Hierarchical module namespaces in Erlang

Richard Carlsson
Computing Science Department
Uppsala University
Box 311, 751 05 Uppsala, Sweden
richardc@csd.uu.se

ABSTRACT

This paper describes how the Erlang language has been extended with a hierarchical namespace for modules, generally known as "packages", similar to that used for classes in Java.

1. INTRODUCTION

When ERLANG [1] was first created, it inherited a lot of its flavour from languages like Strand, Prolog and Parlog, which (at least in many implementations) have a similar view of program modules: these are program files, or "compilation units", each given a globally unique name, and each declaring some or all of its functions as *exported*. Exported functions may also be accessed from any other module in the system, while non-exported ("local") functions can only be referred to from within the same module. In ERLANG, it is required that files containing source or object code for a module must be given the same name as the module, plus a suffix which is `.erl` for source files, and `.beam` for object files for the BEAM abstract machine.

The name space for modules in all these languages is *flat*, i. e., when a particular module is referred to, this is always done by its full name, regardless of the context in which the reference is made: there is no way to express a reference to another module in relation to the current module. Furthermore, since programmers like to keep names short, module names such as `lists`, `math`, `queue`, `shell`, `random`, etc., quickly become used (all these examples are taken from the ERLANG standard library). When code from different vendors is combined in the same system, each such distribution possibly consisting of several hundred modules, the likelihood of one or more name clashes becomes large. Also, because of the meta-calls relatively often used in ERLANG programs, it is not always an easy task in such cases to rename the clashing modules uniquely without introducing errors, even if the source code is available. To keep the risk of clashes down, some programmers give modules abbreviated cryptic names such as `gb`, `rb`, `dbg`, etc., which is uninformative and could be considered bad programming style. The solution adopted by most programmers today is to always use prefixed names such as `snmp_supervisor`, `snmp_error`, `snmp_generic`, etc., but this is clumsy and is limited by the maximum length of file names on the host operating system.

In [3], I suggested how a package system like that used for classes in the Java [4] language could be adopted for ERLANG. This paper describes the final implementation, which will be included in the forthcoming release R9 of the ERLANG/OTP compiler.[1]

The rest of the paper is divided as follows: Section 2 describes the package system of Java and introduces the basics of a similar system for ERLANG. Section 3 describes extensions to the ERLANG language in order to make life easier for the programmer when using packages. Section 4 describes the extensions made to the ERLANG user shell, and section 5 contains a summary.

Erlang Workshop '02 Pittsburgh, USA

2. PACKAGES

In the Java language, each compilation unit is a *class*, which is similar to a module in ERLANG, the main difference being that ERLANG modules do not have distinct instances, and do not support inheritance. Java source and object files are, like ERLANG modules, given the name of the (public) class they contain plus the extensions `.java` and `.class`, respectively.

In Java, however, classes also belong to a *package*: this is a way of structuring the *names* of classes, and is orthogonal to the class inheritance hierarchy of Java. Thus the same concept can be applied to ERLANG, even though it is not an object-oriented language.

2.1 Packages in Java

If a Java source file (having the suffix `.java`) contains a *package declaration* stating a package name, then any class defined in that file belongs to the named package.[2] Furthermore, the package name also indicates to the Java implementation where the object file is located, but the details of this is implementation-dependent. If a file does not contain a package declaration, its definitions belong to the "empty" package; this is often useful for testing or writing simple applets.

A Java package name consists of a sequence of names separated by period characters, such as

```
java.rmi.server
```

[1] See `www.erlang.org` for the Open Source release.
[2] A Java source file can actually contain more than one class definition, but at most one may be declared public.

The full name of a class `RemoteObject` contained in this package, is then

```
java.rmi.server.RemoteObject
```

When the Java implementation tries to load the object file for a class by its full name, it typically uses its `CLASSPATH` variable. This is simply a sequence of file system paths in the host operating system, which are to be used as roots for the search in the order they are given. The package name is then subdivided at each period character into a sequence of one or more names. This is interpreted as a relative path in the host operating system, in a way that is system-dependent: in a Unix-like system, the relative path corresponding to the above package name would be

```
./java/rmi/server/
```

An attempt is then made to, for each path `ROOT` listed in `CLASSPATH`, load the file whose name is the concatenation of `ROOT`, the relative path for the package, the class name, and the object file suffix `.class`; in this example:

```
ROOT/java/rmi/server/RemoteObject.class
```

until such a file is found for some `ROOT`, or all paths in `CLASSPATH` have been tried. With this approach, object files are thus located in a set of directory trees, rather than in a set of flat directories. This can be generalised to other ways of loading object code, such as reading from network connections or archive files.

When a Java program refers to a class that is not defined in the same file, and by its class name only, the Java compiler will assume that the class is defined in the same package as that of the current file, and will not confuse it with classes of the same name in other packages. In this way, a package declaration creates a distinct name space for classes.

2.2 Introducing packages in Erlang

In the ERLANG package system, modules are named analogously to full class names in Java: for instance, the full name of a module `m` in a package `a.b.c` will be `a.b.c.m`, while its object file will typically be named `m.beam` and reside in a directory `ROOT/a/b/c/` for some `ROOT` in the search path of the ERLANG code server. The object file for a module whose full name does *not* contain any period characters, such as e. g. `io_lib`, is thus assumed to be located in some directory `ROOT/`, exactly as in the ERLANG implementations of today; for this example, the file would be `ROOT/io_lib.beam`.

To handle this first step, the only necessary modifications were to the code server, and a few related functions handling the location of source/object files. Only when the name of a module actually contains period characters does the behaviour differ from previously. This convention is fully compatible with existing code: code in packages can pass a full module name (typically as an atom) to old code, which can use the name obliviously, even for making meta-calls. This however requires that the full name is always used for identifying a module at run-time; we will discuss this in more detail in the next section.

To give an example of how this is used in practice, assume that the code path is set to some directory `ROOT`, and that we have two modules `com.acme.product_one.gui` and `com.acme.product_two.gui` (adopting the convention that the Internet domain name of the producer, in reverse, is used as a prefix for the package). The corresponding source and object files would in both cases be named `gui.erl` and `gui.beam` respectively, but reside in the different subdirectories `com/acme/product_one` and `com/acme/product_two` under `ROOT`. Thus, the code path can list relatively few directories (in this example a single directory), while object files as well as source files can be organized in subdirectories, and file names can be kept short. The code server is always given full module names, and will only look in the proper subdirectories for the corresponding object code.

Existing modules can easily be moved into packages without requiring recompilation of old code, by simply creating "stub" replacement modules in which all exported functions make a direct jump to the function of the same name in the corresponding packaged module; such stub modules will be small, and the extra call is a relatively small cost. A library module such as `lists` could e. g. be renamed `erlang.lists` or `erlang.lang.list`. Note that this presents a good opportunity to completely restructure (by renaming, splitting, moving individual functions, etc.) the existing standard modules, preferably according to the suggestion for a new set of standard modules made by Jonas Barklund [2]. However, the details of such a restructuring is outside the scope of this article.

2.3 Erlang module names and meta-calls

ERLANG programs frequently pass around module names as data, and use these names for dynamically choosing which module to call, through a *meta-call* mechanism such as

```
apply(<Module>, <Function>, [...])
```

where `<Module>` and `<Function>` are any expressions that evaluate to atoms at run-time. The `apply` function dynamically looks up and calls the corresponding function in the named module, with the given list of arguments. Similarly, a new process is typically created by calling

```
spawn(<Module>, <Function>, [...])
```

which behaves like `apply`, but spawns a new process to evaluate the call.

To illustrate, an often used way of passing a callback reference to an ERLANG application is to give the name of the module in which the callback function resides. The application typically contains a call like:

```
apply(Module, handle_event, [X, Y])
```

where `Module` is a variable, and the user might start the server with a call such as:

```
event_server:start(..., my_events)
```

after first creating a module `my_events` which exports a function `handle_event` with arity 2.

Thus, to retain compatibility with existing ERLANG programs, a value that represents a module name must be an absolute reference that does not need to be interpreted within the context of a package declaration. Furthermore, because existing code generally assumes that module names are atoms, it is not possible to introduce a new structured representation using e. g. lists. Therefore, the package structure of module names must be encoded in the character sequence, using a separator character.

It is also necessary that all automatically generated module names are expanded at compile time to *full* names, such

as those generated by the `?MODULE` preprocessor macro. For example:

```
spawn(?MODULE, server_code, [...])
```

will work as expected also within a packaged module.

3. LANGUAGE EXTENSIONS

So far, we have only described a way of storing object files in relation to the structure of module names. If the ERLANG language itself remained unchanged, this convention would force the programmer to write the full module names, always within single quotes, in all situations. This would be cumbersome and ugly, and miss one of the main points with structured namespaces: to be able to make references relative to the current package.

Today, ERLANG module names seldom or never contain period characters; one reason for this is that since the module name declaration of an ERLANG module has the form

```
-module(<A>).
```

where `<A>` is an ERLANG atom,[3] such a module would have to be declared as e. g.

```
-module('foo.bar').
```

giving the module name within single-quotes. Hence, it was assumed that the use of the period character as separator in module names could be adopted with few, if any, existing ERLANG programs needing rewriting.

However, simply allowing period characters in atoms without quoting was not possible, because periods are already used as separators in some ERLANG constructs, in a way that prevents this form of extension. Fortunately, it turned out to be quite straightforward to instead modify the grammar for our purposes.

3.1 New module declarations

The ERLANG grammar has been extended to accept module name declarations not only on the form

```
-module(<A1>).
```

where $\texttt{<A}_1\texttt{>}$ is an atom, but also more generally as

$$\texttt{-module(<A}_1\texttt{>.<A}_2\texttt{>}\cdots\texttt{.<A}_n\texttt{>).}$$

for $n \geq 1$, where all $\texttt{<A}_i\texttt{>}$, $i \in [1, n]$, are atoms. Because each atom might be given within single-quotes, it is necessary to check that the concatenation of atoms and periods do not form an illegal name, as in e. g.

```
-module(foo.'.bar'.baz).
```

where the resulting name `'foo..bar.baz'` would contain an empty segment, which is not allowed.

We now introduce some terminology:

- The *full module name* is the concatenation of the print names of the atoms $\texttt{<A}_1\texttt{>}\cdots\texttt{<A}_n\texttt{>}$ and the separating period characters. A full module name must not contain two adjacent period characters or end with a period character.
- A full module name that contains one or more period characters is said to be *qualified*.
- The *module name* is the part of the full module name that follows the last period character, if any, or is otherwise equivalent to the full module name.
- The *package name* is the part of the full module name that precedes the last period character, if any, or is otherwise the empty string.

For example, in a declaration

```
-module(fee.fie.foe_fum).
```

the full module name is `fee.fie.foe_fum`, the package name is `fee.fie` and the module name is `foe_fum`. For a module whose full name contains no period characters, as in

```
-module(io_lib).
```

the package name is the empty string, and the module name is equal to the full module name; thus, *the meaning of module name declarations in old code does not change.*

3.2 Remote calls

In general, within a module, all non-qualified module references are interpreted as relative to the same package. When a remote call on the form

```
<A>:<F>(...)
```

is encountered in a module, where `<F>` is any expression and `<A>` is an atom whose print name *does not contain period characters*, then that atom is interpreted as the name of a module in the same package as the current module. In this case, the compiler will automatically replace `<A>` with the corresponding full module name, by prefixing `<A>` by the package name and a period character. For example, if the call `fred:f()` occurs in a module whose package name is `foo.bar`, it will be replaced by the call `'foo.bar.fred':f()`.

This allows the programmer to e. g. create a module named `lists` in a package, and refer to that module directly by that name, without confusion with the standard module of the same name (which is in the "empty" package).

3.3 The period pseudo-operator

We also extend the grammar to allow calls to be written

$$\texttt{<A}_1\texttt{>.<A}_2\texttt{>.}\cdots\texttt{<A}_n\texttt{>:<F>(...)}$$

for atoms $\texttt{<A}_i\texttt{>}$, $i \in [1, n]$, $n \geq 1$ (with the same restrictions on the concatenated name as in a module declaration). Thus, a programmer is not forced to write a full module name within single quotes. For example, the calls

```
foo.bar.baz:f()
```

and

```
'foo.bar.baz':f()
```

are equivalent.

In fact, the use of `.` to concatenate constant atom literals is allowed more generally both in expressions and patterns, to make full module names easier to write, e. g. for metaprogramming purposes. It is a compile time error if the result is not a legal module name. Also, because of the lexical conventions of ERLANG, the period character may not be followed by whitespace.

[3] Atoms are a primitive datatype in ERLANG; they can be seen as nullary constructors, and are identified by their print names. Unless surrounded by single-quotes, their names must begin with a lowercase letter, and not contain other characters than letters, digits, or underscore ('_'). Examples of atoms are `foo`, `mad_hatter` and `'foo-bar'`.

3.4 Forcing absolute module references

A module in a named package could occasionally need to refer to a module in the "empty" package. To avoid interpreting such references as local, the module name must contain a leading period character, as in the call

```
'.lists':reverse(X)
```

The syntax for module names therefore also allows an initial period character, as in:

```
.lists:reverse(X)
```

Note that all previously existing ERLANG modules belong to the empty package, and thus do not need any changes.

In section 3.6.1, we describe an alternative way of referencing modules in the empty package, which is cleaner but not as general.

3.5 Imported functions

In ERLANG, functions can be declared as imported from other modules. For example, if a module contains a declaration

```
-import(foo, [... f/1 ...]).
```

all calls on the form `f(X)` will be taken to mean `foo:f(X)`. This now also works for qualified module names, e. g.:

```
-import(fee.fie.foo, [... f/1 ...]).
```

causes all calls `f(X)` to be interpreted as `fee.fie.foo:f(X)`. The specified module name is always assumed to be a *full* module name, even if it does not contain period characters.

3.6 Imported modules

Importing individual functions has the disadvantage of making the code more difficult to understand, because it is not clear at first glance whether a call is to a local function or to another module. However, always stating the whole module name at each remote call is clumsy and clutters the code when module names get long. A new form of import declaration has therefore been added, to allow only the final part of the full module name to be used, regardless of what package it belongs to.

3.6.1 Importing a module name

The new module import declaration has the form

```
-import(<M>).
```

where `<M>` is a *full module name*.

The occurrence of such a declaration allows the use of the final part of the imported module name in references to that module. For example, a declaration

```
-import(foo.bar.baz).
```

in module `a.b.m` would make a call

```
baz:f(...)
```

in that module synonymous to

```
foo.bar.baz:f(...)
```

In particular, note that since the imported name is always a full module name, a declaration such as `-import(lists).` would make a call `lists:f(...)` in the same module be equivalent to `.lists:f(...)`, i. e., referring to the module `lists` in the empty package. Using a module import in this way is the recommended method whenever possible when calls must be made to modules in the empty package, since a leading period character is not very visible and easy to leave out by mistake. However, the period notation must still be available for special cases, such as when the module name is already imported from some other package.

It is a compile-time error if the same module name is imported more than once. Also note that module-import declarations do not affect the interpretation of function-import declarations. For example, in:

```
-import(my.own.lists).
-import(lists, [reverse/1]).
```

the function `reverse/1` is imported from module `lists`, not from `my.own.lists`. It is recommended that functions are not imported in code that uses packages, since this hides the fact that the calls are in fact "remote". The module import mechanism allows a more explicit style of programming using the `:` notation, while keeping module references short.

3.6.2 Similarity to imported packages in Java

Java has a similar form of import declaration, on the two forms

```
import package.class;
import package.*;
```

where the former imports a particular class, and the latter all classes in a particular package. This allows the programmer to refer directly to any imported class without its package name; however, if two classes with the same name are imported, then neither may be used without specifying the package name.

Java is a statically typed language, and needs information about the types of all external classes referenced in a program file in order to compile that file. The Java compiler therefore searches for object files for such classes, recursively compiling source files where possible in order to produce any object files that are missing. This makes importing of all classes in a package possible, because the search order is well-defined, and all referenced classes must be present.

ERLANG, however, is dynamically typed, and the compiler never actually needs to examine other source files in order to compile a particular file. It would not be in line with ERLANG programming conventions to let the sets of existing object files in two distinct packages decide from which of these packages a particular module is imported. E. g., if we would import all modules from the packages `foo` and `bar`, then a reference to a module `m` would be resolved to either `foo.m` or `bar.m` depending on which package actually defines a module `m`. If the programmer assumed that `m` was found in `bar` and not in `foo`, and later a module `m` is added also to `foo`, then after recompilation the program might unexpectedly try to call `foo.m:f(...)` instead of the intended `bar.m:f(...)`, and this error would not be detected at compile time, but cause a run time failure.

For this reason, only full module names may be used in module import declarations in ERLANG.

3.7 Why no sub-package references?

The reader may wonder why, in a "hierarchical" namespace, there are no language constructs that allow the programmer to refer to a module in a sub-package of the current package using a relative name, rather than the full name.

For example, in a module `a.b.m1`, it would certainly be possible to let a call

```
c.m2:f()
```

be interpreted as equivalent to

```
'a.b.c.m2':f()
```

The problem with this approach, however, is that it would force all references to any module `p.m3` that is *not* in a sub-package of the current to be written as

```
.p.m3:f()
```

with a leading period character, to avoid being interpreted as `a.b.p.m3`. This could often result in programming mistakes. In fact, it can be expected that when references need to me made to modules outside of the current package, it will usually *not* be to a sub-package. Thus, the general mechanism for importing module names should be sufficient. The same design decision was apparently made in Java.

4. USER SHELL EXTENSIONS

The ERLANG user shell is a traditional command-line interface, where the user can enter normal ERLANG expressions for evaluation. It also defines a few pseudo-local utility functions; e. g., the user can enter

```
c(foo).
```

to compile the source file named "`foo.erl`" and load the resulting module to memory. (Note that with the package system, a file "`foo.erl`" could actually define a module `fee.fie.foo`; the package name is not reflected in the file name, but only in the search path.)

To save typing when using packages, a new shell function `import(<M>)` has been added, having the same effect as a declaration `-import(<M>).` in a module. E. g., entering

```
import(foo.bar.baz).
```

allows you to write simply `baz:f(...)` instead of the fully qualified `foo.bar.baz:f(...)`. In the shell, it is also allowed to re-import a module name, overriding the previous import.

If you want to check what a particular module name currently expands to, you can enter

```
which(<Module>).
```

Continuing the above example, evaluating `which(baz)` would return `'foo.bar.baz'`.

When working in the shell, you are referring to the system as it looks at the moment. Therefore, an import command similar to the `import` *package*`.*` of Java makes sense in a way it does not in a module definition (cf. section 3.6.2). Entering

```
import_all(foo.bar)
```

in the ERLANG shell will locate all modules belonging to the package `foo.bar` that can be found in the current search path, and import them. This is typically useful when you decide to work with modules in a particular package; by using `import_all(<Package>)` you can immediately refer to them by their module names only.

5. SUMMARY

The ERLANG language has been extended with a system of hierarchical module namespaces, or "packages", which is conceptually simple and backwards compatible with practically all existing code. The necessary changes to the system were relatively small and straightforward, and mainly localized to the syntax checking and pre-expansion stages of the compiler, the code server, and the user shell. Packages will be included in the forthcoming release R9 of Erlang/OTP.

6. ACKNOWLEDGEMENTS

I thank the anonymous referees for their feedback.

7. REFERENCES

[1] Armstrong, Virding, Wikström, and Williams. *Concurrent Programming in Erlang, Second Edition.* Prentice Hall, 1996.

[2] J. Barklund et al. Proposal 15: Built-in functions of Erlang. Erlang specification project, `http://www.bluetail.com/~rv/Erlang-spec/index.shtml`, June 1998.

[3] R. Carlsson. Extending Erlang with structured module packages. In *Proceedings of the 6th International Erlang/OTP User Conference.* Ericsson, 2000.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.