactory.MiniSat);
    Sudoku sudoku = new Sudoku();
    Solution sol = solver.solve(sudoku.rules(), sudoku.puzzle());
    System.out.println(sol);
  }
}
```

Mixing Imperative and Declarative with SQUANDER

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

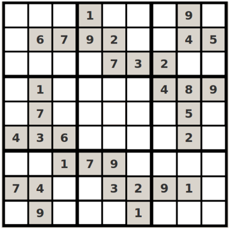
Mixing Imperative and Declarative with SQUANDER

```java
static class Cell { int num = 0; } // 0 means empty

@Invariant("all v: int - 0 | lone {c: this.cells.vals | c.num = v}")
static class CellGroup {
  Cell[] cells;
  public CellGroup(int n) { this.cells = new Cell[n]; }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Mixing Imperative and Declarative with SQUANDER

```
static class Cell { int num = 0; } // 0 means empty

@Invariant("all v: int - 0 | lone {c: this.cells.vals | c.num = v}")
static class CellGroup {
  Cell[] cells;
  public CellGroup(int n) { this.cells = new Cell[n]; }
}

public class Sudoku {
  int n;
  CellGroup[] rows, cols, grids;

  public Sudoku(int n) {
      // (1) create CellGroup and Cell objects,
      // (2) establish sharing of Cells between CellGroups
      init(n);
  }
```



Mixing Imperative and Declarative with SQUANDER

```
static class Cell { int num = 0; } // 0 means empty

@Invariant("all v: int - 0 | lone {c: this.cells.vals | c.num = v}")
static class CellGroup {
  Cell[] cells;
  public CellGroup(int n) { this.cells = new Cell[n]; }
}

public class Sudoku {
  int n;
  CellGroup[] rows, cols, grids;

  public Sudoku(int n) {
      // (1) create CellGroup and Cell objects,
      // (2) establish sharing of Cells between CellGroups
      init(n);
  }

  @Ensures("all c:Cell | c.num > 0 && c.num <= this.n")
  @Modifies("Cell.num | _<1> = 0")
  public void solve() { Squander.exe(this); }
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Mixing Imperative and Declarative with SQUANDER

```
static class Cell { int num = 0; } // 0 means empty

@Invariant("all v: int - 0 | lone {c: this.cells.vals | c.num = v}")
static class CellGroup {
  Cell[] cells;
  public CellGroup(int n) { this.cells = new Cell[n]; }
}

public class Sudoku {
  int n;
  CellGroup[] rows, cols, grids;

  public Sudoku(int n) {
      // (1) create CellGroup and Cell objects,
      // (2) establish sharing of Cells between CellGroups
      init(n);
  }

  @Ensures("all c:Cell | c.num > 0 && c.num <= this.n")
  @Modifies("Cell.num | _<1> = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.rows[0][3].num = 1; s.rows[0][7].num = 9;
    ...
    s.rows[8][1].num = 9; s.rows[8][5].num = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Mixing Imperative and Declarative with SQUANDER

```
static class Cell { int num = 0; } // 0 means empty

@Invariant("all v: int − 0 | lone {c: this.cells.vals | c.num = v}")
static class CellGroup {
  Cell[] cells;
  public CellGroup(int n) { this.cells = new Cell[n]; }
}

public class Sudoku {
  int n;
  CellGroup[] rows, cols, grids;

  public Sudoku(int n) {
     // (1) create CellGroup and Cell objects,
     // (2) establish sharing of Cells between CellGroups
     init(n);
  }

  @Ensures("all c:Cell | c.num > 0 && c.num <= this.n")
  @Modifies("Cell.num | _<1> = 0")
  public void solve() { Squander.exe(this); }

  public static void main(String[] args) {
    Sudoku s = new Sudoku();
    s.rows[0][3].num = 1; s.rows[0][7].num = 9;
    ...
    s.rows[8][1].num = 9; s.rows[8][5].num = 1;
    s.solve();
    System.out.println(s);
  }
}
```

			1				9	
	6	7	9	2			4	5
				7	3	2		
	1					4	8	9
	7						5	
4	3	6					2	
		1	7	9				
7	4			3	2	9	1	
	9				1			

Write more imperative code to make constraints simpler

Everything is a relation

Everything is a relation

				relation name	relation type
classes	$\rightsquigarrow$ unary relations	**class** C {}	$\rightsquigarrow$	$\mathscr{R}_C$	: C
objects	$\rightsquigarrow$ unary relations	**new** C();	$\rightsquigarrow$	$\mathscr{R}_{C_1}$	: C
fields	$\rightsquigarrow$ binary relations	**class** C { A fld ; }	$\rightsquigarrow$	$\mathscr{R}_{fld}$	: $C \rightarrow A \cup \{$**null**$\}$
arrays	$\rightsquigarrow$ ternary relations	T[]	$\rightsquigarrow$	$\mathscr{R}_{T[]\_elems}$	: $T[] \rightarrow$ **int** $\rightarrow T \cup \{$**null**$\}$

Minimizing the Universe Size

Relations in Kodkod

$$r_k \xrightarrow{\text{in Kodkod}} M_{|univ| \times |univ| \times \cdots \times |univ|}$$

a relation of arity k

a matrix of dim $|univ|^k$

Minimizing the Universe Size

Relations in Kodkod

$$\underset{\text{a relation of arity } k}{r_k} \xrightarrow{\text{in Kodkod}} \underset{\text{a matrix of dim } |univ|^k}{M_{|univ| \times |univ| \times \cdots \times |univ|}}$$

so

if $|univ| > 1291 \land (\exists_{r_k} \mid k \geq 3)$

Minimizing the Universe Size

Relations in Kodkod

$$\underbrace{r_k}_{\text{a relation of arity } k} \xrightarrow{\text{in Kodkod}} \underbrace{M_{|univ| \times |univ| \times \cdots \times |univ|}}_{\text{a matrix of dim } |univ|^k}$$

so

if $|univ| > 1291 \wedge (\exists_{r_k} \mid k \geq 3)$

$\implies dim(M) > 1291^3 = 2151685171 > \text{Integer.MAX\_VALUE}$

Minimizing the Universe Size

Relations in Kodkod

$$r_k \xrightarrow{\text{in Kodkod}} M_{|univ| \times |univ| \times \cdots \times |univ|}$$

a relation of arity k — a matrix of dim $|univ|^k$

so

if $|univ| > 1291 \wedge (\exists_{r_k} \mid k \geq 3)$

$\Longrightarrow$ $dim(M) > 1291^3 = 2151685171 >$ Integer.MAX_VALUE

$\Longrightarrow$ can't be represented in Kodkod

Minimizing the Universe Size

Relations in Kodkod

$$r_k \xrightarrow{\text{in Kodkod}} M_{|univ|\times|univ|\times\cdots\times|univ|}$$

a relation of arity k — a matrix of dim $|univ|^k$

so

if $|univ| > 1291 \wedge (\exists_{r_k} \mid k \geq 3)$

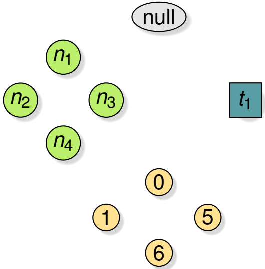
$\implies dim(M) > 1291^3 = 2151685171 >$ Integer.MAX_VALUE

$\implies$ can't be represented in Kodkod

- ternary relations are not uncommon in SQUANDER (e.g. arrays)
- *MIT course scheduler* case study: almost 2000 objects
- **solution**:
  - partitioning algorithm that allows atoms to be shared

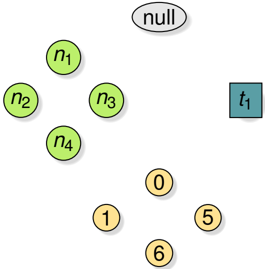
Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects



Minimizing the Universe

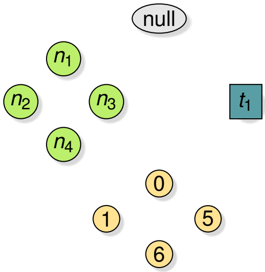
goal: use fewer Kodkod atoms than heap objects
$\rightarrow$ multiple objects must map to same atoms
$\rightarrow$ mapping from objects to atoms is not injective



Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
  → multiple objects must map to same atoms
  → mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
  → *instances of the same type must map to distinct atoms*

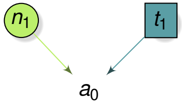


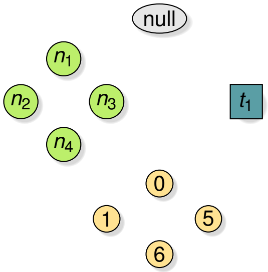
Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

restoring field values (e.g. a_0 for the field `BSTNode.left`)



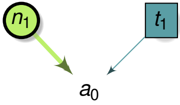


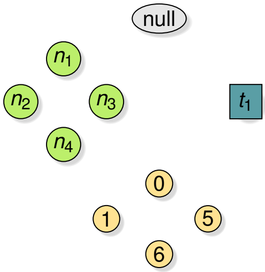
Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

restoring field values (e.g. a_0 for the field `BSTNode.left`)





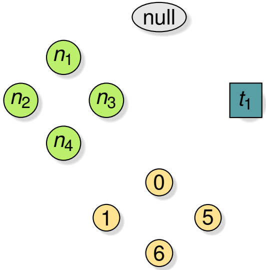
Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
$\to$ multiple objects must map to same atoms
$\to$ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
$\to$ *instances of the same type must map to distinct atoms*

algorithm

1. discover all used types (clusters)



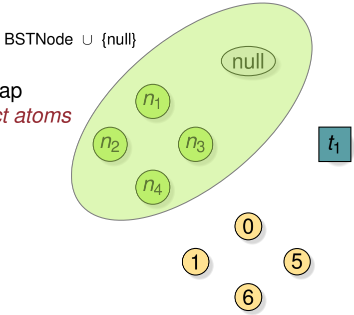
Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

algorithm

1. discover all used types (clusters)



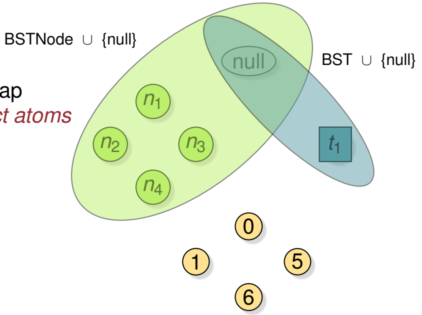
Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

algorithm

1. discover all used types (clusters)



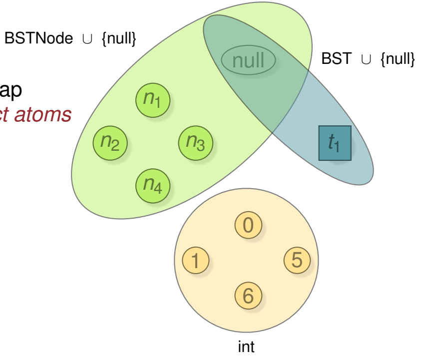
Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
$\rightarrow$ multiple objects must map to same atoms
$\rightarrow$ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
$\rightarrow$ *instances of the same type must map to distinct atoms*

algorithm

1. discover all used types (clusters)



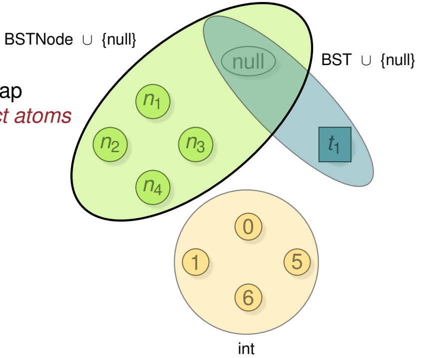
Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
  → multiple objects must map to same atoms
  → mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
  → *instances of the same type must map to distinct atoms*

algorithm

1. discover all used types (clusters)
2. find the largest cluster



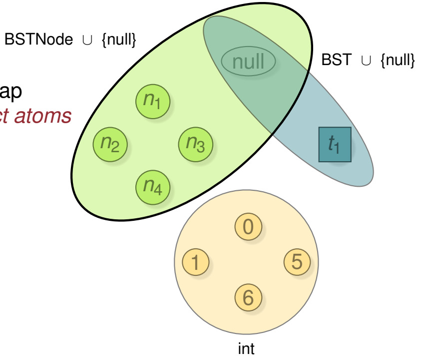
Minimizing the Universe

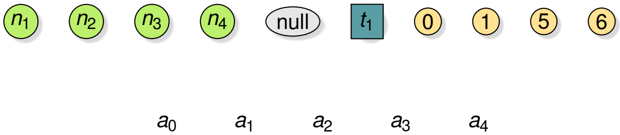
goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

algorithm

1. discover all used types (clusters)
2. find the largest cluster
3. create that many atoms





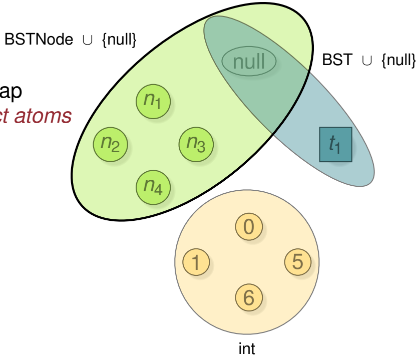
Minimizing the Universe

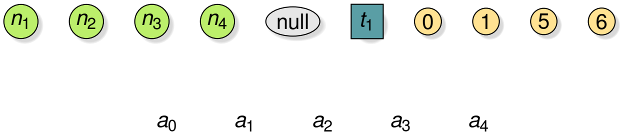
goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

algorithm

1. discover all used types (clusters)
2. find the largest cluster
3. create that many atoms
4. assign atoms to instances

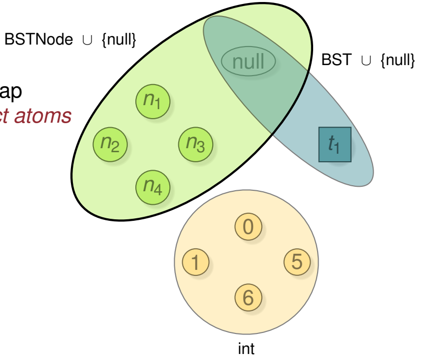




Minimizing the Universe

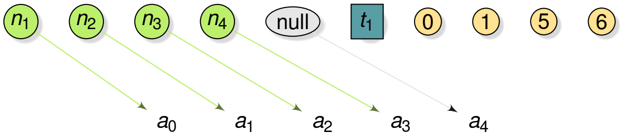
goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*



algorithm

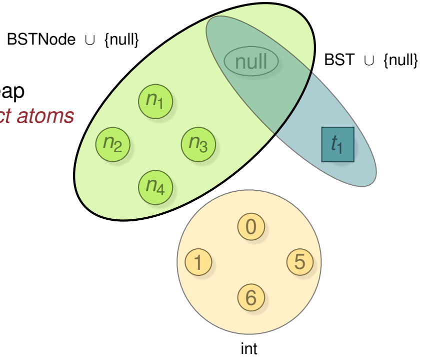
1. discover all used types (clusters)
2. find the largest cluster
3. create that many atoms
4. assign atoms to instances



Minimizing the Universe

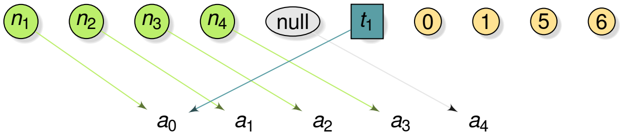
goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*



algorithm

1. discover all used types (clusters)
2. find the largest cluster
3. create that many atoms
4. assign atoms to instances



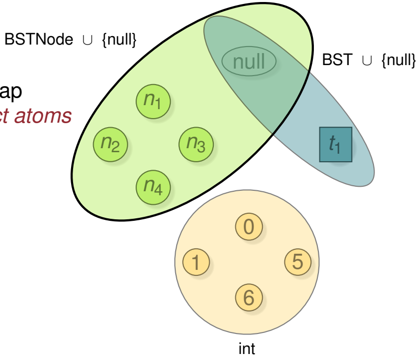
Minimizing the Universe

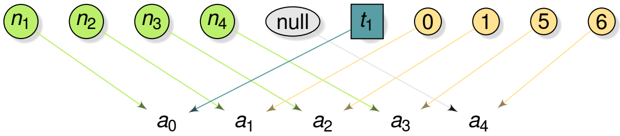
goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

algorithm

1. discover all used types (clusters)
2. find the largest cluster
3. create that many atoms
4. assign atoms to instances

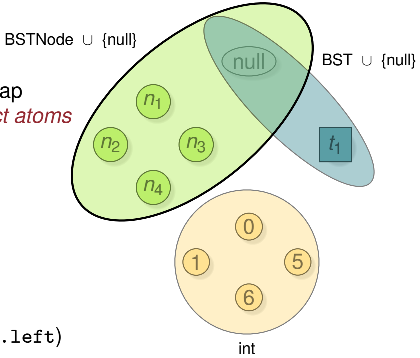




Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
$\rightarrow$ multiple objects must map to same atoms
$\rightarrow$ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
$\rightarrow$ *instances of the same type must map to distinct atoms*

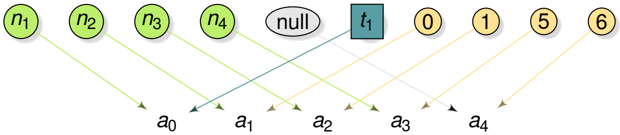


algorithm

1. discover all used types (clusters)
2. find the largest cluster
3. create that many atoms
4. assign atoms to instances

restoring field values (e.g. a_0 for the field `BSTNode.left`)

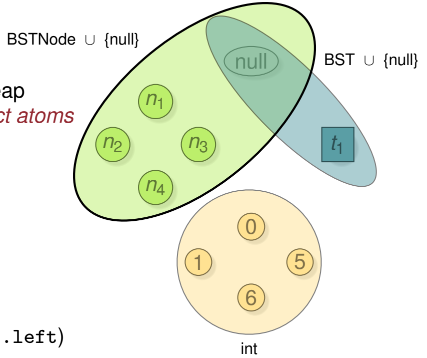
1. based on the field's type, select its cluster
2. select the instance from that cluster that maps to the given atom



Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

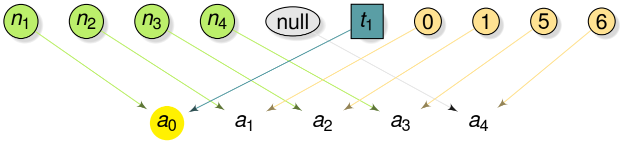


algorithm

1. discover all used types (clusters)
2. find the largest cluster
3. create that many atoms
4. assign atoms to instances

restoring field values (e.g. a_0 for the field `BSTNode.left`)

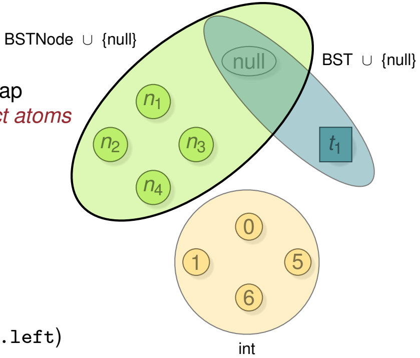
1. based on the field's type, select its cluster
2. select the instance from that cluster that maps to the given atom



Minimizing the Universe

goal: use fewer Kodkod atoms than heap objects
→ multiple objects must map to same atoms
→ mapping from objects to atoms is not injective

also: must be able to unambiguously restore the heap
→ *instances of the same type must map to distinct atoms*

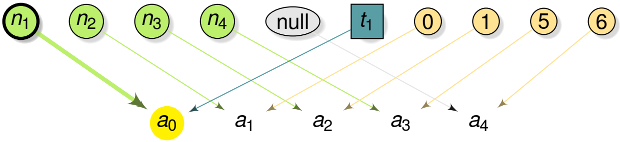


algorithm

1. discover all used types (clusters)
2. find the largest cluster
3. create that many atoms
4. assign atoms to instances

restoring field values (e.g. a_0 for the field `BSTNode.left`)

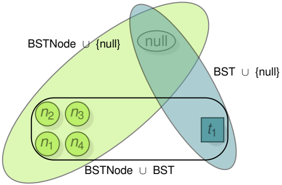
1. based on the field's type, select its cluster
2. select the instance from that cluster that maps to the given atom



Partitioning Algorithm – Discussion

Why is this algorithm sufficient?

- what if we had partitions like this:

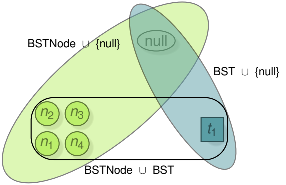


- 5 atoms would not be enough!
- the algorithm would have to discover strongly connected components
- **but**, SQUANDER type checker disallows types like BSTNode ∪ BST

Partitioning Algorithm – Discussion

Why is this algorithm sufficient?

- what if we had partitions like this:



- 5 atoms would not be enough!
- the algorithm would have to discover strongly connected components
- **but**, SQUANDER type checker disallows types like BSTNode ∪ BST

- or a spec like:

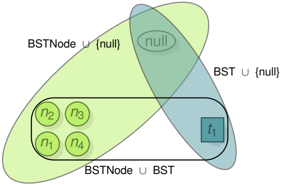
"no BSTNode & int"

- if nodes and ints shared atoms, then the intersection would not be empty!
- **again**, in Java, such expressions don't make much sense, so SQUANDER disallows them.

Partitioning Algorithm – Discussion

Why is this algorithm sufficient?

- what if we had partitions like this:



- 5 atoms would not be enough!
- the algorithm would have to discover strongly connected components
- **but**, SQUANDER type checker disallows types like BSTNode ∪ BST

- or a spec like:

"no BSTNode & int"

- if nodes and ints shared atoms, then the intersection would not be empty!
- **again**, in Java, such expressions don't make much sense, so SQUANDER disallows them.

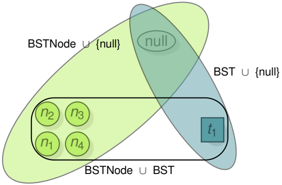
Limitations

- no performance gain

Partitioning Algorithm – Discussion

Why is this algorithm sufficient?

- what if we had partitions like this:



- 5 atoms would not be enough!
- the algorithm would have to discover strongly connected components
- **but**, SQUANDER type checker disallows types like BSTNode ∪ BST

- or a spec like:

"no BSTNode & int"

- if nodes and ints shared atoms, then the intersection would not be empty!
- **again**, in Java, such expressions don't make much sense, so SQUANDER disallows them.

Limitations

- no performance gain
- if a field of type Object is used, this algorithm has no effect
  - everything is a subtype of Object so everything has to go to the same partition

Related Work

Executable Specifications:

- **An Overview of Some Formal Methods for Program Design**, C.A.R. Hoare (*IEEE Computer 1987*)
- **Specifications are not (necessarily) executable**, I. Hayes et al. (*SEJ 1989*)
- **Specifications are (preferably) executable**, N.E. Fuchs (*SEJ 1992*)
- **Programming from Specification**, C. Morgan, PrenticeHall, 1998
- **Agile Specifications**, D. Rayside et al. (*Onward! 2009*)
- **Falling Back on Executable Specifications**, H. Samimi et al. (*ECOOP 2010*)
- **Unified Execution of Imperative and Declarative Code**, A. Milicevic et al. (*ICSE 2011*)

Specification Languages

- **JFSL: JForge Specification Language**, K. Yessenov, MIT 2009
- **Software Abstractions: Logic, Language, and Analysis**, D. Jackson, MIT Press 2006

Programming Languages with Constraint Programming:

- **Jeeves: Programming with Delegation**, J. Yang, MIT, 2010
- **Programming with Quantifiers**, J.P. Near, MIT, 2010