Rascal: Functional programming for source code analysis and transformation

Tijs van der Storm
storm@cwi.nl / @tvdstorm
About me

• Researcher at CWI across the street
• Teacher at here at UvA
  • Master Software Engineering
• Interests:
  • DSLs, MDE, Meta-programming, PL
• Co-designer of Rascal (w/Vinju, Klint)
Meta Software

Analysis
• Dead code detection
• Slicing/Dependence
• Metrics
• Reverse engineering
• Verification
• Architecture recovery
• Code-to-model
• ...

Transformation
• Goto elimination
• Dialect transformation
• Aspect weaving
• DSL compilers
• API migration
• Model-to-code
• Model-to-model
• ...

Friday, January 10, 14
Example application: DSLs
Rascal

http://www.rascal-mpl.org

https://github.com/cwi-swat/rascal
Well-known FP stuff

• Higher-order functions
• Algebraic data types
• Immutable data
• Pattern-matching (but...)
• Comprehensions (but...)
More notable

• “Java-inspired” syntax (Blasphemy! ;-) 
• Built-in context free grammars & parsing 
• Concrete syntax patterns and values 
• Traversal primitives 
• Relational calculus 
• IDE hooks
Extraction and analysis
module Syntax
extend lang::std::Layout;

start syntax Controller =
    controller:
        Events events
        ResetEvents? resets
        Commands? commands
        State+ states;

syntax Events
    = "events" Event* "end";
syntax ResetEvents
    = "resetEvents" Id* "end";
syntax Commands
    = "commands" Command* "end";
Syntax definition

module Syntax
extend lang::std::Layout;

start syntax Controller =
controller:
  Events events
  ResetEvents? resets
  Commands? commands
  State+ states;

syntax Events
  = "events" Event* "end";
syntax ResetEvents
  = "resetEvents" Id* "end";
syntax Commands
  = "commands" Command* "end";
module Syntax
extend std::Layout;

start syntax Controller =
controller:
    Events events
ResetEvents? resets
Commands? commands
State+ states;

syntax Events  
    = "events" Event* "end";
syntax ResetEvents
    = "resetEvents" Id* "end";
syntax Commands
    = "commands" Command* "end";
module Syntax
extend lang::std::Layout;

start syntax Controller =
  controller:
    Events events
    ResetEvents? resets
    Commands? commands
    State+ states;

syntax Events = "events" Event* "end";
syntax ResetEvents = "resetEvents" Id* "end";
syntax Commands = "commands" Command* "end";
module Syntax
extend lang::std::Layout;

start syntax Controller =
controller:
  Events events
  ResetEvents? resets
  Commands? commands
  State+ states;

syntax Events
  = "events" Event* "end";
syntax ResetEvents
  = "resetEvents" Id* "end";
syntax Commands
  = "commands" Command* "end";
Lexical syntax

**lexical** Id
   = ([a-zA-Z][a-zA-Z0-9_]*)!>> [a-zA-Z0-9_])
   \ Reserved ;

**keyword** Reserved
   = "events"
   | "end"
   | "resetEvents"
   | "state"
   | "actions" ;
Lexical syntax

lexical Id
   = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
\ Reserved ;

keyword Reserved
   = "events"
    | "end"
    | "resetEvents"
    | "state"
    | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
  \ Reserved ;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
\ Reserved ;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
\ Reserved ;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
\  Reserved ;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions" ;
Parse trees
Friday, January 10, 14
Abstract Syntax

data Controller = controller(list[Event] events,
                list[str] resets,
                list[Command] commands,
                list[State] states);

data State = state(str name,
                list[str] actions,
                list[Transition] transitions);

data Command = command(str name, str token);
data Event = event(str name, str token);
data Transition = transition(str event, str state);
controller(pannel("doorClosed", "DCL"),
    event("doorOpened", "D20P")
)
Pattern matching

```plaintext
int x := 3;

event(x, y) := event("a", "b");

event("c", "d") !:= event("a", "b");

[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];

{1, *x} := {4, 5, 6, 1, 2, 3};

/transition(e, "idle") := ast;
/state(x, _, /transition(_, x)) := ast;

3 <- {1,2,3}
int x <- {1,2,3}
```
Pattern matching

```plaintext
int x := 3;

event(x, y) := event("a", "b");

event("c", "d") !:= event("a", "b");

[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];

{1, *x} := {4, 5, 6, 1, 2, 3};

/transition(e, "idle") := ast;
/state(x, _, /transition(_, x)) := ast;

3 <- {1,2,3}
int x <- {1,2,3}
```
Pattern matching

**type-based matching**

```plaintext
int x := 3;

event(x, y) := event("a", "b");

event("c", "d") !:= event("a", "b");

[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];

{1, *x} := {4, 5, 6, 1, 2, 3};

/transition(e, "idle") := ast;
/state(x, _, /transition(_, x)) := ast;

3 <- {1,2,3}
int x <- {1,2,3}
```

**structural matching**
Pattern matching

int x := 3;

event(x, y) := event("a", "b");

event("c", "d") !:= event("a", "b");

[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];

{1, *x} := {4, 5, 6, 1, 2, 3};

/transition(e, "idle") := ast;
/state(x, _, /transition(_, x)) := ast;

3 <- {1,2,3}
int x <- {1,2,3}
Pattern matching

int x := 3;

event(x, y) := event("a", "b");

event("c", "d") !:= event("a", "b");

[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];

{1, *x} := {4, 5, 6, 1, 2, 3};

/transition(e, "idle") := ast;
/state(x, _, /transition(_, x)) := ast;

3 <- {1,2,3}

int x <- {1,2,3}
Pattern matching

- type-based matching
- structural matching
- anti-matching
- list matching
- set matching

```plaintext
int x := 3;

event(x, y) := event("a", "b");

event("c", "d") !:= event("a", "b");

[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];

{1, *x} := {4, 5, 6, 1, 2, 3};

/transition(e, "idle") := ast;
/state(x, _, /transition(_, x)) := ast;

3 <- {1,2,3}
int x <- {1,2,3}
```
Pattern matching

- type-based matching
- structural matching
- anti-matching
- list matching
- set matching
- deep matching

```plaintext
int x := 3;

event(x, y) := event("a", "b");

event("c", "d") !:= event("a", "b");

[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];

{1, *x} := {4, 5, 6, 1, 2, 3};

/transition(e, "idle") := ast;
/state(x, _, /transition(_, x)) := ast;

3 <- {1,2,3}
int x <- {1,2,3}
```
Pattern matching

- **type-based matching**
  - `int x := 3;`

- **structural matching**
  - `event(x, y) := event("a", "b");`
  - `event("c", "d") !:= event("a", "b");`
  - `[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];`
  - `{1, *x} := {4, 5, 6, 1, 2, 3};`
  - `/transition(e, "idle") := ast;`
  - `/state(x, _, /transition(_, x)) := ast;`

- **anti-matching**

- **list matching**

- **set matching**

- **deep matching**

- **element matching**
  - `3 <- {1,2,3}`
  - `int x <- {1,2,3}`
Backtracking contexts

rascal>for ([*x, *y] := [1,1,1,1,1,1]) println("<x> <y>");
[] [1,1,1,1,1,1]
[1] [1,1,1,1,1,1]
[1,1] [1,1,1,1,1]
[1,1,1] [1,1,1]
[1,1,1,1] [1,1]
[1,1,1,1,1] [1]
[1,1,1,1,1,1] []
Backtracking contexts

rascal>for ([*x, *y] := [1,1,1,1,1,1]) println("<x> <y>");
[] [1,1,1,1,1,1]
[1] [1,1,1,1,1,1]
[1,1] [1,1,1,1,1]
[1,1,1] [1,1,1]
[1,1,1,1] [1,1]
[1,1,1,1,1] [1]
[1,1,1,1,1,1] []

rascal>for ([*x, *y] := [1,1,1,1,1,1], x == y) println("<x> <y>");
[1,1,1] [1,1,1]
Set matching

rascal>for ({{x, y} := {1,2,3,4}}) println("<x> <y>");
{4,3,2,1} {}
{4,3,2} {1}
{4,3,1} {2}
{4,3} {2,1}
{4,2,1} {3}
{4,2} {3,1}
{4,1} {3,2}
{4} {3,2,1}
{3,2,1} {4}
{3,2} {4,1}
{3,1} {4,2}
{3} {4,2,1}
{2,1} {4,3}
{2} {4,3,1}
{1} {4,3,2}
{} {4,3,2,1}
Comprehensions

- list: $[ i \mid i \leftarrow [1..100], i \% 2 == 0 ]$;
- map: $( i : i*i \mid i \leftarrow [1..10] )$;
- set & relation: ${ <i, i*i> \mid i \leftarrow [1..10] }$;
Relational calculus

```c
r = {
    "active","waitingForDrawer">,
    "idle","active">,
    "unlockedPanel","idle">,
    "waitingForLight","unlockedPanel">,
    "active","waitingForLight">,
    "waitingForDrawer","unlockedPanel">;

r<0>;  

r<1,0>;  

r["active"];  

r+;  

r*;  

r o r
```
Relational calculus

\[ r = \{ \langle \text{"active"}, \text{"waitingForDrawer"} \rangle, \langle \text{"idle"}, \text{"active"} \rangle, \langle \text{"unlockedPanel"}, \text{"idle"} \rangle, \langle \text{"waitingForLight"}, \text{"unlockedPanel"} \rangle, \langle \text{"active"}, \text{"waitingForLight"} \rangle, \langle \text{"waitingForDrawer"}, \text{"unlockedPanel"} \rangle \} \]

- \( r < 0 >; \)
- \( r < 1, 0 >; \)
- \( r["active"]\);
- \( r +; \)
- \( r *; \)
- \( r \circ r \)
Relational calculus

$r = \{<"active","waitingForDrawer">,
<"idle","active">,
<"unlockedPanel","idle">,
<"waitingForLight","unlockedPanel">,
<"active","waitingForLight">,
<"waitingForDrawer","unlockedPanel">\}$

projection

$r\{0\}$

invert

$r\{1,0\}$

$r["active"]$

$r+$

$r*$

$r \circ r$
Relational calculus

\[ r = \{ \]
\[ \langle \text{"active"}, \text{"waitingForDrawer"} \rangle, \]
\[ \langle \text{"idle"}, \text{"active"} \rangle, \]
\[ \langle \text{"unlockedPanel"}, \text{"idle"} \rangle, \]
\[ \langle \text{"waitingForLight"}, \text{"unlockedPanel"} \rangle, \]
\[ \langle \text{"active"}, \text{"waitingForLight"} \rangle, \]
\[ \langle \text{"waitingForDrawer"}, \text{"unlockedPanel"} \rangle \}; \]

projection

invert

\[ r<0>; \]
\[ r<1,0>; \]
\[ r["active"]; \]
\[ r+; \]
\[ r*; \]
\[ r \circ r \]
Relational calculus

\[ r = \{ \langle \text{"active"}, \text{"waitingForDrawer"} \rangle, \langle \text{"idle"}, \text{"active"} \rangle, \langle \text{"unlockedPanel"}, \text{"idle"} \rangle, \langle \text{"waitingForLight"}, \text{"unlockedPanel"} \rangle, \langle \text{"active"}, \text{"waitingForLight"} \rangle, \langle \text{"waitingForDrawer"}, \text{"unlockedPanel"} \rangle \}; \]

- projection: \( r < 0 \);
- invert: \( r < 1, 0 \);
- transitive closure: \( r + \);
- right image: \( r * \);
- power set: \( r \circ r \).
Relational calculus

\[ r = \{ <"active","waitingForDrawer">, <"idle","active">, <"unlockedPanel","idle">, <"waitingForLight","unlockedPanel">, <"active","waitingForLight">, <"waitingForDrawer","unlockedPanel"> \}; \]

- projection: \( r<0>; \)
- invert: \( r<1,0>; \)
- transitive closure: \( r+; \)
- transitive reflexive closure: \( r*; \)
- right image: \( r \circ r \)
Relational calculus

\[ r = \{ <\text{"active"}, \text{"waitingForDrawer"}>, <\text{"idle"}, \text{"active"}>, <\text{"unlockedPanel"}, \text{"idle"}>, <\text{"waitingForLight"}, \text{"unlockedPanel"}>, <\text{"active"}, \text{"waitingForLight"}>, <\text{"waitingForDrawer"}, \text{"unlockedPanel"}> \} \]

- Projection: \( r <0> \)
- Invert: \( r <1,0> \)
- Right image: \( r[\text{"active"]} \)
- Transitive closure: \( r^+ \)
- Transitive reflexive closure: \( r^* \)
- Relation composition: \( r \circ r \)
Transformation
Transformation in Rascal

- Functional programming :-)  
- Type-preserving visit
Visit

- Similar to case-based match construct
- Visits all nodes of data structure
- Specify cases of interest only
  - “Structure shy”
- Bottom-up, top-down, innermost, outermost strategies
Print all state names

```java
visit (ast) {
    case state(str name, _, _): println(name);
}
```

Controller desugar(Controller ctl) {
    init = ctl.states[0].name;
    ctl = visit (ctl) {
        case state(n, as, ts) => state(n, as, ts + nts)
            when nts := [ transition(e, init) | e <- ctl.resets ]
    };
    ctl.resets = [];
    return ctl;
}
Traversals strategies

bottom-up visit (ast) {
  case state(str name, _, _): println(name);
}

top-down visit (ast) {
  case state(str name, _, _): println(name);
}
Analysis & transformation in DSL implementation

- Analysis: parsing, name resolution, type checking, model checking, etc.
- Transformation: desugaring, visualization, refactoring, optimization, compilation, etc.
Concrete syntax
Abstract syntax
Unparse
Desugaring
Checking
Outline
Hyperlinking
Compilation
Visualization
Rename refactoring
Parallel merge
Concluding

- Functional programming for source code analysis and transformation
- Grammars, ADTs, pattern matching, comprehensions, relational calculus, ...
- Hooks into the Eclipse IDE
- Language workbench