Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 137–145.

Static rules of variable scoping in Erlang*

László Lövei, Zoltán Horváth, Tamás Kozsik, Roland Király, Róbert Kitlei

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
e-mail: {lovei,hz,kto,kiralyroland,kitlei}@inf.elte.hu

Abstract

Erlang/OTP is a functional programming environment designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics. The dynamic nature of this environment, which partly comes from concurrency and partly from dynamic language features, offers a great challange for a refactoring tool. Refactoring is a programming technique for improving the design of a program without changing its behaviour. Many refactorings are concerned with variables in some way. This paper presents variable scoping rules for Erlang that are more suitable for describing refactoring conditions and transformations than those given in the Erlang reference manual.

1. Introduction

The phrase "refactoring" stands for program transformations that preserve the meaning of programs [5]. Such transformations are often applied in order to improve the quality of program code: make it more readable, satisfy coding conventions, prepare it for further development etc. Refactoring may precede a program modification or extension, it may be used after finishing the work in order to bring the program into a nicer shape, but it can be used for optimisation purposes as well. Simple refactorings are used by developers almost every day; they rename variables, introduce new arguments to functions, or create new functions from duplicated code.

In order to examine the feasibility of refactorings and to perform the transformations on a program, it is essential to analyse the structure of functions and expressions and the use of variables in that program. This paper proposes a way to analyse variable bindings, and introduces rules for variable scoping and visibility.

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

The presented rules are more appropriate for implementing refactorings than those given in the Erlang reference manual [3].

1.1. Erlang

Erlang/OTP is a functional programming environment developed by Ericsson. It was designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics (like telecommunication systems). The core Erlang language consists of simple functional constructs extended with message passing to handle concurrency. The language has a very strong dynamic nature that partly comes from concurrency and partly from dynamic language features. Another characteristics of the language that makes refactoring harder is that Erlang processes cannot utilize shared memory, so message passing is used to exchange data between processes.

OTP is a set of design principles and libraries that supports building fault-tolerant systems [2]. It contains the *gen-server*, *gen-fsm*, *gen-event*, *supervisor* etc. modules, which formalize a common pattern for client/server applications. These modules (also called *behaviours*) are combined with *callback modules* – ones that contain callback functions invoked from the behaviours. This complex structure of modules hides many details of the inter-process communication, thus facilitating the static analysis of Erlang/OTP applications.

1.2. Refactoring in Erlang

From the point of view of refactoring, the most important characteristic of a programming language is the extent of semantical information available by static analysis. Erlang is a functional programming language, which is an advantage for refactoring. Side effects are restricted to message passing and built-in functions,[1] and variables are assigned a value only once in their lifetime. In Erlang the code is organised into modules with explicit interface definitions and static export and import lists.

An unusual property for a functional language is that variables are not typed statically: they can have a value of any data type. This and a few similar dynamic features offer a real challenge to static analysis. As an example, consider the matching of corresponding message send and receive instructions. A destination of a message can be a process identifier or a registered name, which are bound to function code at runtime. Data flow analysis might help in discovering these relations, but it is a hard research topic in itself.

Another kind of problem is the possibility of running dynamically created code. The most prominent example of this is the `eval` built-in function, which evaluates a string – which is constructed at runtime – as Erlang code. This functionality is clearly out of the scope of a static refactoring tool. However, there are similar other constructs that are widely used and are possible to cover at least partially,

[1] Built-in functions, or BIFs, are functions that are implemented in the runtime system.

like the `spawn` function that starts the evaluation of a function in a new process (and the function name and arguments might be constructed at runtime), or the `apply` function that calls a function (with the same runtime-related problems). The normal function call syntax has some runtime features too: variables are allowed instead of static module or function names.

Due to the afore-mentioned difficulties, if a programmer wants to carry out semantics preserving transformations, (s)he must be aware of the limitations of the applied refactoring tool and those of the static analyses performed by the tool.

1.3. Variables in Erlang

Many refactorings are concerned with variables in some way. Expressions in Erlang programs can use and define variables almost anywhere in the code. The meaning of variables in an expression depends on its context. So the relation between the variables and the context of the expression must be maintained during a refactoring, otherwise the meaning of the modified program text would differ from that of the original.

The relation of variables and expressions is defined in terms of visibility rules. An expression can only use visible variables, and the visibility of variables begins with their creation, often in an expression. Every refactoring that works with variables or expressions must be able to determine the exact visibility region of every variable.

The exact rules defining variable visibility are given in [3], using input and output contexts for every language construct. These – operational semantics based – rules are hard to follow and they are not really helpful in defining the conditions of a refactoring. This paper proposes a more compact definition which is suitable for static analyses and hence for verifying refactoring conditions.

In Erlang, variables have a name[2] and a value bound to them (which never changes during the lifetime of the variable). According to the rules presented in this paper, the *scope* of a variable is a non-contiguous region of the program text where a value is bound to the variable, and a variable is *visible* in a region where its name can be used to refer to the variable.

2. Variable scoping and visibility

The scope of a variable is always limited to a function clause, a list comprehension or an element of a tuple or list. There are no global variables: every variable is *local* to some *scope delimiter*, namely to a function clause (of either a declared function or an explicit fun-expression), or to a list comprehension, or to an element of a tuple or list. The outermost scope delimiters are the clauses of declared functions: they are never nested in other scope delimiters, and every other scope delimiter is nested (directly or indirectly) in a clause of a declared function.

[2]Variable names always begin with a capital letter or an underscore, the latter meaning an ignored value.

```
sign_abs(X) ->
   if
      X >  0 -> Z =  X,  Y =  1;
      X <  0 -> Z = -X,  Y = -1;
      X == 0 -> Z =  0,  Y =  0
   end,
  {Y,Z}.
```

Figure 1: The definition of the `sign_abs/1` function

A variable occurrence is *direct* in a scope delimiter, if it is an occurrence in that scope delimiter, but not an occurrence in some enclosed scope delimiter. A variable occurs directly in a scope delimiter, if it has at least one direct occurrence in that scope delimiter.

The scope of a variable is a non-contiguous region of the outermost scope delimiter where the variable directly occurs. (There is exactly one such outermost scope delimiter. Given a scope delimiter with no direct occurrences of a certain variable name, but with two or more independent nested scope delimiters with direct occurrences of that variable name, the variable name refers to two or more independent variables.) The scope of a variable is given as a set of expressions.

Variables are created by the pattern matching mechanism. Pattern matching is used in

- heads of function clauses (of declared functions or of fun-expressions),
- pattern match expressions,
- `case`, `receive` and `try` constructs and
- generators of list comprehension expressions.

The same variable might be defined by multiple pattern matching expressions; for example, in different branches of a branching expression. Therefore the scope of such a variable begins at multiple places of the program text: this is why the scope is a non-contiguous region of the program text. Consider the function definition in Figure 1. Its single clause is the outermost scope delimiter where variable `Y` directly occurs. The expressions "`Z = X`", "`Z = -X`" and "`Z = 0`" are not part of the (non-contiguous) scope of `Y`.

Expression B is to the right of expression A in expression E if A and B are directly in the same scope delimiter, and there exist expressions C, D_1, D_2, ..., D_n such that A is a subexpression of C, and the expression "$C, D_1, D_2, \ldots, D_n, B$" is a subexpression of E. Note that "$C, D_1, D_2, \ldots, D_n, B$" can be a sequence of expressions (in a "block"), a sequence of "patterns" (in a composite pattern), a sequence of "qualifiers" in a list comprehension or a sequence of "guards".

If a variable occurs in a pattern matching, then *the scope of this pattern matching* contains every expression to the right of the variable occurrence. Furthermore, if the pattern matching is

- in the head of a function clause, then the guards and the body of the clause is also part of the scope;
- in a list comprehension expression (the pattern of a generator), then the qualifiers to the right of the generator and the body of the list comprehension are also part of the scope;
- in a pattern match expression, then the right-hand side of that expression, and all expressions to the right of the pattern match expression are also part of the scope;
- in a branch of a "case", "receive" or "try" expression, then that branch of the expression and all the expressions to the right of the concerned ("case", "receive" or "try") expression are also part of the scope.

The *scope of a variable* is defined in the following way. Take the (existent and unique) outermost scope delimiter in which the variable directly occurs. The scope of the variable is the union of the scopes of the pattern matching occurrences of that variable that directly occur in that scope delimiter.

The rules describing the locality of variables are the following.

- Variable names occurring in the formal argument list of a function clause denote variables local to that function clause. A variable name occurring directly in a function clause but not occurring in the formal argument list of the function clause denotes a variable local to that function clause if and only if the function clause is not enclosed in the scope of a variable with that same name. (Note that the function clause is enclosed in the scope of a variable if and only if the function clause is a clause of an explicit fun-expression and either (1) the variable is a local variable of the enclosing scope delimiter, and the enclosed explicit fun-expression is within its–non-contiguous–scope, or (2) the enclosing scope delimiter itself is also enclosed in the scope of the variable.)
- Variable names occurring in the Pattern of a generator of a list comprehension denote variables local to the list comprehension. A variable name occurring directly in a list comprehension, but not in the Pattern of a generator of the list comprehension denotes a variable local to the list comprehension if and only if the list comprehension is not enclosed in the scope of a variable with that name.
- A variable name directly occurring in an element of a tuple or list denotes a variable local to that element of tuple or list if and only if that tuple or list is not enclosed in the scope of a variable with that name.

These rules reveal that occurrences of variable names in formal argument lists and patterns of generators of list comprehensions introduce variables that might *shadow variables* of the enclosing scope delimiter.

A *variable is visible* within its scope where none of the following limitations apply:

- A fun-expression creates a new scope for its parameters. When an existing variable name is used in one of the formal arguments, it creates a new variable that shadows the existing one (i.e. the outer variable is not visible inside the function.)
- A nested scope delimiter may introduce a new variable with the same name. The scope of such variables are excluded from the visibility region of the concerned variable (shawoding).
- Variables created in a `catch` or `try` expression are unsafe to be used outside that expression, so they are visible only inside the innermost enclosing `catch` expression.
- Variables that are created inside a branch of a branching expression (those are: `if`, `case` and `receive`), but are not bound a value in every branch, are unsafe to be used outside that expression, so they are not visible outside the expression.
- Variables created in the timeout expression of a `receive` construct's `after` branch, but are not bound in the body of that branch, are not considered to be bound in that branch at all.[3]

3. Binding variables

The different occurrences of a variable are classified by [3] as "binding occurrences" and "non-binding occurrences". The binding occurrences are those occurrences where the variable is bound to its (not unique, but final) value. Consider again the `sign_abs/1` function on Figure 1. The first three occurrences of `Y` are binding occurrences, binding three different, but final values to variable `Y`. The last occurrence is a non-binding one.

Note that the same syntactic form (pattern match) can refer to a binding or a non-binding occurrence of a variable. In Figure 2, the second occurrence of `X` in `d/1` is clearly a non-binding occurrence.

It is not possible to decide which are the binding occurrences of a variable: this depends on the specific Erlang implementation. In Figure 2, one of the occurrences of `Z` in `g/1` is a binding occurrence, and the other occurrence is a non-binding one; but it is unspecified (left to the Erlang implementation) which is the binding occurrence. To cover this strange situation, the concept of *potentional binding*

[3]Note that this behaviour does not comply with [3], and may be a compiler bug.

```
d(X) -> receive
          {X,Y} -> Y;
          _     -> 0
        end.

g() -> (Z=1) + (Z=1).
```

Figure 2: Binding and non-binding occurrences of variables

occurrences can be introduced. If a variable occurs in a pattern matching, then this occurrence is a potentional binding occurrence, if the scope of this occurrence is not part of the scope of another pattern matching occurrence of the same variable. In function `g/1`, both occurrences of `Z` are potentional binding occurrences.

A binding occurrence of a variable within a function clause is characterized as a node in the abstract syntax tree where a value is bound to its identifier. These locations are determined at runtime, and may depend on the execution control flow. To cover all of the potentional binding occurrences, a set of AST nodes, called *binding occurrence candidates*, is proposed. This set is determinable by static analysis in the following way.

- Consider a variable occurring in the program. An identifier that matches the name of this variable is a binding occurrence candidate, and we deem the subtree that contains only the identifier *closed*. Any other leaves in the syntax tree – identifiers with non-matching names or other kinds of nodes – are not closed, and yield no candidates.

- We group nodes with subtrees into two categories. Let us call these *conjunctive* and *disjunctive*, as is the connection between the closedness of the node and its children.

  - In the case of disjunctive nodes, of which the function body is a prime example, we search until we find the first subtree that is closed. If we find one, the parent node is closed, and the binding occurrence candidates are those that we have found up to the closed subtree inclusively. Note that the open subtrees before the closed subtree may contain candidates as well.
  - A conjunctive node (an `if` node, for example) is closed iff all of its subtrees are closed. Such nodes yield all the bindings of their subtrees whether closed or not.
  - The `case` and `try` constructs form a special case. Their argument may contain a match expression; in case it does, identifiers in the clauses are certainly not to bind. Therefore these constructs are treated as a

conjunctive node with the argument as the first child and the clauses' part as a second, disjunctive child.

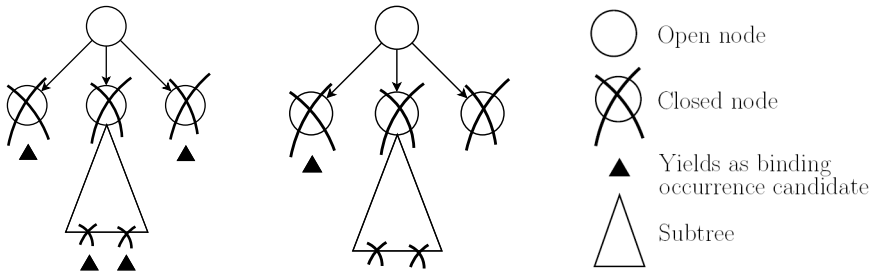

Figure 3: Difference between a closed conjunctive and disjunctive case

4. Conclusions and related work

Refactoring was first recognised as a distinct programming technique of its own in [5] which addressed a wide range of refactorings for object-oriented software, providing examples in Java. Many tools are available that support various kinds of renamings, extracting or inlining code, manipulating the class hierarchy etc. There is a good summary of the available tools and a catalog of well-known refactorings at [6]. Refactoring in functional languages has received much less attention. Haskell was the first functional language to gain tool support for refactoring [7]. Refactoring functional programs using database representation first appeared in [4] for the Clean language – a prototype is available from this research [9].

Refactoring Erlang programs is a joint research with the University of Kent [8], building on experiences with Haskell and Clean. The workgroup at Eötvös Loránd University is developping a refactoring tool that is based on a graph representation of Erlang programs and is mapped onto a database. This paper describes rules which are useful for the static analysis of variable occurrences: scoping, visibility and potential binding occurrences. These rules provide a better basis for verifying the conditions of, and performing, those refactoring transformations that are related to variables than the rules given in the Erlang reference manual [3]. Based on the rules presented here, a number of refactorings have already been implemented, such as Merge Subexpression Duplicates, Eliminate Variable and Extract Function.

References

[1] Armstrong, J., Virding, R., Williams, M., Wikstrom, C., Concurrent Programming in Erlang, *Prentice Hall*, (1996).

[2] Armstrong, J., Making reliable distributed systems in the presence of software errors, *PhD thesis, The Royal Institute of Technology*, Stockholm, Sweden, (December 2003).

[3] Barklund, J., Virding, R., Erlang Reference Manual, (1999), `http://www.erlang.org/download/erl_spec47.ps.gz`.

[4] Diviánszky, P., Szabó-Nacsa, R., Horváth, Z., Refactoring via database representation, *The Sixth International Conference on Applied Informatics (ICAI 2004), Eger, Hungary*, vol. 1, 129–136.

[5] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., Refactoring: Improving the Design of Existing Code, *Addison-Wesley*, (1999).

[6] Martin Fowler's refactoring site, `http://www.refactoring.com/`

[7] Li, H., Reinke, C., Thompson, S., Tool support for refactoring functional programs, *Haskell Workshop: Proceedings of the ACM SIGPLAN workshop on Haskell, Uppsala, Sweden*, (2003), 27–38.

[8] Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T., Refactoring Erlang Programs, *Proceedings of the 12th International Erlang/OTP User Conference*, (November 2006).

[9] Szabó-Nacsa, R., Diviánszky, P., Horváth, Z., Prototype environment for refactoring Clean programs, *The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, July 1–4.*