Department of Computer Science, University of Otago

Technical Report OUCS-2017-05

Child Modules for Erlang and Prolog

Author:

Richard O'Keefe
Department of Computer Science, University of Otago, New Zealand

---

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.php

Child Modules for Erlang and Prolog

Richard A. O'Keefe
Computer Science, Otāgo

Revised June 2017

Abstract

Prolog and Erlang have similar module systems, where modules in a flat namespace are both the sole form of encapsulation and the units of code loading and replacement. They also support the inclusion of text files as a way of sharing declarations and private functions between modules. This note proposes a replacement for text inclusion inspired by child modules in Ada.

1 A common problem and a common solution

Prolog and Erlang have many similarities. For historic reasons, the syntax of Erlang is very close to the syntax of Prolog. Both of them have a "flat" module system where a module is a chunk of code named by an atom, which may import routines from other modules and export routines to any module that is interested.

Neither Prolog nor Erlang has any "structured" kind of unit that is smaller than a module file. If you want to compose a module from smaller pieces, you use an "include" directive.

1.1 Prolog includes

ISO/IEC 13211-1:1995 section 7.4.2.7 says

> If *F* is an implementation-defined ground term designating a Prolog text unit, then Prolog text *P1* which contains a directive `include(`*F*`)` is identical to a Prolog text *P2* obtained by replacing the directive `include(`*F*`)` in *P1* by the Prolog text denoted by *F*.

This is typical standardese (more precisely, typical ISO Prolog standardese) for

> if a top level form is `:- include(`*F*`).`, where *F* is a file name, your Prolog system will act as though the top level forms inside that file appeared in place of the `:-include` directive.

In the formal version: "the Prolog text denoted by *F*" should be "the Prolog text contained in the text unit designated by *F*". Above all, the standard text suggests that the contents of the file just replace the `include(F)` part, leaving the `:-` and the full stop stranded just so they can cause trouble.

Include directives are mainly useful so that operator declarations can be shared by several modules; anything else can be imported as normal code.

1.2 Erlang includes

Section 4.2.4 "File Inclusion" of the Erlang Reference Manual, version 5.4.13[1] says

> The same syntax as for module attributes is used for file inclusion:
>
> ```
> -include(File).
> -include_lib(File).
> ```
>
> `File`, a string, should point out a file. The contents of this file are included as-is, at the position of the directive.
>
> Include files are typically used for record and macro definitions that are shared by several modules. It is recommended that the file name extension `.hrl` be used for include files.
>
> `File` may start with a path component `$VAR`, for some string name `$VAR`. If that is the case, the value of the environment variable `VAR` as returned by `os:getenv(VAR)` is substituted for `$VAR`. If `os:getenv(VAR)` returns `false`, `$VAR` is left as is.
>
> If the file name `File` is absolute (possibly after variable substitution), the include file with that name is included. Otherwise, the specified file is searched for in the current working directory, in the same directory as the module being compiled, and in the directories given by the `include` option, in that order. See `erlc(1)` and `compile(3)` for details.
>
> Examples:
>
> ```
> -include("my_records.hrl").
> -include("incdir/my_records.hrl").
> -include("/home/user/proj/my_records.hrl").
> -include("$PROJ_ROOT/my_records.hrl").
> ```
>
> `include_lib` is similar to `include`, but should not point out an absolute file. Instead, the first path component (possibly after variable substitution) is assumed to be the name of an application. Example:
>
> ```
> -include_lib("kernel/include/file.hrl").
> ```

[1] `http://www.erlang.org/doc/doc-5.4.13/doc/reference_manual/part_frame.html`

The code server uses `code:lib_dir(kernel)` to find the directory of the current (latest) version of Kernel, and then the subdirectory `include` is searched for the file `file.hrl`

Erlang's `include_lib` is not as big an addition to Prolog's `include` as it looks; Prolog systems commonly allow structured file names that do the same job more generally.

2 What the common solution gets right

- Common code only has to be written once; it can then be used many times.
- The feature is a standard part of the language; you can always use it.
- Any kind of declaration can be shared this way.
- While current practice seems to ignore it, you *can* build a module out of *un*-shared pieces this way.
- Source file inclusion is a familiar technique available in a wide range of languages, so programmers feel comfortable using it.

3 What the common solution gets wrong

- Source file inclusion is a familiar technique available in a wide range of languages, so programmers feel comfortable using it, even when another approach might be more appropriate. The whole Erlang preprocessor is seductively familiar to C hackers.
- Common code gets recompiled inside each module where it is used.
- The interface between an included file and a file that includes it is totally implicit; a maintenance programmer looking at an `:-include` or `-include` directive has no way whatever of telling what the included file needs from its includer or what it provides to its includer, other than going and looking.
- There are no encapsulation boundaries between inclusions in the same module. If module M includes files F and G, then F can see *everything* in M, including everything in G. So a maintenance programmer who is looking at F cannot be sure of understanding it without looking at everything it is included *into* and everything it is included *with*.
- In both Prolog and Erlang you can replace a module at run time; this can be an important part of keeping a long-lived program like a web service running while it is maintained. But you cannot replace *part* of a module that was included from another file.

- Typical cross referencers for these languages can tell you that routine *UR* in module *UM* uses routine *PR* from module *PM*, but do not tell you about dependencies that involve included files. For example, it cannot tell you that modules *M1* and *M2* both use included file *F*.

- A major problem for portability is that included files are not named in a portable way; the file names have to be operating-system specific, possibly even installation specific. There is nothing analogous to an OASIS catalogue or an Eiffel LACE file that can map from portable program-specific inclusion names to system-specific file names.

- If there is some feature provided by an inclusion, and there is more than implementation of it, the only way to conditionally select the implementation is by using some sort of preprocessor to conditionally select a directive. Example:

  ```
  -ifdef(use_ping).
  -include("ping.hrl").
  -else.
  -include("pong.hrl").
  -endif.
  ```

  This is an issue for Prolog, because there is no standard or commonly accepted conditional compilation feature for Prolog, but it's also a problem for Erlang. In particular, you cannot "drop in" a new or experimental version of an inclusion without changing the module file.

4 What could we do instead?

I propose a solution in which

- There are modules, just like we had before, and child modules, analogous to inclusions.

- File names never appear in source files, only module names and child module names.

- A separate configuration language says how to map module names and child module names to file names.

- By having more than one configuration file, you can have several different configurations for a set of modules.

- The interface between a module and a child is explicit.

- Children can be replaceable, although the default is that they are not.

- Items provided by a child are used without a module prefix in the parent.

5 Syntax, Erlang version

5.1 Module (reference manual 4.2.1)

-module(*Module_Name*).
(-export(*Exports*).
| -import(*Other_Module_Name*, *Imports*).
| -compile(*Options*).
| -vsn(*Version*).
| -behaviour(*Behaviour_Module_Name*).
| -*Other_Tag*(*Term*).
)*
(*Preprocessor directive.*
| -include(*File_Name*).
| -include_lib(*File_Name*).
| *Out of line child*
| *In line child*
| -*Other_Tag*(*Term*).
| *Function definition.*
)+

The only change here is the addition of *Out of line child* and *In line child*.

Modern Erlang has an optional type system, with `-type` and `-spec` declarations, and types may be imported and exported. Think of function `-spec`ifications as another kind of *Function declaration* and types as function-like values in a separate namespace.

5.2 *Out of line child*

-use_child(*Child_Name*,
From_The_Child[,
To_The_Child[,
integrated | replaceable]]).

From_The_Child is like an `-import` directive; it says that the child module is required to provide the listed functions. Those functions will be available in the parent *without* any module qualification; a child module is not a full module and module qualification applies only to things imported from full modules.

For any module or child module X, the *From_The_Child* lists of all its children must be disjoint, and none of them may mention anything defined in X proper. Nor may a *From_The_Child* list have any element in common with the *To_The_Child* list for the same child. However, "cyclic" dependencies between children are allowed. Example:

:- use_child(fred, [roast/1], [beef/2]).
:- use_child(mary, [beef/2], [roast/1]).

To_The_Child is like an `-export` directive; it says that the child module is allowed to use the listed functions visible in the parent, and those only. If *To_The_Child* is omitted, it is taken to be an empty list. Note that a child module is allowed to import functions from full modules, and that includes its own ancestor.

The `integrated` option says that the child is to be bound early with the parent. The interface specification controls visibility, but the compiler may consider the module together with all its integrated children (and their integrated children, transitively) as a single unit and do whatever type inference, inlining, or other optimisation it wishes. The `replaceable` options says that the child is to be bound late with the parent. Whatever code is generated must allow the child to be replaced at run time. The default is `integrated`.

5.3 *In line child*

-begin_child(*Child_Name*,
From_The_Child[,
To_The_Child).

(-import(*Other_Module_Name*, *Imports*).
| -vsn(*Version*).
| -*Other_Tag*(*Term*).
)*
(*Preprocessor directive.*
| -include(*File_Name*).
| -include_lib(*File_Name*).
| *Out of line child*
| *In line child*
| -*Other_Tag*(*Term*).
| *Function definition.*
)+

-end_child(*Child_Name*).

The preprocessor `?MODULE` hack remains available in the proper body of a full module. It is *not* available in any child module, whether in line or out of line. You are supposed to be able to understand most things of importance for understanding a child module just by looking at it. Even an in line child may have been `-included`, so a child module might be shared by any number of modules in which case you don't know what `?MODULE` means. In particular, code like

```
-child(fred, [f/0]).

f() ->
    f(?MODULE).
```

```
f(mummy) -> true;
f(daddy) -> false.
```

can be done with plain `-include`, but intentionally *cannot* be written using child modules of any kind.

A child module may not `-export` anything. The closest it can come is to provide features to its parent.

In contrast, a child module may `-import` from other (full) modules. Functions imported from other modules cannot be provided to the parent, only functions defined in the child or available in it from children of its own. The scope of an `-import` directive in an *In line child* is limited to that child. Conversely, an `-import` directive in a parent has no effect on its children. The idea is that you should be able to take an out of line child and move it in line, or an in line child and move it out of line, without any change to its body. There is one exception, discussed next.

An in line child may not contain a `-compile` directive; the compiler options that apply to an integrated child are the same as those that apply to its parent. An out of line child may contain such directives.

An in line child may contain a `-vsn` directive of its own.

A child module may not contain a `-behaviour` directive. Only a full module may be an instance of a behaviour.

The body of a child module is just like the body of a full module.

An in line child is closed by an `-end_child` directive; the *Child_Name* is repeated for readability and must match the *Child_Name* in the corresponding `-begin_child`.

5.4 *Out of line child*

-child(*Child_Name*,
From_The_Child[,
To_The_Child]).

(-import(*Other_Module_Name*, *Imports*).
| -compile(*Options*).
| -vsn(*Version*).
| -*Other_Tag*(*Term*).
)*
(*Preprocessor directive.*
| -include(*File_Name*).
| -include_lib(*File_Name*).
| *Out of line child*
| *In line child*
| -*Other_Tag*(*Term*).
| *Function definition.*
)+

Module_Name, *Other_Module_Name*, and *Child_Name* are all unquoted atoms.

5.5 Exports

[*functor* (**,** *functor*)*]

An export list is a non-empty list of functors, where a functor is either *name*/*arity*, referring to an ordinary function, or #*name*/*arity*, referring to an abstract pattern, or a reference to a type. There is no point in an empty export list, so it isn't allowed.

5.6 Imports

[(*functor* (**,** *functor*)*)?]

An import list is a possibly empty list of functors. An empty import list can be useful to state a dependency on another module without allowing the abbreviation of any function names, so it is allowed.

From_The_Child is

[*item* (**,** *item*)*]

This is a non-empty list of items, where an item is either a functor or #*record_name* or a reference to a type. Records may only be required of or provided to an integrated child (either in line or out of line). Long term, abstract patterns are envisaged as a replacement for records. These days, Erlang has "maps", which are also meant as a replacement for records.

Restricting record items to integrated children means that there is no need to mention anything more than the record name. Mentioning the record name means that it is obvious to a maintenance programmer which children what records come from.

5.7 Exports to children

To_The_Child is

[(*item* (**,** *item*)*)?]

The list of things provided to a child is a possibly empty list of exportable items. Records may only be provided to an integrated child (either in line or out of line).

5.8 Example

```
-module(demo).
-export([f/0]).
-use_child(shared_stuff, [k/1, #r]).
f() -> k(#r{x=1}).
```

```
% Eof

-child(shared_stuff, [k/1, #r]).
-record(r, {x=0}).

k(#r{x=0}) -> 137;
k(#r{x=1}) ->  42.
% \textit{Eof}

-module(listy).
-export([length/1, reverse/1]).

-begin_child(length, [length/1]).
length(Xs) -> length(Xs, 0).

length([_|Xs], N) -> length(Xs, N+1);
length([],     N) -> N.
-end_child(length).

-begin_child(reverse, [reverse/1]).
reverse(Xs) -> reverse(Xs, []).

reverse([X|Xs], Ys) -> reverse(Xs, [X|Ys]);
reverse([],     Ys) -> Ys.
-end_child(reverse).
% Eof
```

6 Syntax, Prolog version

The Prolog version is very similar to the Erlang version, so is described in less detail. Prolog directives begin with `:-` instead of `-`, and Prolog module declarations have always had the form

:- module(*Module_Name*, [
functor (, *functor*)*]).

The new forms are

:- child(*Child_Name*,
From_The_Child[,
To_The_Child]).

:- use_child(*Child_Name*,
From_The_Child[,
To_The_Child[,
integrated | replaceable]]).

:- begin_child(*Child_Name*,
From_The_Child[,
To_The_Child[,
integrated | replaceable]]).

:- end_child(*Child_Name*).

6.1 Example

```
:- module(demo, [f/1]).
:- use_child(shared_stuff, [k/2]).
f(X) :- k(r(1), X).
end_of_file.

:- child(shared_stuff, [k/2]).
k(r(U), V) :-
    k_aux(U, V).

k_aux(0, 137).
k_aux(1,  42).
end_of_file.

:- module(listy, [length/2, reverse/2]).

:- begin_child(length, [length/2]).
length(Xs, N) :- length(Xs, 0, N).

length([],     N,  N).
length([_|Xs], N0, N) :-
    N1 is 1 + N0,
    length(Xs, N1, N).
:- end_child(length).

:- begin_child(reverse, [reverse/2]).
reverse(Xs, Ys) :- reverse(Xs, [], Ys).

reverse([],     Ys, Zs).
reverse([X|Xs], Ys0, Ys) :-
    reverse(Xs, [X|Ys0], Ys).
:- end_child(reverse).
end_of_file.
```

The predicate names `child/[2,3]` are too useful to take away from programmers, so `:- child` is only interpreted as a child module header when it is the very first directive in a file.

7 The Configuration Language

This is a very preliminary draft, and is more intended as something to get the idea across than as anything approximating a serious proposal.

The configuration language has two primary tasks:

- To map module names (and child module names) to file names.
- To provide a *conditional* mapping so that the versions appropriate to a particular configuration can be chosen.

Why have an elaborate system like this?

So that we can separate module names from file names.

Both Prolog and Erlang in principle allow a module name to be any sequence of characters that could be used as the name of a procedure, and that is literally *any* sequence of characters whatever. The problem is that this is not a good fit for file systems, in three ways.

- A file system may, and Windows does, impose a limit on the total length of a file name that is shorter than the limit set by Prolog or Erlang on name length.
- A file system may, and real file systems do, impose limits on the characters that may be used in file names. For example, UNIX and Windows both disallow the NUL character in file names, whereas 'a\0b' is a perfectly good identifier in Prolog and Erlang.
- A file system may identify characters that the language distinguishes. For example, Windows ignores alphabetic case when matching file names, and treats / and \ the same, whereas 'a' and 'A' are different in Erlang and Prolog.

The other issue with mapping module names to file names, as Erlang does and Prolog typically does, is that it is hard to change the location of a file without changing the name of the module.

Basically, the whole point of the configuration language is to solve this.

SGML catalogues are a good analogy for what we are trying to do here. The official specification is SGML Open Technical Resolution TR401:1997[2]. There is now an XML equivalent[3], with all the readability disadvantages of XML. James Clark[4] has an explanation of the SGML version. Some of the entries that can occur in a catalogue are:

PUBLIC *pubid sysid* These map a portable (public) object name to a system-dependent name. Practically everything that can be named in an SGML document can have up to three names: a simple name that is unique within

[2] `http://xml.coverpages.org/sotr9401-a2.html`
[3] `http://xml.coverpages.org/walsh-ent-spec20010109.html`
[4] `http://www.jclark.com/sp/catalog.htm`

some class of nameable objects, a public identifier, which is a portable unique identifier which is supposed to be unique across the whole planet-wide world of things in SGML documents, and a system identifier, which is any system-dependent way of referring to an object, but in XML is always a URL.

ENTITY ***entity-id sysid***

NOTATION ***notation-id sysid***

DOCTYPE ***doctype-id sysid*** These map type-specific simple names to files (or URLs).

SGMLDECL ***sysid***

DOCUMENT ***sysid*** These say where to find the SGML declaration for a document (a sort of parametric meta-grammar) or the document to be parsed, if either is not otherwise specified.

SYSTEM ***sys-id-1 sys-id-2*** An SGML document may already contain system-dependent identifiers. This catalogue entry lets you override those, forcibly remapping some file.

BASE ***sysid*** If something is mapped to an absolute system identifier, there's no more to be said. If it's mapped to a relative system identifier v(relative file name or relative URL), that is to be interpreted relative to some base. The base for interpreting relative names is either the location of the catalogue itself, or the sysid provided in a BASE declaration.

CATALOG ***sysid*** This is a sort of (nearly) position-independent inclusion feature. If you can't resolve an object name using the rules in a catalogue, try each of the catalogues named in any CATALOG declarations.

DELEGATE ***pubid-prefix catalogue-sysid*** ¿This is also a sort of inclusion feature. What it says is that any public identifier which has *pubid-prefix* as a prefix should be resolved by looking in the catalogue found at *catalogue-sysid* instead of this catalogue.

The simplest possible scheme for our purposes would be a simple list of {module name, file name} pairs, with all conditional processing done by some other means, such as the macro processor M4. This could work, but M4 is Turing-complete, and it would be nice to have something simpler.

7.1 Grammar of the configuration language

```
configuration = (inclusion | var-def | search-def)*
                default? module-def*

inclusion = "<" file-name
```

```
var-def = uc-identifier ( "|" guard "=" expression )+
        | uc-identifier "=" expression

guard = guard "&&" guard
      | guard "||" guard
      | "~" guard
      | "(" guard ")"
      | expression relop expression

expression = expression "+" expression
           | expression "-" expression
           | "(" expression ")"
           | lc-identifier
           | number
           | uc-identifier

search-def = "$" uc-identifier ( "|" guard "=" search-list )+
           | "$" uc-identifier "=" search-list

search-list = (search-list ",")? file-name

file-name = "/"? file-part ("/" file-part)* ("(" file-part ")")?

file-part = regular-file-part "." simple-file-part
          | regular-file-part

regular-file-part = (regular-file-part "++")?
                    (simple-file-part | "*")

simple-file-part = lc-identifier
                 | uc-identifier
                 | "$" uc-identifier
                 | string

default = "*" "=" search-list

module-def = lc-identifier ( "|" guard "=" module-rhs )+
           | lc-identifier "=" module-rhs

module-rhs = search-list children?
           | children

children = "{" search-def* default child-def* "}"
         | "{" search-def*         child-def+ "}"
```

```
child-def = child-name ( "|" guard "=" child-rhs )+
          | child-name

child-name = ("." (lc-identifier | uc-identifier))+

child-rhs = module-rhs
```

A file-part may only contain a "*" if it is in the search list of a default. A default rule says that unless overridden by a later rule, a module is to be sought by substituting its name for the "*" in the search list.

A module-rhs may omit the search-list only when there is a default; an omitted search-list means to use the default fule.

Example:

```
$STDLIB = lib/stdlib/src
$ERL    = erl

lists = $STDLIB/lists.$ERL {
    $LISTS = $STDLIB/lists.d
    .deprecated = $LISTS/old_stuff.$ERL
    .sorting    = $LISTS/sorting  .$ERL
}
...
```

Example with defaults:

```
$STDLIB = lib/stdlib/src
$ERL    = erl

* = $STDLIB/*.$ERL
lists = {
    $LISTS = $STDLIB/lists.d
    * = $LISTS/*.$ERL
    .deprecated = $LISTS/old_stuff.$ERL
    % .sorting is handled by the inner default
}
% sets is handled by the outer default
...
```

Basically, a configuration file is a back-to-front lazy functional program, because there are no mutable data structures. Lazy, because nothing is evaluated until it is needed. Evaluation is driven by first processing the tops of all the module declarations, and then looking up the children of those modules as they are demanded by the compiler. Back-to-front, because the usual approach in functional languages is that the *first* declaration wins, while here the *last* rule to match any need is used. This ordering is chosen so that inclusions, going at the front, can be over-ridden by later definitions.

File names use slashes, but those slashes are operators, not literal text. Whether they map to "/", to "\", to ":", or even whether /a/b/c maps to [a.b]c, is system-dependent. In the same way, "." precedes an "extension" (also known as a file type), and whether that maps to "." or to ";" or to something else is system-dependent. Code may be kept in plain or compressed archives (Unix ".a", ".zip", ".jar", and so on, or MVS partitioned data sets), and the "(" ")" part of a file name refers to selecting a member from such a file. For example, we might have

```
$MYLIB = lib/otago/raok.zip
* = $MYLIB(*)
```

Identifiers in simple-file-parts beginning with a lower case letter are literal text. Identifiers beginning with an upper case letter are meant for "wild-card" child module matching. Identifiers preceded by a dollar sign are path names.

Conditional selection uses Haskell syntax.

An inclusion says to simply copy all the definitions in the included file.

8 Implementation

Child modules share a feature with textual inclusion: if two modules use the same child, each has its own copy. This is because a child module may use items from its parent.

This means that `integrated` child modules basically are textual inclusion, just with encapsulation, and can be processed using renaming.

If a child module is `replaceable`, the simplest way to deal with it is to generate a normal separate module, with the parent's module name prefixing calls to imported functions. Each use needs its own name. For example, a replaceable child Z of a replaceable chld Y of a module X might be called "Z,Y,X". The automatically generated modules would be held separately from normal modules.