Dataflow Testing
of Declarative Programs

Herbert Kuchen
University of Münster
Germany

Sebastian Fischer
University of Kiel
Germany

ICFP 2008

Test Generation

- usually: generators based on input types
- **QuickCheck** (Claessen/Hughes, 2000)
  random testing
- **SmallCheck** (Runciman et al, 2008)
  exhaustive testing

Functional Logic Programming

```
> reverse l where l free
```

Functional Logic Programming

```
> reverse l where l free
[]
```

More?

Functional Logic Programming

```
> reverse l where l free
[]                          More? yes
[x2]                        More?
```

Functional Logic Programming

```
> reverse l where l free
[]                          More? yes
[x2]                        More? yes
[x6,x2]                     More?
```

Functional Logic Programming

```
> reverse l where l free
[]                          More? yes
[x2]                        More? yes
[x6,x2]                     More? yes
[x9,x6,x2]                  More? no
```

Functional Logic Programming

```
> (l,reverse l) where l free
```

Functional Logic Programming

```
> (l,reverse l) where l free
([]          ,[])          More?
```

Functional Logic Programming

```
> (l,reverse l) where l free
([]      ,[])      More? yes
([x2]    ,[x2])    More?
```

Functional Logic Programming

```
> (l,reverse l) where l free
([]          ,[])          More? yes
([x2]        ,[x2])        More? yes
([x2,x6]     ,[x6,x2])     More?
```

Functional Logic Programming

```
> (l,reverse l) where l free
([]             ,[])                More? yes
([x2]           ,[x2])              More? yes
([x2,x6]        ,[x6,x2])           More? yes
([x2,x6,x9],[x9,x6,x2])             More? no
```

Functional Logic Programming

- An FLP language is a test-case generator
  - it's demand driven,

    i.e., it generates what the program needs
  - Lazy SmallCheck simulates this

Bound on #Tests

- **QuickCheck**: fixed number of tests
- **SmallCheck**: fixed max size of input
- numbers justified only experimentally

Dynamic Bound

- Stop when there is no new behaviour
- i.e., when executions are similar
- What differentiates executions?

Code Coverage

- monitors execution details
- e.g., which expressions are evaluated
  cf. HPC (Gill/Runciman 2007)
- control flow (Kuchen/F. 2007)
- or dataflow!
- why new notions of coverage?

There's a Bug

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

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

test xs ys =
   reverse xs ++ reverse ys
     == reverse (ys++xs)

main = test [0] [] -- yields True
```

Imperative Data Flow

```
x = 1;                    // definition of x
y = 2;

if ( p() ) {
   z = x;                 // use of x
}
```

Declarative Data Flow

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ x:[]
```

Declarative Data Flow

Call	Result	Dataflow
`reverse []`	`[]`	–
`reverse [x]`	`[x]`	[]-rev1 → ++
`reverse [x,y]`	`[y,x]`	[]-rev1 → ++ []-rev2 → ++ (:)-rev → ++
`reverse [x,y,z]`	`[z,y,x]`	[]-rev1 → ++ []-rev2 → ++ (:)-rev → ++ (:)-app → ++

Declarative Data Flow

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ x:[]
```

Declarative Data Flow

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ x:[]
```

Declarative Data Flow

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ x:[]
```

Implementation

- program transformation
- coverage: additional result
- no side effects

  important due to nondeterminism

  (each test has its own coverage!)
- details in the paper

Test Generation

- instrument program to test
- execute on unknown input
- generate results as long as new coverage
- important: complete strategy

  iterative deepening, breadth first search

What else?

- compute minimal set of tests
- simplifies manual check for correctness
- dispensable with property-based testing (result type **`Bool`**)

Experiments

```
test :: [Int] -> [Int] -> Bool
test xs ys =
   reverse xs ++ reverse ys
     == reverse (ys++xs)
```

```
-- covering tests:
-- test []   []   = True
-- test [0] [1] = False
```

Experiments

- AVL, Heap(sort), Strassen, Tabled CMM, Dijkstra, Kruskal, Cheapest Train Ticket
  - most thorough testing with combination of all notions of coverage
  - details are in the paper

Outlook

- adaptive search
- correlate coverage with test failure

  (each test has its own coverage!)

Conclusions

- FLP: **demand driven** test generation
- coverage: **justified limit** for #tests
- simple notion of **dataflow coverage**

  constructors flow to patterns

  lambdas flow to applications (cf. paper)
- program transformation: pure
- more **thorough** testing