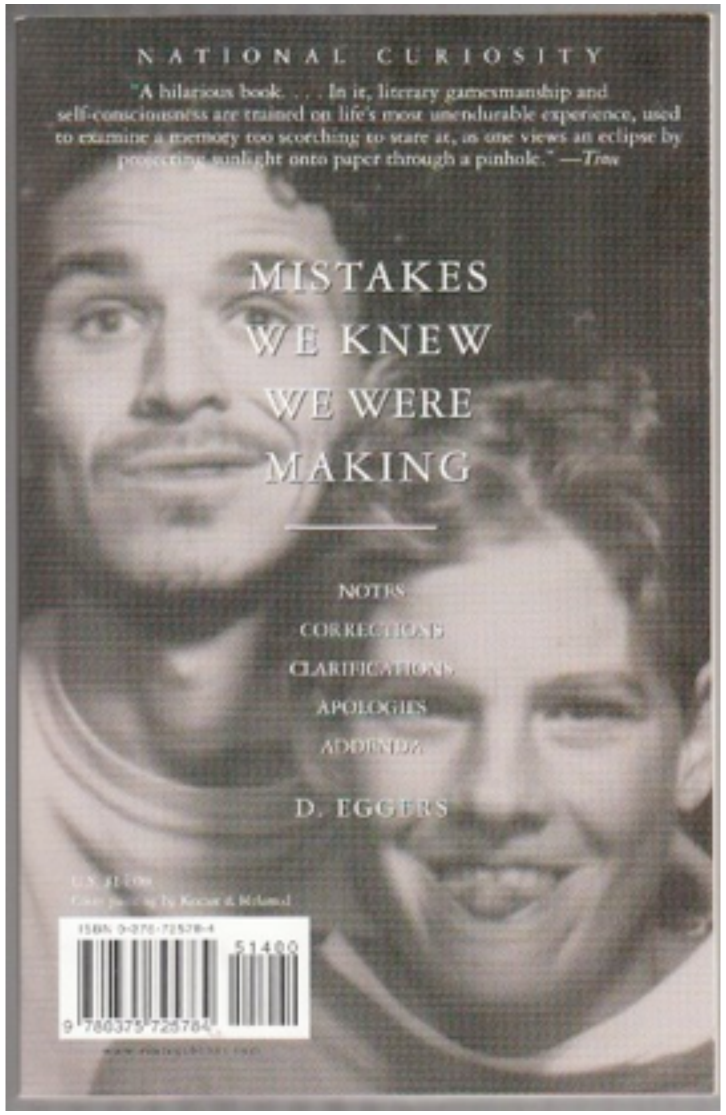
ORTHOGONALITY

Orthogonality

- Rascal, a heartbreaking work of staggering genius ;-)
- Some mistakes we have made...
- or we are about to make...
- or not...
- have to do with orthogonality...



STICHTING

MATHEMATISCH CENTRUM

2e BOERHAAVESTRAAT 49

AMSTERDAM

REKENAFDELING

MR 76

Orthogonal design

and description of

a formal language

by

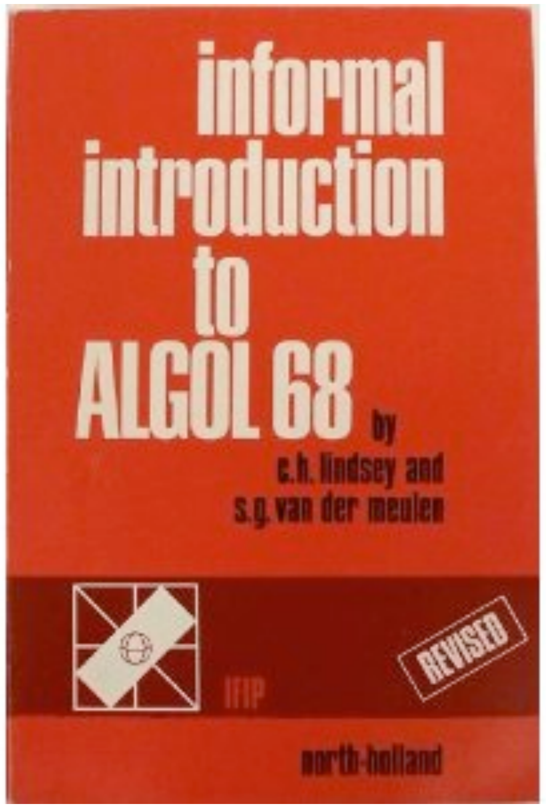
A. van Wijngaarden

IFIP WG 2.1, October 1965

As to the design of a language I should like to see the definition of a language as a Cartesian product of its concepts.

Algol 68

- procedures as params
- values as params
- values can be assigned
- so procedures can be assigned.



Cartesian product

	assign	pass
expr	yes	yes
proc	?	yes

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-224

STAN-CS-73-403

HINTS ON PROGRAMMING LANGUAGE DESIGN

BY

C. A. R. HOARE

Another replacement of simplicity as an objective has been orthogonality of design. An example of orthogonality is the provision of complex integers, on the argument that we need reals and integers and complex reals, so why not complex integers? In the early days of hardware design, some very ingenious but arbitrary features turned up in order codes as a result of orthogonal combinations of the function bits of an instruction, on the grounds that some clever programmer would find a use for them, -- and some clever programmer always did. Hardware designers have now learned more sense; but language designers are clever programmers and have not.

ON THE DESIGN OF PROGRAMMING LANGUAGES*

N. WIRTH

*Reprinted from *Proc. IFIP Congress 74*, 386–393, North-Holland, Amsterdam, North-Holland Publishing Company.

The author is with the Institut für Informatik, Eidg. Technische Hochschule, Zurich, Switzerland.

The new trend was to discover the fundamental concepts of algorithms, to extract them from their various incarnations in different language features, and to present them in a pure, distilled form, free from arbitrary and restrictive rules of applicability.

Backfiring orthogonality

```
( real x,y;                    {  Declare two local variables.              }
  read((x,y));                 {  Read two values in them from input.       }
  if x < y then a else b       {  Choose one of them as the                 }
  fi                           {  left hand side of the assignment.         }
 ) :=                          {  The right hand side consists of b and     }
   b +                         {  a conditionally selected second           }
     if a:= a+1; a > b         {  operand.  If a > b, increment a,          }
     then                      {  and the second operand is                 }
       c:=c+1; +b              {  +b, increment c in the meanwhile          }
     else                      {  If a is not > b, then the second          }
       c:=c-1; a               {  operand is a, decrement c.                }
     fi                        {  End of selection of the second operand }
```

Algol 68

HUMAN-COMPUTER INTERACTION, 1989, Volume 4, pp. 95–120

Testing the Principle of Orthogonality in Language Design

Edward M. Bowden, Sarah A. Douglas, and
Cathryn A. Stanford
University of Oregon

A bit of history...

```
module Booleans
  exports
    sorts BOOL
    lexical syntax
      [\t \ n]                    → LAYOUT
    context-free syntax
      true                        → BOOL
      false                       → BOOL
      BOOL "&" BOOL               → BOOL {left}
  equations
  [B1]  true & true               = true
  [B2]  true & false              = false
  [B3]  false & true              = false
  [B4]  false & true              = false
module Naturals
  imports Booleans
  exports
    sorts NAT
    context-free syntax
      "0"                         → NAT
      succ "(" NAT ")"            → NAT
      NAT " < " NAT               → BOOL
    variables
      N → NAT
      M → NAT
  equations
  [N1]  0 < 0                     = false
  [N2]  succ (N) < 0              = false
  [N3]  0 < succ(N)               = true
  [N4]  succ(N) < succ(M)         = N < M
```

- ASF+SDF
- "Just" two concepts
- Beautiful
- Orthogonal!
- Unusable

Rascal

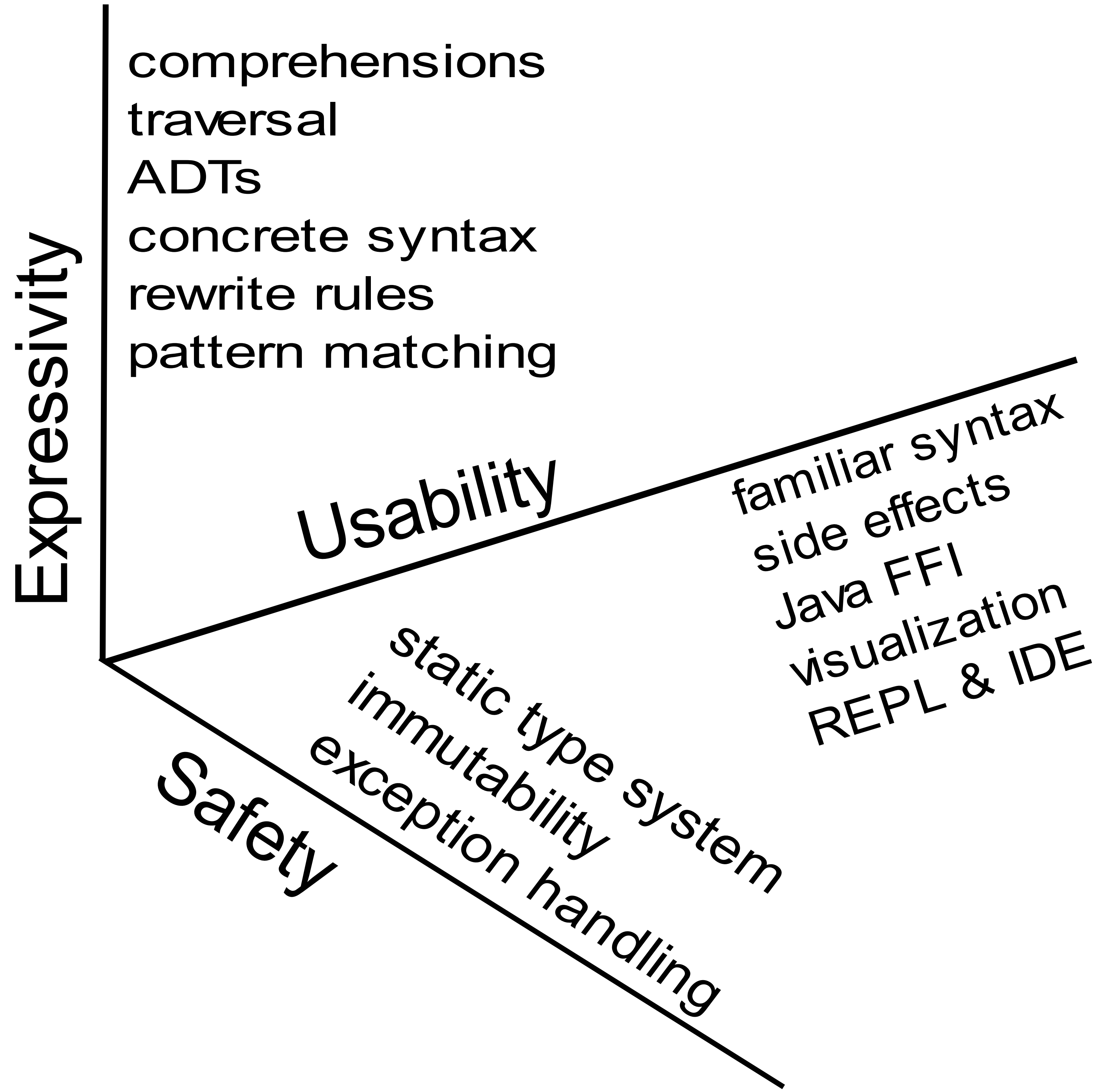
- Functional meta-programming language
- DSL implementation and program understanding/renovation
- Source code in, source code out
- Source code in the broadest sense

http://www.rascal-mpl.org

Rascal’s Unique (?) features

- Integrated context-free grammars
- Very powerful pattern matching
- Transitive closure, solve statement
- Resources (Type providers reloaded)
- Source location data type
- Built-in (randomized) testing features

http://www.rascal-mpl.org


Expressivity
comprehensions
traversal
ADTs
concrete syntax
rewrite rules
pattern matching
Usability
familiar syntax
side effects
Java FFI
visualization
REPL & IDE
Safety
static type system
immutability
exception handling

Language design

- Design = hypothesis
- Observe use in practice
- Revise design if needed
- Learn by doing!
- Today: questions more than answers

A taste of Rascal

Relational calculus

```
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

set of tuples

```
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

set of tuples

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

projection

```
r<0>;

r<1,0>;

r["active"];

r+;

r*;

r o r
```

Relational calculus

set of tuples

```
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 o r
```

Relational calculus

set of tuples

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

projection

```
r<0>;
```

invert

```
r<1,0>;
```

```
r["active"];
```

right image

```
r+;

r*;

r o r
```

Relational calculus

set of tuples

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

projection

```
r<0>;
```

invert

```
r<1,0>;
```

right image

```
r["active"];
```

transitive closure

```
r+;

r*;

r o r
```

Relational calculus

set of tuples

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

projection

```
r<0>;
```

invert

```
r<1,0>;
```

right image

```
r["active"];
```

transitive closure

```
r+;
```

transitive reflexive closure

```
r*;
```

```
r o r
```

Relational calculus

set of tuples

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

projection

```
r<0>;
```

invert

```
r<1,0>;
```

right image

```
r["active"];
```

transitive closure

```
r+;
```

transitive reflexive closure

```
r*;
```

relation composition

```
r o r
```

Relations...

Container	Equivalent type	Operations
set[tuple[...]]	rel	_o_, _+, _*, _[_]
list[tuple[...]]	orel	same?
bag[tuple[...]]	mrel	same?
map	map	same?

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}
```

Matching

type-based 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}
```

Matching

type-based matching

structural 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}
```

Matching

type-based matching

structural matching

anti-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}
```

Matching

type-based matching

structural matching

anti-matching

list 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}
```

Matching

type-based matching

structural matching

anti-matching

list matching

set 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}
```

Matching

type-based matching

```
int x := 3;
```

structural matching

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

anti-matching

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

list matching

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

set matching

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

deep matching

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

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

Matching

type-based matching

```
int x := 3;
```

structural matching

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

anti-matching

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

list matching

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

set matching

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

deep matching

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

element matching

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

list-matching

```
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] []
```

list-matching

```
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] []
```

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

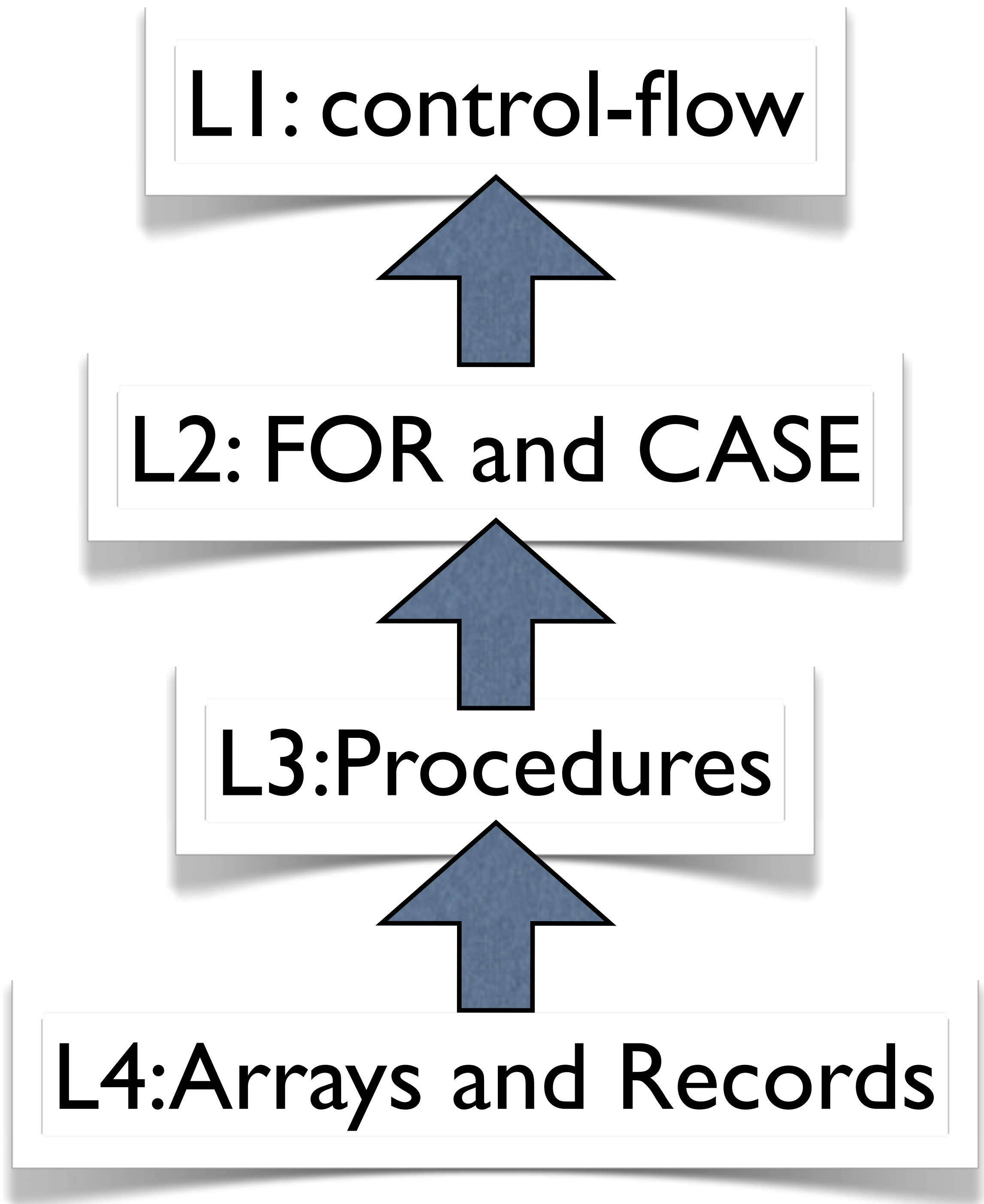
set-matching

```
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}
```

Collection types

Collection	Matching
Lists	Associative
Bags	Associative, commutative 
Sets	Associative, commutative, idempotent

Language extensibility: LDTA'11 ToolChallenge



A simple interpreter

```
data Exp
  = add(Exp lhs, Exp rhs)
  | lit(int n)
  ;

public int eval0(Exp e) {
  switch (e) {
    case add(l, r): return eval(l) + eval(r);
    case lit(n): return n;
  }
}
```

Extension

```
module Neg

extend Add;

data Exp = neg(Exp arg);
```

Extension

```
module Neg

extend Add;

data Exp = neg(Exp arg);
```

How to extend
the interpreter?

```
public int eval0(Exp e) {
  switch (e) {
    case add(l, r): return eval(l) + eval(r);
    case lit(n): return n;
  }
}
```

Pattern-based dispatch

- Open up "switch"
- Allow arbitrary *patterns* in function signatures
- Liberalize overloading of functions...

Open interpreter

```
module Add

data Exp = add(Exp lhs, Exp rhs) | lit(int n);

public int eval1(add(l, r)) = eval1(l) + eval1(r);
public int eval1(lit(n)) = n;
```

Open interpreter

```
module Add

data Exp = add(Exp lhs, Exp rhs) | lit(int n);

public int eval1(add(l, r)) = eval1(l) + eval1(r);
public int eval1(lit(n)) = n;
```

```
module Neg
extend Add;

data Exp = neg(Exp arg);

public int eval1(neg(a)) = - eval1(a);
```

Traversal using *visit*

```
public Exp propagate0(Exp e) {
  return innermost visit (e) {
    case add(lit(a), lit(b)) => lit(a + b)
  }
}
```

Traversal using *visit*

traversal
strategy

```
public Exp propagate0(Exp e) {
  return innermost visit (e) {
    case add(lit(a), lit(b)) => lit(a + b)
  }
}
```

Traversal using *visit*

traversal strategy

```
public Exp propagate0(Exp e) {
  return innermost visit (e) {
    case add(lit(a), lit(b)) => lit(a + b)
  }
}
```

structure shy

Traversal using *visit*

traversal strategy

```
public Exp propagate0(Exp e) {
  return innermost visit (e) {
    case add(lit(a), lit(b)) => lit(a + b)
  }
}
```

structure shy

type preserving

Feature	“Open”
switch	pattern-based dispatch
visit	? 

Visit using functions

```
module Add
public Exp propStep(add(lit(a), lit(b))) = lit(a + b);

public Exp propagate1(Exp e) = innermost visit (e, propStep);
```

!!!

Visit using functions

```
module Add
public Exp propStep(add(lit(a), lit(b))) = lit(a + b);

public Exp propagate1(Exp e) = innermost visit (e, propStep);
```

!!!

```
module Neg
extend Add;

public Exp propStep(neg(lit(n))) = lit(-n);
```

Comprehensions

```
[ i | i <- [1..100], i % 2 == 0 ];

( i: i*i | i <- [1..10] );

{ <i, i*i> | i <- [1..10] };
```

Comprehensions

list

```
[ i | i <- [1..100], i % 2 == 0 ];

( i: i*i | i <- [1..10] );

{ <i, i*i> | i <- [1..10] };
```

Comprehensions

list

```
[ i | i <- [1..100], i % 2 == 0 ];
```

map

```
( i: i*i | i <- [1..10] );
```

```
{ <i, i*i> | i <- [1..10] };
```

Comprehensions

list

```
[ i | i <- [1..100], i % 2 == 0 ];
```

map

```
( i: i*i | i <- [1..10] );
```

set & relation

```
{ <i, i*i> | i <- [1..10] };
```

Higher-order reduce

```
public &T reduce(list[&T] l, &T init, &T(&T, &T) op) {
  &T n = init;
  for (e <- l)
    n = op(e, n);
  return n;
}

public int sum1(list[int] l) =
  reduce(l, 0, int(int e,int a) { return e + a; });
```

Put in a library

Clunky, needs if's for conditions

Ugly because of types and curly syntax

Reducers

```
public int sum(list[int] l) =
     ( 0 | it + x | x <- l );
```

accumulator

```
public int sumEven(list[int] l) =
  ( 0 | it + x | x <- l, x % 2 == 0 );
```

like comprehensions

Folds...

Type...	Collection	Tree
Preserving	comprehension	*visit*
Transforming	comprehension	? 
Unifying	reducer	? 

Summarizing

- Set of tuple is a *rel*: why not *mrel* and *orel*?
- Sets and list: why not bags?
- Open *switch*: why not open *visit*?
- Folds over collections: why not over trees?

Orthogonal

pure
hard to implement
terse
minimalism
modernism
one-way-to-do-it
dead corners
compositional
general
complex
clean

?

Non-

Domain-specific:
Most things discussed in context of Rascal are general purpose stuff.

ad hoc
keyworditis
baroque
post-modernism
eclectic
many-ways-to-do-it
to the point
non-compositional
domain-specific
direct
dirty

Scylla & Charybdis

Algol 68,
Smalltalk,
Haskell

ABAP, Cobol,
4GL etc.

- Orthogonality = design constraint
- Minimize concepts, maximize combinatorics
- More concepts => orthogonality is harder
- Trade-offs: slippery slope, turing tarpit, simplicity lost

Orthogonality by Stefano Bertolo CC BY-NC-SA 2.0

http://www.rascal-mpl.org