Towards trustworthy refactoring in Erlang

Dániel Horpácsi, Judit Kőszegi, Simon Thompson

„Could we design a refactoring formalism in which any definition is inherently correct?”

Refactoring language: design goals

- Executable
- Verifiable
- Applicable
- Intuitive
- Representation-independent

All at once

Execution

Formalism: high-level DSL

- Mostly declarative
- Partly Erlang-specific
- Concrete term rewriting with simple strategies

Interpreted in the existing refactoring framework

- Static analysis
- Transformation
- Pretty-printing

Verification

- Correctness of the refactoring definition: for any program, the refactoring transformation results in a *semantically equivalent* program
- Correctness of a refactoring application: the original and the resulting program are *semantically equivalent*

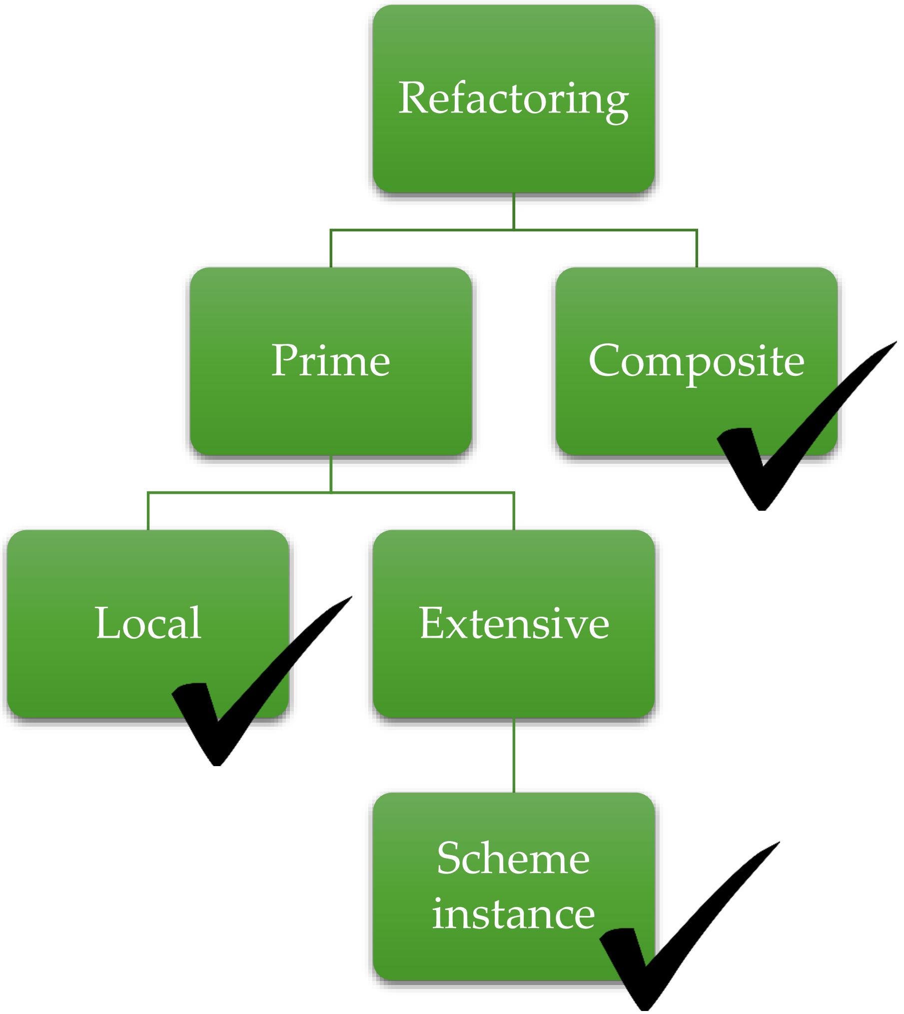
- Consequently, the refactoring correctness is relative to
  - The language semantics
  - The language metatheory

- (The existing framework is not subject to this verification)

Types of refactoring definitions

The smaller the better

- Local
  - a single conditional rewrite rule
- Extensive
  - combination of rules
- Composite
  - combination of refactorings



A local example

```
[ X*Y || X <- Numbers1,
         Y <- Numbers2,
         X > Y ]
```

```
List = [{X, Y} || X <- Numbers1, Y <- Numbers2, X>Y],
Fun  = fun({X, Y}) -> X*Y end,
lists:map(Fun, List)
```

```
REFACTORING listcomp2map()
    [ Head || GeneratorsFilters.. ]
   ----------------------------------------------
    List = [ { Vars.. } || GeneratorsFilters.. ],
    Fun  = fun ({ Vars.. }) -> Head end,
    lists:map(Fun, List)
WHEN
  Vars.. = intersect(bound_vars(GeneratorsFilters..), vars(Head)))
  AND fresh(List)
  AND fresh(Fun)
```

Proving correctness

- Correctness of refactoring
- Equivalence of program patterns
- Validity in reachability logic

Proving equivalence

- Operational semantics (+metatheory) defined in reachability logic
  - Special sort: configuration
  - Special predicate: basic pattern
  - Pairs of pure patterns
- Equivalence property expressed in reachability logic
  - Pairs of pure patterns with configuration pairs
- Symbolic circular coinduction to derive formula validity
  - Sound but not complete
  - Tactic and implementation for automatic proofs

An extensive example

```
REFACTORING rename_function(NewName)
ON function_definition(THIS)
  Name(Args..) -> Body..
  -------------------------
  NewName(Args..) -> Body..
WHEN NOT function_exists(module(THIS), NewName, length(Args..))
THEN ON function_calls(THIS)
  Name(Args..)
  ---------------
  NewName(Args..)
THEN ON ...
THEN ON ...
```

Schemes

- **Guarantee consistent changes**
- Hide the complexity of extensive refactorings
- Simplify definition and verification by splitting into two parts
- Contract on the parameters ensures correctness

Function signature refactoring

Skeleton	Applying the signature rewrite (name + args) on the function definition and every function reference (calls, directives, etc.)
Parameter	A „function head" rewrite rule specifying how the signature is changed
Contract	Formal and generality requirements on the arguments

Examples

```
FUNCTION SIGNATURE REFACTORING rename_function(NewName)
    Name(Args..)
    ---------------
    NewName(Args..)

FUNCTION SIGNATURE REFACTORING tuple_function_arguments()
    Name(Args..)
    --------------
    Name({Args..})
```

Forward dataflow refactoring

Skeleton	Applying any of the „definition" rules on the selected data source and applying any of the „reference" rules on each element of the dataflow path
Parameters	At least one „definition" transformation rule and at least one „reference" trasformation rule
Contract	Each pair of „definition" and „reference" rules are consistent

Example

FORWARD DATAFLOW REFACTORING fun2value()

DEFINITION

```
     fun() -> E end
    ---------------- WHEN pure(E)
           E
```

REFERENCE F

```
     F()
    -----
     F
```

```
X = fun() -> 123 end,
some_code(),
Y = X() + 1,
X() - 5
```

⇒

```
X = 123,
some_code(),
Y = X + 1,
X - 5
```

Correctness of scheme instances

Towards trustworthy refactoring

- Simple, executable formalism for defining refactorings
- Local, extensive and composite definitions
- High-level refactoring schemes for extensive transformations
- A method for turning any refactoring definition into a formally verifiable logic formula