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!
// 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
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!
// 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;
}
// 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;
}
pos = 0
array = [1, 2, 3, 4]
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 ]

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

← stop here!
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 ]

→ stop here!
**Imperative Programming**

\[
\begin{align*}
pos & = 0 \\
array & = [ 1 , 2 , 3 , 4 ] \\
pos & = 1 \\
array & = [ 4 , 2 , 3 , 1 ] \\
pos & = 2 \\
array & = [ 4 , 3 , 2 , 1 ] \leftarrow \text{stop here!} \\
pos & = 3 \\
array & = [ 4 , 2 , 3 , 1 ] \\
pos & = 4 \\
array & = [ 1 , 2 , 3 , 4 ]
\end{align*}
\]
// 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
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

reverse [] = []
reverse (x : xs) = reverse xs ++ (x : [])
Functional Programming

Language: Haskell

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

[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
Functional Programming

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

reverse (1 : 2 : 3 : 4 : [])
  = reverse (2 : 3 : 4 : []) ++ (1 : [])
Functional Programming

\[
\begin{align*}
\text{reverse } [ & ] & = [ & ] \\
\text{reverse } (x : xs) & = \text{reverse } xs \mathbin{++} (x : [ & ]) \\
\text{reverse } (1 : 2 : 3 : 4 : [ & ]) & \\
& = \text{reverse } (2 : 3 : 4 : [ & ]) \mathbin{++} (1 : [ & ]) \\
& = \text{reverse } (3 : 4 : [ & ]) \mathbin{++} (2 : [ & ]) \mathbin{++} (1 : [ & ]) \\
& = \text{reverse } (4 : [ & ]) \mathbin{++} (3 : [ & ]) \mathbin{++} (2 : [ & ]) \mathbin{++} (1 : [ & ]) \\
& = \text{reverse } [ & ] \mathbin{++} (4 : [ & ]) \mathbin{++} (3 : [ & ]) \mathbin{++} (2 : [ & ]) \mathbin{++} (1 : [ & ]) \\
& = 4 : 3 : 2 : 1 : [ & ]
\end{align*}
\]
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
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

\[ \text{list} \equiv \text{xs} + [x] \] -- guard for the equation defining 'last'
\[ = x \]
where \( x, \text{xs} \) free -- free variables for unknown values

if \( \text{list} = [1, 2, 3, 4] \) guard is satisfied for \( \text{xs} = [1, 2, 3] \) and \( x = 4 \):

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

\( \rightarrow \) last \( [1, 2, 3, 4] = 4 \)
```plaintext
insert x xs  |  xs ≡ ys ++ zs
              = ys ++ [x] ++ zs
where ys, zs free
```
Functional Logic Programming

\[ \text{insert } x \text{ xs } | \text{ xs } \equiv \text{ ys } \cons \text{ zs} \]
\[ = \text{ ys } \cons [x] \cons \text{ zs} \]

where \(\text{ys}, \text{zs}\) free

\[ \text{insert } 1 [2, 3, 4] \]
\[ = \text{ insert } 1 ([2] \cons [3, 4]) \]
\[ = [2, 1, 3, 4] \]
Functional Logic Programming

\[
\text{insert } x \text{ xs } | \quad \text{xs } \equiv \text{ ys } + + \text{ zs } \\
\quad = \text{ ys } + + [x] + + \text{ zs }
\]

where \( \text{ys}, \text{zs} \) free

\[
\text{insert } 1 \ [2, 3, 4] \\
\quad = \text{ insert } 1 \ ([2] + + [3, 4]) \\
\quad = [2] + + [1] + + [3, 4] \\
\quad = [2, 1, 3, 4]
\]

\[
\text{insert } 1 \ [2, 3, 4] \\
\quad = \text{ insert } 1 \ ([2, 3] + + [4]) \\
\quad = [2, 3] + + [1] + + [4] \\
\quad = [2, 3, 1, 4]
\]
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

\texttt{\texttt{cyi>\hspace{1em} reverse xs where xs free}}

\begin{verbatim}
{x \equiv [\_]} [\_]
\end{verbatim}

More solutions? [Y(es)/n(o)/a(ll)] yes

\begin{verbatim}
{x \equiv [\_a]} [\_a]
\end{verbatim}

More solutions? [Y(es)/n(o)/a(ll)] yes
Narrowing

\texttt{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

pos < array.size / 2

yes

elem = array[index]

array[pos] = array[array.size - pos - 1]

array[array.size - pos - 1] = elem

pos = pos + 1

no
Statement coverage

Monitors covered nodes in control-flow graph

$\text{pos} = 0$

$\text{pos} < \text{array.size} / 2$

Yes

$\text{elem} = \text{array}[\text{index}]$

$\text{array}[\text{pos}] = \text{array}[\text{array.size} - \text{pos} - 1]$

$\text{array}[\text{array.size} - \text{pos} - 1] = \text{elem}$

$\text{pos} = \text{pos} + 1$

No
control-flow coverage

monitors covered edges/paths in control-flow graph

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
data-flow coverage

monitors definitions and uses of variables

\[
pos = 0\]

\[
pos < \text{array.size} / 2\]

\[
\text{yes} \\
\text{elem = array[index]} \\
\text{array[pos] = array[array.size - pos - 1]} \\
\text{array[array.size - pos - 1] = elem} \\
pos = pos + 1 \]

\[
\text{no} \\
\text{pos = 0} \\
\text{pos < array.size / 2} \\
\text{elem = array[index]} \\
\text{array[pos] = array[array.size - pos - 1]} \\
\text{array[array.size - pos - 1] = elem} \\
pos = pos + 1 \]
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?

\[
\text{test } xs \ ys = \text{reverse } (xs \uplus ys) \equiv \text{reverse } ys \uplus \text{reverse } xs
\]

\[
\text{reverse } [] = []
\]
\[
\text{reverse } (x : xs) = [x] \uplus \text{reverse } xs
\]
\[
[] \uplus ys = ys
\]
\[
(x : xs) \uplus ys = x : (xs \uplus ys)
\]

\text{test } [1] [ ] \text{ demands evaluation of all expressions in program}
Expression Coverage

which expressions need to be evaluated?

\[
\text{test } xs \ ys = \text{reverse } (xs \uplus ys) \equiv \text{reverse } ys \uplus \text{reverse } xs
\]

\[
\begin{align*}
\text{reverse } [ ] &= [ ] \\
\text{reverse } (x : xs) &= [x] \uplus \text{reverse } xs \\
[ ] \uplus ys &= ys \\
(x : xs) \uplus ys &= x : (xs \uplus ys)
\end{align*}
\]

\textbf{test } [1] \ [2] \ would \ have \ revealed \ a \ bug \ \sim \ criterion \ too \ coarse
used equations of (buggy) `reverse` for evaluation of `test` \([1]\) \([\,]\):

\[
\text{test } xs \ ys = \text{reverse } (xs \ + \ ys) \equiv \text{reverse } ys \ + \ \text{reverse } xs
\]

\[
\text{reverse } [] = []
\]

\[
\text{reverse } (x : xs) = [x] + \text{reverse } xs
\]
Control-Flow Coverage

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

\[
\text{test } xs \ ys = \text{reverse } (xs ++ ys) \equiv \text{reverse } ys ++ \text{reverse } xs
\]

\[
\text{reverse } [ ] = [ ]
\]

\[
\text{reverse } (x : xs) \equiv [x] ++ \text{reverse } xs
\]

different arrows

function call equation
Control-Flow Coverage

full control-flow coverage for (buggy) reverse:

\[
\begin{align*}
\text{test } [ & ] [ & ] = \text{True} \\
\text{test } [1] [2] &= \text{False}
\end{align*}
\]

errors may remain undetected despite full coverage!
Control-Flow Coverage

full control-flow coverage for (buggy) reverse:

\[
\begin{align*}
\text{test } [ ] [ ] &= \text{True} \\
\text{test } [1] [2] &= \text{False} \\
\text{test } [ ] [1, 2] &= \text{True}
\end{align*}
\]

errors may remain undetected despite full coverage!
Data-Flow Coverage

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

\[
\begin{align*}
\text{reverse} & \quad = \quad [] \\
\text{reverse} \  (x : xs) & = \text{reverse} \  xs \  +\  (x : [] ) \\
[] + ys & = ys \\
(x : xs) + ys & = x : (xs + ys)
\end{align*}
\]
data flow for (correct) evaluation of reverse \([1, 2]\):

\[
\begin{align*}
\text{reverse } [] &= [] \\
\text{reverse } (x : xs) &= \text{reverse } xs \mathbin{+\!+} (x : []) \\
[] + ys &= ys \\
(x : xs) + ys &= x : (xs + ys)
\end{align*}
\]
Data-Flow Coverage

data flow for (correct) evaluation of \texttt{reverse} \[1, 2, 3\]:

\begin{align*}
\texttt{reverse} \; [ ] &= [ ] \\
\texttt{reverse} \; (x : xs) &= \texttt{reverse} \; xs + + (x : [ ] ) \\
[ ] + + ys &= ys \\
(x : xs) + + ys &= x : (xs + + ys)
\end{align*}
Data-Flow Coverage

data flow for (correct) evaluation of \texttt{reverse} \([1, 2, 3, 4]\):

\[
\begin{align*}
\text{reverse } [ & ] = [ ] \\
\text{reverse } (x : xs) = \text{reverse } xs ++ (x : []) \\
[] ++ ys = ys \\
(x : xs) ++ ys = x : (xs ++ ys)
\end{align*}
\]

no new data flow for length 4

constructor call \quad \text{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

<table>
<thead>
<tr>
<th></th>
<th>Control Flow</th>
<th>Data Flow</th>
<th>Control + Data Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVL insert</td>
<td>9</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>AVL delete</td>
<td>9</td>
<td>23</td>
<td>27</td>
</tr>
<tr>
<td>Heapsort</td>
<td>6</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>Strassen</td>
<td>4</td>
<td>13</td>
<td>18</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>6</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Kruskal</td>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Matrix CM</td>
<td>2</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

**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

Christiansen and F. FLOPS 2008. EasyCheck – test data for free

F. and Kuchen ICFP 2008. Data-flow testing of declarative programs

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