On Functional Logic Programming and its Application to Testing

Sebastian Fischer

Programming Languages and Compiler Construction
Department of Computing Science
Christian-Albrechts University of Kiel

Software Correctness

Astronauts died because of programming errors:

- In *2001: A Space Odyssey* (by Arthur C. Clarke) computer killed astronauts because of **programming contradiction**
- In 1996, unmanned *Ariane 5* rocket exploded on its first flight because of **error in software design**

Software Correctness

Astronauts died because of programming errors:

- In *2001: A Space Odyssey* (by Arthur C. Clarke) computer killed astronauts because of **programming contradiction**
- In 1996, unmanned *Ariane 5* rocket exploded on its first flight because of **error in software design**

How to avoid errors in software:

- prove program correct regarding specification
- test program properties with example input

Software Correctness

Astronauts died because of programming errors:

- In *2001: A Space Odyssey* (by Arthur C. Clarke) computer killed astronauts because of **programming contradiction**
- In 1996, unmanned *Ariane 5* rocket exploded on its first flight because of **error in software design**

How to avoid errors in software:

- prove program correct regarding specification
- test program properties with example input
- use declarative programming language

Software Correctness

Astronauts died because of programming errors:

- In *2001: A Space Odyssey* (by Arthur C. Clarke) computer killed astronauts because of **programming contradiction**
- In 1996, unmanned *Ariane 5* rocket exploded on its first flight because of **error in software design**

How to avoid errors in software:

- prove program correct regarding specification
- test program properties with example input
- use declarative programming language

Even declarative programs can contain bugs!

Imperative Programming

```
// reverse an array beginning at position 0
pos = 0;

// loop through whole array
while (pos < array.size) {

  // swap two elements
  elem = array[pos];
  array[pos] = array[array.size - pos];
  array[array.size - pos] = elem;

  // advance to next position
  pos = pos + 1;
}
```

Imperative Programming

```
// reverse an array beginning at position 0
pos = 0;

// loop through whole array
while (pos < array.size) {

  // swap two elements
  elem = array[pos];
  array[pos] = array[array.size - pos];
  array[array.size - pos] = elem;

  // advance to next position
  pos = pos + 1;
}
```

off-by-one error!

Imperative Programming

```
// reverse an array beginning at position 0
pos = 0;

// loop through whole array
while (pos < array.size) {

  // swap two elements
  elem = array[pos];
  array[pos] = array[array.size - pos - 1];
  array[array.size - pos - 1] = elem;

  // advance to next position
  pos = pos + 1;
}
```

Imperative Programming

```
// reverse an array beginning at position 0
pos = 0;

// loop through whole array
while (pos < array.size) {

  // swap two elements
  elem = array[pos];
  array[pos] = array[array.size - pos - 1];
  array[array.size - pos - 1] = elem;

  // advance to next position
  pos = pos + 1;
}
```

Imperative Programming

```
pos   = 0
array = [ 1 , 2 , 3 , 4 ]
```

Imperative Programming

```
pos   = 0
array = [ 1 , 2 , 3 , 4 ]

pos   = 1
array = [ 4 , 2 , 3 , 1 ]
```

Imperative Programming

```
pos   = 0
array = [ 1 , 2 , 3 , 4 ]

pos   = 1
array = [ 4 , 2 , 3 , 1 ]

pos   = 2
array = [ 4 , 3 , 2 , 1 ]
```

Imperative Programming

```
pos   = 0
array = [ 1 , 2 , 3 , 4 ]

pos   = 1
array = [ 4 , 2 , 3 , 1 ]

pos   = 2
array = [ 4 , 3 , 2 , 1 ]

pos   = 3
array = [ 4 , 2 , 3 , 1 ]
```

Imperative Programming

```
pos   = 0
array = [ 1 , 2 , 3 , 4 ]

pos   = 1
array = [ 4 , 2 , 3 , 1 ]

pos   = 2
array = [ 4 , 3 , 2 , 1 ]      ← stop here!

pos   = 3
array = [ 4 , 2 , 3 , 1 ]

pos   = 4
array = [ 1 , 2 , 3 , 4 ]
```

Imperative Programming

```
// reverse an array beginning at position 0
pos = 0;

// loop through half array
while (pos < array.size / 2) {

  // swap two elements
  elem = array[pos];
  array[pos] = array[array.size - pos - 1];
  array[array.size - pos - 1] = elem;

  // advance to next position
  pos = pos + 1;
}
```

Imperative Programming

- **program is sequence of statements**
- **built-in control structures (loops, conditional branches)**
- **values of variables change (side effects)**
- **results depend on evaluation order**
- **easy to make mistakes (index manipulations)**

Functional Programming

Language: Haskell

$$\begin{aligned}
\mathbf{reverse}\ [\,] &= [\,] \\
\mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) &= \mathbf{reverse}\ \mathbf{xs} +\!\!\!+ (\mathbf{x} : [\,])
\end{aligned}$$

Functional Programming

Language: Haskell

```
reverse []       = []
reverse (x : xs) = reverse xs ++ (x : [])
```

```
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Functional Programming

$$\begin{aligned}
\mathbf{reverse}\;[\,] &= [\,] \\
\mathbf{reverse}\;(\mathbf{x} : \mathbf{xs}) &= \mathbf{reverse}\;\mathbf{xs} \mathbin{+\!\!+} (\mathbf{x} : [\,])
\end{aligned}$$

$$\begin{aligned}
&\mathbf{reverse}\;(1 : 2 : 3 : 4 : [\,]) \\
&\quad = \mathbf{reverse}\;(2 : 3 : 4 : [\,]) \qquad\qquad\qquad\qquad\qquad\qquad \mathbin{+\!\!+} (1 : [\,])
\end{aligned}$$

Functional Programming

$$\begin{aligned}
\mathbf{reverse}\ [\,] &= [\,]\\
\mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) &= \mathbf{reverse}\ \mathbf{xs} \mathbin{+\!\!+} (\mathbf{x} : [\,])
\end{aligned}$$

$$\begin{aligned}
&\mathbf{reverse}\ (1 : 2 : 3 : 4 : [\,])\\
&\quad = \mathbf{reverse}\ (2 : 3 : 4 : [\,]) \mathbin{+\!\!+} (1 : [\,])\\
&\quad = \mathbf{reverse}\ (3 : 4 : [\,]) \mathbin{+\!\!+} (2 : [\,]) \mathbin{+\!\!+} (1 : [\,])\\
&\quad = \mathbf{reverse}\ (4 : [\,]) \mathbin{+\!\!+} (3 : [\,]) \mathbin{+\!\!+} (2 : [\,]) \mathbin{+\!\!+} (1 : [\,])\\
&\quad = \mathbf{reverse}\ [\,] \mathbin{+\!\!+} (4 : [\,]) \mathbin{+\!\!+} (3 : [\,]) \mathbin{+\!\!+} (2 : [\,]) \mathbin{+\!\!+} (1 : [\,])\\
&\quad = 4 : 3 : 2 : 1 : [\,]
\end{aligned}$$

Functional Programming

- **program is set of equations between expressions**
- **recursion instead of built-in control structures**
- **values of variables do not change**
- **results independent of evaluation order**
- **constructors create, patterns eliminate data**

Functional Logic Programming

Language: Curry

```
last list
    | list ≡ xs ++ [x]    -- guard for the equation defining ‘last‘
   = x
  where x, xs free        -- free variables for unknown values
```

Functional Logic Programming

Language: Curry

```
last list
   | list ≡ xs ++ [x]    -- guard for the equation defining ‘last‘
  = x
  where x, xs free       -- free variables for unknown values
```

if **list** $= [1, 2, 3, 4]$ guard is satisfied for **xs** $= [1, 2, 3]$ and **x** $= 4$:

$$[1, 2, 3, 4] \equiv [1, 2, 3] +\!\!+ [4]$$

$\leadsto$ **last** $[1, 2, 3, 4] = 4$

Functional Logic Programming

```
insert x xs | xs ≡ ys ++ zs
            = ys ++ [x] ++ zs
  where ys, zs free
```

Functional Logic Programming

$$\begin{aligned}&\textbf{insert x xs} \;\mid\; \textbf{xs} \equiv \textbf{ys} +\!\!+ \textbf{zs}\\ &\qquad\qquad = \textbf{ys} +\!\!+ [\textbf{x}] +\!\!+ \textbf{zs}\end{aligned}$$

where **ys**, **zs** free

$$\begin{aligned}&\textbf{insert}\ 1\ [2,3,4]\\ &\quad = \textbf{insert}\ 1\ ([2] +\!\!+ [3,4])\\ &\quad = [2] +\!\!+ [1] +\!\!+ [3,4]\\ &\quad = [2,1,3,4]\end{aligned}$$

Functional Logic Programming

$$\begin{aligned}
&\mathbf{insert}\ \mathbf{x}\ \mathbf{xs}\ \mid\ \mathbf{xs} \equiv \mathbf{ys} \mathbin{+\!\!+} \mathbf{zs} \\
&\qquad\qquad = \mathbf{ys} \mathbin{+\!\!+} [\mathbf{x}] \mathbin{+\!\!+} \mathbf{zs} \\
&\quad \text{where } \mathbf{ys}, \mathbf{zs} \text{ free}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{insert}\ 1\ [2, 3, 4] \\
&\quad = \mathbf{insert}\ 1\ ([2] \mathbin{+\!\!+} [3, 4]) \\
&\quad = [2] \mathbin{+\!\!+} [1] \mathbin{+\!\!+} [3, 4] \\
&\quad = [2, 1, 3, 4]
\end{aligned}$$

$$\begin{aligned}
&\mathbf{insert}\ 1\ [2, 3, 4] \\
&\quad = \mathbf{insert}\ 1\ ([2, 3] \mathbin{+\!\!+} [4]) \\
&\quad = [2, 3] \mathbin{+\!\!+} [1] \mathbin{+\!\!+} [4] \\
&\quad = [2, 3, 1, 4]
\end{aligned}$$

Functional Logic Programming

- **free variables for unknown values**
- **built-in search for solving equational guards**
- **nondeterministic results**

Narrowing

```
cyi> reverse xs where xs free
```

Narrowing

```
cyi> reverse xs where xs free

{xs = []} []
More solutions? [Y(es)/n(o)/a(ll)] yes
```

Narrowing

```
cyi> reverse xs where xs free

{xs = []} []
More solutions? [Y(es)/n(o)/a(ll)] yes

{xs = [_a]} [_a]
More solutions? [Y(es)/n(o)/a(ll)] yes
```

Narrowing

```
cyi> reverse xs where xs free

{xs = []} []
More solutions? [Y(es)/n(o)/a(ll)] yes

{xs = [_a]} [_a]
More solutions? [Y(es)/n(o)/a(ll)] yes

{xs = [_b,_c]} [_c,_b]
More solutions? [Y(es)/n(o)/a(ll)] yes
```

Narrowing

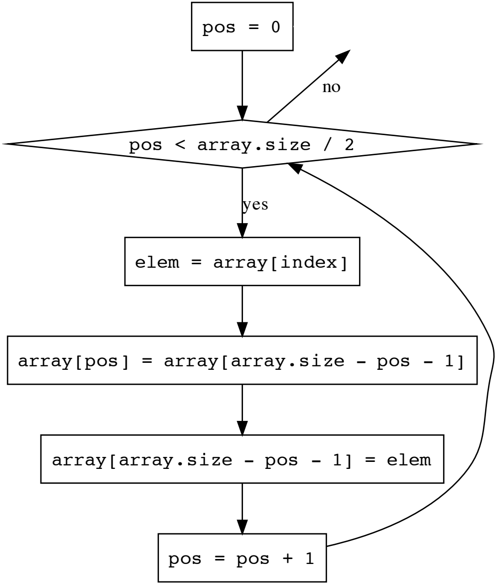
```
cyi> reverse xs where xs free

{xs = []} []
More solutions? [Y(es)/n(o)/a(ll)] yes

{xs = [_a]} [_a]
More solutions? [Y(es)/n(o)/a(ll)] yes

{xs = [_b,_c]} [_c,_b]
More solutions? [Y(es)/n(o)/a(ll)] yes

{xs = [_d,_e,_f]} [_f,_e,_d]
More solutions? [Y(es)/n(o)/a(ll)] no
```

Narrowing

- **generates test cases for free** (F. and Kuchen, PPDP 2007)
- **often infinitely many**
- **which tests are *redundant*?**

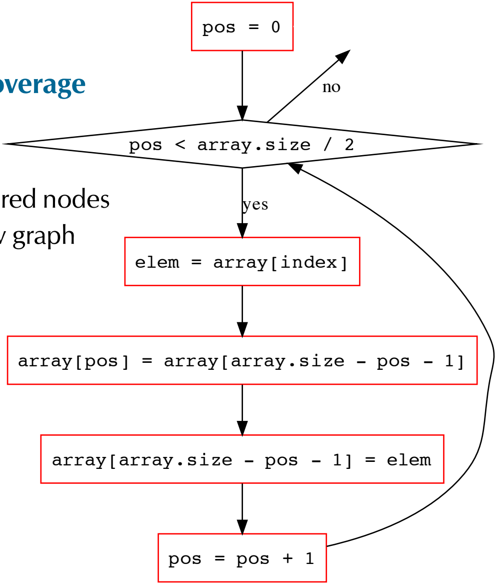
Code Coverage

- **groups tests into equivalence classes**
- **maps all possible program behaviours to finite set of classes**
- **approximation of program behaviour**
    - too fine: similar tests considered different
    - too coarse: interesting tests considered redundant
- **different notions, depending on paradigm**


pos = 0
no
pos < array.size / 2
yes
elem = array[index]
array[pos] = array[array.size - pos - 1]
array[array.size - pos - 1] = elem
pos = pos + 1

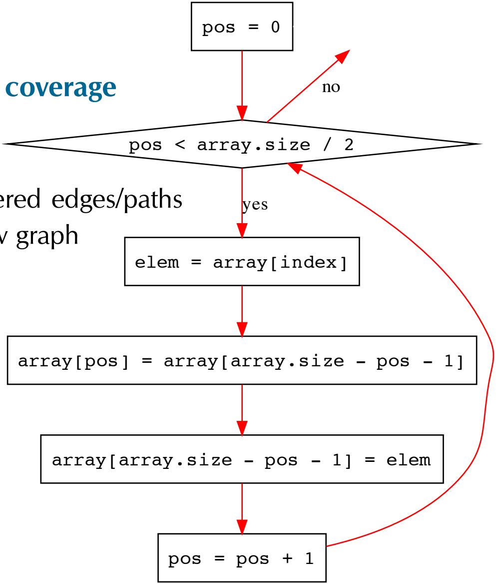
statement coverage

monitors covered nodes
in control-flow graph



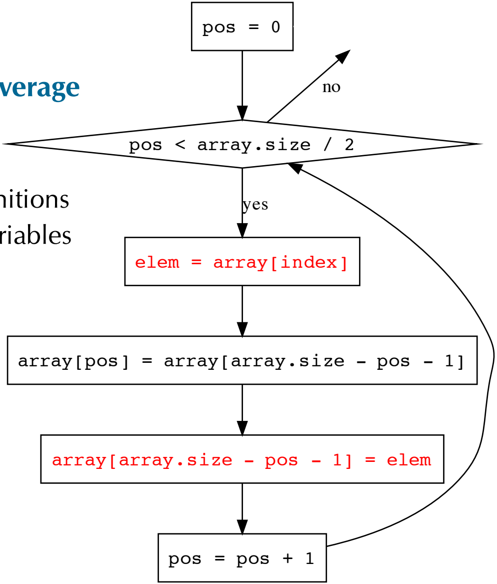
control-flow coverage

monitors covered edges/paths
in control-flow graph



data-flow coverage

monitors definitions
and uses of variables



Declarative Code Coverage

criteria from imperative programming cannot be transferred easily

- no sequence of statements
- no modifiable variables

new notions of code coverage for declarative programs

- **expression coverage**
- **control flow** (F. and Kuchen, PPDP 2007)
- **data flow** (F. and Kuchen, ICFP 2008)

Expression Coverage

which expressions need to be evaluated?

$$\mathbf{test}\ \mathbf{xs}\ \mathbf{ys} = \mathbf{reverse}\ (\mathbf{xs} +\!\!+ \mathbf{ys}) \equiv \mathbf{reverse}\ \mathbf{ys} +\!\!+ \mathbf{reverse}\ \mathbf{xs}$$

$$\begin{aligned}
\mathbf{reverse}\ [\,] &= [\,] \\
\mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) &= [\mathbf{x}] +\!\!+ \mathbf{reverse}\ \mathbf{xs} \\
[\,] +\!\!+ \mathbf{ys} &= \mathbf{ys} \\
(\mathbf{x} : \mathbf{xs}) +\!\!+ \mathbf{ys} &= \mathbf{x} : (\mathbf{xs} +\!\!+ \mathbf{ys})
\end{aligned}$$

test $[1]\ [\,]$ demands evaluation of all expressions in program

Expression Coverage

which expressions need to be evaluated?

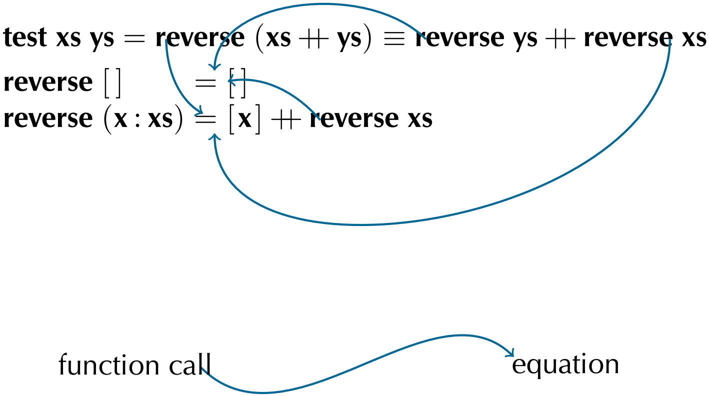
$$\mathbf{test}\ \mathbf{xs}\ \mathbf{ys} = \mathbf{reverse}\ (\mathbf{xs} \mathbin{+\!\!+} \mathbf{ys}) \equiv \mathbf{reverse}\ \mathbf{ys} \mathbin{+\!\!+} \mathbf{reverse}\ \mathbf{xs}$$

$$\begin{aligned}
&\mathbf{reverse}\ [\,] && = [\,] \\
&\mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) && = [\mathbf{x}] \mathbin{+\!\!+} \mathbf{reverse}\ \mathbf{xs} \\
&[\,] \mathbin{+\!\!+} \mathbf{ys} && = \mathbf{ys} \\
&(\mathbf{x} : \mathbf{xs}) \mathbin{+\!\!+} \mathbf{ys} && = \mathbf{x} : (\mathbf{xs} \mathbin{+\!\!+} \mathbf{ys})
\end{aligned}$$

test $[1]$ $[2]$ would have revealed a bug ⇝ criterion too coarse

Control-Flow Coverage

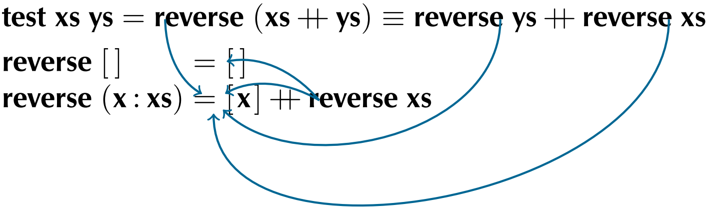
used equations of (buggy) **reverse** for evaluation of **test** $[1]\ [\,]$:



$$\begin{aligned}
&\mathbf{test\ xs\ ys} = \mathbf{reverse}\ (\mathbf{xs} +\!\!+\ \mathbf{ys}) \equiv \mathbf{reverse\ ys} +\!\!+\ \mathbf{reverse\ xs}\\
&\mathbf{reverse}\ [\,] \qquad\quad = [\,]\\
&\mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) = [\mathbf{x}] +\!\!+\ \mathbf{reverse\ xs}
\end{aligned}$$

Control-Flow Coverage

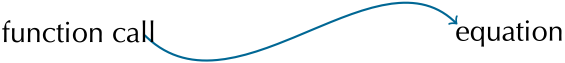
used equations of (buggy) **reverse** for evaluation of **test** $[1]\ [2]$:



$$\mathbf{test}\ \mathbf{xs}\ \mathbf{ys} = \mathbf{reverse}\ (\mathbf{xs} \mathbin{+\!\!+} \mathbf{ys}) \equiv \mathbf{reverse}\ \mathbf{ys} \mathbin{+\!\!+} \mathbf{reverse}\ \mathbf{xs}$$

$$\begin{aligned} \mathbf{reverse}\ [\,] &= [\,] \\ \mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) &= [\mathbf{x}] \mathbin{+\!\!+} \mathbf{reverse}\ \mathbf{xs} \end{aligned}$$

different arrows



Control-Flow Coverage

full control-flow coverage for (buggy) **reverse**:

$$\textbf{test}\ [\,]\ \ [\,]\ \ = \textbf{True}$$
$$\textbf{test}\ [1]\ [2] = \textbf{False}$$

Control-Flow Coverage

full control-flow coverage for (buggy) **reverse**:

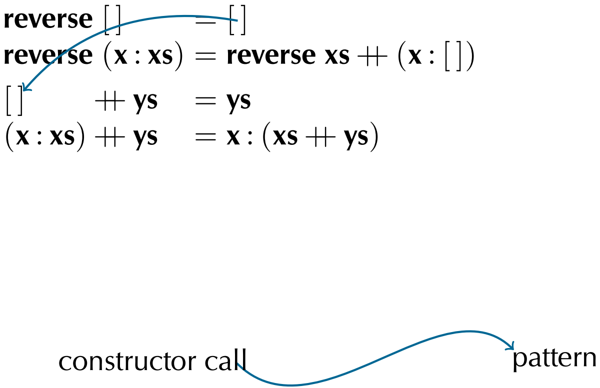
test $[\,]$ $[\,]$ $=$ **True**
test $[1]$ $[2]$ $=$ **False**

test $[\,]$ $[\,]$ $=$ **True**
test $[1]$ $[\,]$ $=$ **True**
test $[\,]$ $[1, 2]$ $=$ **True**

errors may remain undetected despite full coverage!

Data-Flow Coverage

data flow for (correct) evaluation of **reverse** $[1]$:

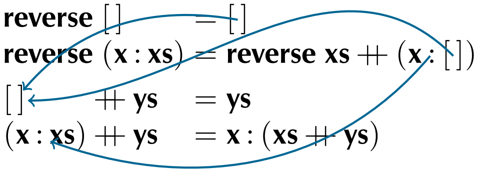


$$\begin{aligned}
&\mathbf{reverse}\ [\,] && = [\,]\\
&\mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) && = \mathbf{reverse}\ \mathbf{xs} +\!\!+ (\mathbf{x} : [\,])\\
&[\,] \qquad\ +\!\!+\ \mathbf{ys} && = \mathbf{ys}\\
&(\mathbf{x} : \mathbf{xs}) +\!\!+ \mathbf{ys} && = \mathbf{x} : (\mathbf{xs} +\!\!+ \mathbf{ys})
\end{aligned}$$

constructor call → pattern

Data-Flow Coverage

data flow for (correct) evaluation of **reverse** $[1, 2]$:

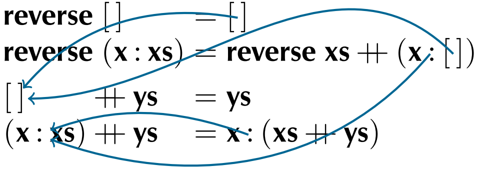


$$\begin{aligned}
\mathbf{reverse}\ [\,] &= [\,] \\
\mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) &= \mathbf{reverse}\ \mathbf{xs} \mathbin{+\!\!+} (\mathbf{x} : [\,]) \\
[\,] \mathbin{+\!\!+} \mathbf{ys} &= \mathbf{ys} \\
(\mathbf{x} : \mathbf{xs}) \mathbin{+\!\!+} \mathbf{ys} &= \mathbf{x} : (\mathbf{xs} \mathbin{+\!\!+} \mathbf{ys})
\end{aligned}$$

constructor call → pattern

Data-Flow Coverage

data flow for (correct) evaluation of **reverse** $[1,2,3]$:

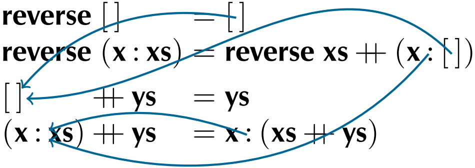


$$\begin{aligned}
&\mathbf{reverse}\ [\,] &&= [\,] \\
&\mathbf{reverse}\ (\mathbf{x} : \mathbf{xs}) &&= \mathbf{reverse}\ \mathbf{xs} +\!\!+ (\mathbf{x} : [\,]) \\
&[\,] +\!\!+ \mathbf{ys} &&= \mathbf{ys} \\
&(\mathbf{x} : \mathbf{xs}) +\!\!+ \mathbf{ys} &&= \mathbf{x} : (\mathbf{xs} +\!\!+ \mathbf{ys})
\end{aligned}$$

constructor call → pattern

Data-Flow Coverage

data flow for (correct) evaluation of **reverse** $[1, 2, 3, 4]$:



```
reverse []       = []
reverse (x : xs) = reverse xs ++ (x : [])

[]       ++ ys   = ys
(x : xs) ++ ys   = x : (xs ++ ys)
```

no new data flow for length 4

constructor call → pattern

Experiments

- introduce errors in algorithmically challenging programs
    - AVL tree (insert and delete functions)
    - Heapsort with purely functional heap
    - Strassen's algorithm for matrix multiplication
    - Dijkstra's shortest path algorithm
    - Kruskal's minimum spanning tree algorithm
    - Matrix Chain Multiplication (dynamic programming)
- generate tests automatically
- eliminate redundant tests regarding different coverage criteria
- check whether failing tests remain

Experiments

- introduce errors in algorithmically challenging programs
  - AVL tree (insert and delete functions)
  - Heapsort with purely functional heap
  - Strassen's algorithm for matrix multiplication
  - Dijkstra's shortest path algorithm
  - Kruskal's minimum spanning tree algorithm ← **textbook error**
  - Matrix Chain Multiplication (dynamic programming)
- generate tests automatically
- eliminate redundant tests regarding different coverage criteria
- check whether failing tests remain

Experiments

	Control Flow	Data Flow	Control + Data Flow
AVL insert	**9**	**12**	**20**
AVL delete	**9**	**23**	**27**
Heapsort	**6**	**7**	**16**
Strassen	**4**	**13**	**18**
Dijkstra	**6**	**9**	**10**
Kruskal	**4**	**9**	**10**
Matrix CM	**2**	**7**	**4**

Table: Sizes of reduced sets of tests for different coverage criteria

Declarative Code Coverage

used to classify tests for algorithmically challenging program units

control-flow coverage

- which equations are used by function calls?
- distinguishes different calls that use an equation

data-flow coverage

- which constructors are matched in patterns?
- distinguishes different values matched in an equation

combination is more thorough than one criterion alone

Thesis Contents

- **Declarative Programming**
  - **Functional programming**
  - **Functional logic programming**
- Generating Tests
  - Black-box testing
  - Glass-box testing
- **Code coverage**
  - **Control flow**
  - **Data flow**
  - Monitoring code coverage
  - **Experimental evaluation**
- Explicit nondeterminism
  - Nondeterminism monads
  - Combining laziness with nondeterminism

F. and Kuchen PPDP 2007. **Systematic generation of glass-box test cases for functional logic programs**
Proceedings of the 9th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming. ACM Press.

Christiansen and F. FLOPS 2008. **EasyCheck – test data for free**
Proceedings of the 9th International Symposium on Functional and Logic Programming. Springer LNCS 4989.

F. and Kuchen ICFP 2008. **Data-flow testing of declarative programs**
Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ACM Press.

F. ATPS 2009. **Reinventing Haskell backtracking**
Proceedings der GI-Jahrestagung Informatik 2009. GI LNI.

F., Kiselyov, and Shan ICFP 2009.
Purely functional lazy non-deterministic programming
Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. ACM Press.