Refactoring in Erlang, a Dynamic Functional Language

László Lövei, Zoltán Horváth, Tamás Kozsik, Roland Király, Anikó Víg, and Tamás Nagy

Eötvös Loránd University, Budapest, Hungary

Introduction

- Erlang: programming language for telecom SW
  - Functional language with possible side effects
  - Built-in concurrency with message passing
  - Dynamically typed: no compile-time type checking
- A refactoring catalog for Erlang is being built
  - Object oriented refactorings not applicable
  - Haskell-like data type transformations not feasible
- Final goal is tool support for them
  - Find out if Erlang itself is a good platform for refactoring
  - Trying to find methods for proven refactoring

Implementation ideas

- Source code is stored in a semantical graph
- The syntax tree is extended by attributes and edges representing semantical information
  - Function calls linked with the function definition
  - Variable references linked with the binding occurrence
  - Subexpressions are linked with their contexts
- Semantical links are calculated right after parsing
- Condition checks and transformations don't need traversals, only fixed length paths
- Persistent graphs can be utilized to improve efficiency in case of a large codebase

Prototype experiences

- 7 different refactorings are working
- User interface is provided through GNU Emacs
  - Handles selections and other user input
- Analysis and refactoring logic is written in Erlang
- Graph representation in a relational database
  - Graph manipulations are expressed in SQL
  - Fixed-length paths are described by joining tables
- SQL database didn't work out well
  - Inefficient connection with Erlang
  - Promising experiments with Erlang-specific databases (Mnesia)

Function reference tracking

```
-module(ev).
-export([event/2]).
event(Srv, Ev) -> Srv ! {event, Ev}.
stop(F, Srv) -> F(Srv, stop).
calls(Srv) -> event(Srv, event),
   apply(ev, event, [Srv, event]),
   stop(event, Srv).
```

- Functions are identified by data tags called *atoms*
- Function calls can use runtime-generated function names
- Dynamic constructs can be handled by different tactics
  - Type inference to find function names
  - Data flow analysis to find call places
  - Runtime conversions

```
calls(Srv) ->
   apply(ev, event,
     (fun([P1,P2])->[P2,P1] end)
       ([Srv, event]))
```

Properties of expressions

- Variables are bound a value only once (although we don't know the type of that value)
- The binding structure defines which variables are used or bound in an expression (*extract function*)
- Most language constructs are side effect-free, which enables rearranging expressions (*eliminate variable*, *merge expression duplicates*)
  - Message passing and BIFs introduce side effects
  - Functions that use them or call “dirty” functions are “dirty” too

Refactoring data structures

- Trivial function call transformations: *reorder function arguments* and *tuple function arguments*
- A complex refactoring: *introduce record*, which replaces tuples with records of given fields

```
server({Data, Info}) ->
  receive Req ->
    server({Data,
      handle(Req, Info)})
  end.
```

```
-record(state, {data, info}).
server(St=#state{}) ->
  receive  Req ->
    server(St#state{
      info=handle(Req,Info)})
  end.
```

- Tuple instances that are computed from the starting tuple need to be found
- Data flow of tuples and fields should be followed
- Ongoing work

Other information

- Cooperation with Simon Thompson, University of Kent
- Project homepage: `http://plc.inf.elte.hu/erlang/`

Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK, Ericsson Hungary, ELTE CNL, and OMAA-ÖAU 66öu2