

QuickChecking Refactoring Tools

Dániel Horpácsi Dániel Drienyovszky

University of Kent

Eötvös Loránd University

February 17, 2010

Contact

- Dániel Horpácsi *(Generating random programs)*
daniel.horpacsi@gmail.com
daniel_h@inf.elte.hu
- Dániel Drienyovszky *(Testing correctness)*
drienyovszky@gmail.com
monogram@inf.elte.hu

Supervisor:

Simon Thompson

S.J.Thompson@kent.ac.uk

Main topics

- Introduction
- Generating random programs
 - Formalism
 - Example
 - Results
- Testing correctness
 - Implementation
 - Results

Main topics

- Introduction
- Generating random programs
 - Formalism
 - Example
 - Results
- Testing correctness
 - Implementation
 - Results

Erlang

Dynamically typed impure functional language with message passing

Refactoring

Behaviour-preserving transformation of source code, which increases code quality

Refactoring examples

Example: rename Foo to X

```
id(Foo) -> Foo.
```

```
id(X) -> X.
```

Refactoring examples

Example: generalize function

`inc(N) -> N + 1.`

`inc(N, X) -> N + X.`

Tool support

2 available refactoring tools for erlang

- Wrangler (University of Kent)
- RefactorErl (Eötvös Loránd University)

Testing the tools

- Refactoring tools are poorly tested
- Idea: use QuickCheck!
 - Generate random erlang programs
 - Test refactoring correctness

Testing the tools

- Refactoring tools are poorly tested
- Idea: use QuickCheck!
 - Generate random erlang programs
 - Test refactoring correctness

Testing the tools

- Refactoring tools are poorly tested
- Idea: use QuickCheck!
 - Generate random erlang programs
 - Test refactoring correctness

Testing the tools

- Refactoring tools are poorly tested
- Idea: use QuickCheck!
 - Generate random erlang programs
 - Test refactoring correctness

Main topics

- Introduction
- Generating random programs
 - Formalism
 - Example
 - Results
- Testing correctness
 - Implementation
 - Results

Main topics

- Introduction
- **Generating random programs**
 - Formalism
 - Example
 - Results
- Testing correctness
 - Implementation
 - Results

QuickCheck

- Method and framework for property based testing
- Originally for Haskell
- Since 2006 implemented for Erlang as well

Provides library, containing functions and macros

- Generators for creating random test data
- Formalism for describing properties
- Push-button testing functionality against the user-written specification

QuickCheck generators

- First-order (*int/0, choose/2, elements/1*)
- Higher-order (*list/1, oneof/1, bind/2*)

Example

```
oneof(lists:seq(1, 20))  
?SUCHTHAT(I, int(), I >= 1 andalso I =< 20)  
choose(1, 20)
```

Very first attempt

- Erlang Syntax Tools helps to build the AST
- Describing Erlang only with generators is difficult

In the case of very large data, writing generators

- Results in unduly large Erlang source code
- Hard to understand and maintain

A metalanguage for the generators would be helpful

First attempt: eqc_grammar

- Included in the latest Quviq QuickCheck release
- Designed for describing protocol grammars
- Based on yecc notation
- Ideal for simple situations

For us

- Not expressive enough, cannot describe static semantics
- Wherein lies the problem?

Attribute grammars

Attribute grammars are extensions of context-free grammars

- Attributes attached to grammar symbols
 - Synthesised
 - Inherited
- Attribute computation rules for the grammar rules

Two important subclasses

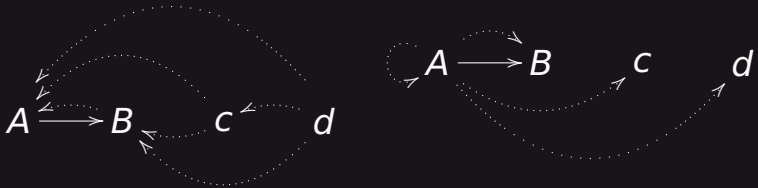
- S-Attributed
- L-Attributed

S-Attributed grammars

- Involve synthesised attributes only
- eqc_grammar notation
- Without inheritance cannot express/manage
 - Bound variables in the actual context
 - Defined functions in a module
 - Essential type information
 - Etc.

L-Attributed grammars

- Allow inheritance with restriction
- No dependencies from a child
 - To the child itself
 - To the child's right



- For the generation these restrictions don't really matter
- Perfect for describing the Erlang grammar

Formalising L-Attributed grammars

- eqc_grammar is a good starting point
- Needed extensions
 - More than one synthesised attributes
 - Inherited attributes
 - Conditions
- Additional extensions
 - Formalising various QuickCheck generators

Formalising rules and attributes

- eqc_grammar rule notation

A -> B C

: exprs calculating synthesised value .

- Extended rule notation

A -> B [@inattr1 = exprs1, @inattr2 = exprs2]

 C [@inattr1 = exprs3, @inattr2 = exprs4]

:: exprs calculating synthesised attr 'value'

[@synattr1 = exprs5, @synattr2 = exprs6] .

Formalising rules and attributes

- eqc_grammar rule notation

```
A -> B C
      : exprs calculating synthesised value .
```

- Extended rule notation

```
A -> B [@inattr1 = exprs1, @inattr2 = exprs2]
      C [@inattr1 = exprs3, @inattr2 = exprs4]
      :: exprs calculating synthesised attr 'value'
      [@synattr1 = exprs5, @synattr2 = exprs6] .
```

Formalising rules and attributes

For instance...

```
clause -> patterns [@count = '$0'.arity]
      ~> guard
          exprs [@type = '$0'.type,
                @vars = '$1'.vars]
      :: erl_syntax:clause('$1','$2','$3')
      [@difficulty = some_metrics('$3')].
```

Formalising QuickCheck generators

- Our notation is a high-level formalism of generators
- `eqc_grammar` doesn't formalise repetition, alternatives (in spite of the fact that QuickCheck supports it)
- Why not represent the most useful built-in generators?
 - List (Vector)
 - Oneof (Frequency)
 - Sized, Lazy
- The notation becomes closer to EBNF rather than to BNF

Formalising lists

Note the difference between

- Independent lists (elements are generated simultaneously, inheriting the parent's attributes)

`{symbol}` `{size, symbol}`

- Dependent lists (elements are generated one after the other, inheriting attributes from the previous one)

`{~ symbol}` `{~ size, symbol}`

Formalising alternatives

- Simple alternatives (oneof)

a -> b c
 | d e .

- Weighted alternatives (frequency)

a -> (*3) b c
 | (*5) d e .

Formalising guards

- Rule alternative guard

```
a -> b c
    | (when guard_foo) d e .
```

- The guard itself (defined by the user), based on the current attribute values

```
guard_foo(Attributes) ->
  ?get_attr(bar, Attributes) != [].
```

Main topics

- Introduction
- **Generating random programs**
 - Formalism
 - Example**
 - Results
- Testing correctness
 - Implementation
 - Results

A simple example ($a^n b^n c^n$)

```
abc_seq -> a_seq
         ~> b_seq [@size = '$1'.size]
           c_seq [@size = '$1'.size].
```

```
a_seq -> a      :: [@size = 1]
       | a_seq a :: [@size = '$1'.size + 1].
```

```
b_seq -> (when size_is_1) b
       | b_seq [@size = '$0'.size - 1] b.
```

```
c_seq -> {'$0'.size, c}.
```

A simple example ($a^n b^n c^n$)

```
abc_seq -> a_seq  
         ~> b_seq [@size = '$1'.size]  
           c_seq [@size = '$1'.size].
```

```
a_seq -> {a} :: [@size = length('$1')].
```

```
b_seq -> {'$0'.size, b}.
```

```
c_seq -> {'$0'.size, c}.
```

Main topics

- Introduction
- **Generating random programs**
 - Formalism
 - Example
 - Results**
- Testing correctness
 - Implementation
 - Results

Results

- High-level QuickCheck formalism for L-AG designed
- About 5 times more compact than QC generators
- Easier to understand and maintain
- Compiler created (*rules.yrl* → *generators.erl*)
- A large part of the Erlang language is already formalised

Main topics

- Introduction
- Generating random programs
 - Formalism
 - Example
 - Results
- Testing correctness
 - Implementation
 - Results

Main topics

- Introduction
- Generating random programs
 - Formalism
 - Example
 - Results
- Testing correctness
 - Implementation
 - Results

Behavioural equivalency

If a refactoring was performed correctly, the program should return the same output for the same input, throw the same exceptions, and send the same messages in the same order.

Overall structure

- Decide on some module names
- Generate arguments for the refactoring
- Perform the refactoring
- Test equivalency

Overall structure

- Decide on some module names
- Generate arguments for the refactoring
- Perform the refactoring
- Test equivalency

Overall structure

- Decide on some module names
- Generate arguments for the refactoring
- Perform the refactoring
- Test equivalency

Overall structure

- Decide on some module names
- Generate arguments for the refactoring
- Perform the refactoring
- Test equivalency

Testing equivalency

Difficulties

- generating arguments
- exceptions
- programs that don't terminate
- message passing
- I/O

Testing equivalency

Difficulties

- generating arguments
- exceptions
- programs that don't terminate
- message passing
- I/O

Testing equivalency

Difficulties

- generating arguments
- exceptions
- programs that don't terminate
- message passing
- I/O

Testing equivalency

Difficulties

- generating arguments
- exceptions
- programs that don't terminate
- message passing
- I/O

Testing equivalency

Difficulties

- generating arguments
- exceptions
- programs that don't terminate
- message passing
- I/O

Generating arguments

We make use of dialyzer's type inference to generate random input for the program.

Exceptions

Wrap the function call in a try catch block

```
try_apply(Fun) ->  
  try Fun() of  
    Result -> {ok, Result}  
  catch  
    C:E     -> {error, C, E}  
end.
```

Non termination

Run the functions in a separate process that can be killed after a timeout

```
try_running(F) ->  
  Self = self(),  
  Pid = spawn(fun() ->  
    Self ! {self(), try_apply(F)}  
  end),  
  receive  
    {Pid, R} -> R  
  after ?TIMEOUT ->  
    exit(Pid, kill),  
    {error, timeout}  
end.
```

Message passing

We can only test a special form of message passing, which arises during I/O.

IO device

I/O works by sending messages to a process using a certain protocol. Any process that implements this protocol is an I/O device.

IO device

Our I/O device works by collecting logs of requests, and then comparing them either for equality or one being a prefix of the other.

Main topics

- Introduction
- Generating random programs
 - Formalism
 - Example
 - Results
- Testing correctness
 - Implementation
 - Results

Refactoring steps tested

- Rename Variable
- Rename Function
- Generalize Function
- Tuple Function Arguments

Refactoring steps tested

- Rename Variable
- Rename Function
- Generalize Function
- Tuple Function Arguments

Rename variable

```
foo(V, V) ->  
  V.
```

```
foo(V, V)  
  V.
```

Rename variable

foo(V, V) ->
V.

foo(X, V) ->
X.

Generalize function

```
foo(-3) ->  
3.
```

```
foo(3)
```

```
3.
```

Generalize function

`foo(-3) ->`
`3.`

`foo(-X) ->`
`X.`

Thank you!

Questions?