Parameterized modules in Erlang

Richard Carlsson
Information Technology Department, Uppsala University, Sweden
richardc@csd.uu.se

ABSTRACT

This paper describes how the Erlang programming language could be extended with parameterized modules, in a way that is compatible with existing code. This provides a powerful way of creating callbacks, that avoids the limitations involved with function closures, and extends current programming practices in a systematic way that also eliminates a common source of errors. The usage of parameterized modules is similar to Object-Oriented programming, and is naturally complemented by the currently underused feature of *behaviours* (interface declarations), which are also explained in detail.

1. INTRODUCTION

1.1 The Erlang language

The ERLANG [1] programming language is a strict, dynamically typed functional language without destructive updates. It has built-in support for concurrency through *processes*, which communicate by asynchronous message passing. The unit of compilation is a *module*, where each module is uniquely named and contains a number of functions, such that only those functions explicitly marked as *exported* can be accessed from other modules. A feature known as "dynamic code replacement" allows a new version of a module to be loaded at any time. ERLANG is mainly used for writing complicated control applications with a high degree of parallellism, such as telecommunication switching software. Some of these applications are very large, ranging from a couple of hundred thousand up to 1.5 million lines of code. Yet, apart from the module system, the language in itself has no particular features that assist the structuring of large applications.

1.2 Motivation

Suppose that a service (e.g., a server of some kind), which is implemented by some module `B`, is called from another module `A`. The interface of `B` allows `A` to specify the name of a *callback module* `C` to be called from `B` for handling certain tasks, as shown in Figure 1. This is a form of parameterization of the behaviour of `B`, and is a common idiom in ERLANG programs. The module `C` must then provide a set of functions (one or more) with specific names and arities, as described in the documentation of `B`. This set of functions is referred to as the *interface* of the callback module, as expected by `B`. (A callback module could implement any number of additional functions as long as it implements all the expected interface functions.)

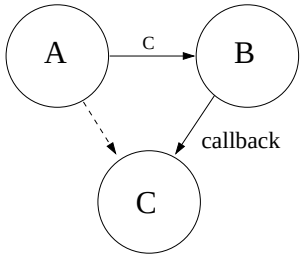

Figure 1: Callback module usage

In this style of programming, it often happens that there is a need for `A` to pass additional information to the callback module `C` (cf. Figure 1), beyond that which is passed directly from `B` to `C` as specified by the callback interface. There are some different ways to handle this:

- If there are only very few variations of `C`, and the data is known at compile time, it may be possible to create different versions $\texttt{C}_1$, $\texttt{C}_2$, ..., either by hand, or automatically using some form of source code templates or code generation tools. Then `A` can pass a suitable name $\texttt{C}_i$ depending on the desired behaviour. This method is obviously very limited, and cannot be generalized.

- A tempting solution is to put the additional information in some named location where it can be accessed by both `A` and `C` – i.e., in some form of global variable. (Some real-world examples found in ERLANG programs are the process dictionaries, `ets`-tables, and files.) It is however universally agreed that arbitrary use of global variables is bad programming practice.

- In some cases, the author of `B` has foreseen this need, and has included an extra parameter in the interfaces from `A` to `B`, and from `B` to `C`, so that `B` is given the extra information (often referred to as "user data") by `A`, and includes it in the calls to `C`. In this case, `B` never

Erlang Workshop '03 29/08/2003, Uppsala, Sweden

inspects or modifies the data, which has no effect on the functionality of `B` itself. It is therefore unfortunate that it should clutter the interfaces of `B` and `C` in this arbitrary and irrelevant way, and furthermore affect the implementation details of `B`.

- If only a single callback function is ever called from `B`, then if the implementation used a function closure (or "fun") instead of naming a callback module, the problem would be solved, since a closure can carry its own "extra information" – the values of its free variables at the time of its instantiation. However, if a later version of `B` could want to call more than one callback function (and the functions are logically related), using a single callback module allows the interface to evolve over time.

Other problems with using function closures for anything except local and short-term purposes are posed by the code replacement mechanism and the transparent network distribution of ERLANG. For example, if `B` is given a function closure for use as a callback whenever the need arises, and holds on to this for a longer period, it is possible that meanwhile, the module containing the code for the closure is replaced, which could render the closure unusable. Furthermore, if a closure is passed across the network to another ERLANG node, the same version of the module that contains the closure's code must exist on both nodes. If the version on the remote node is older (or newer), the closure cannot be used.[1] In contrast, when a function is being called by name (giving both the name of the module and the function), the latest loaded version of the module is always used.

(It should also be mentioned that for historical reasons, ERLANG still supports a form of by-name, pseudo-functional values consisting of pairs `{M, F}` of atoms `M` and `F` naming the module and function to be called. There is no way to pass "extra information". The only advantage of these today are that they do not suffer from the code change problem.)

To solve the above problems, we suggest a new language feature: parameterized modules.

2. PARAMETERIZED MODULES

A parameterized, or *abstract* module, is a module with free variables, much like a lambda expression (or "fun expression"). A lambda expression evaluates to a *closure*, which is a functional value and can be applied like any function. Similarly, an abstract module can be *instantiated* to yield a *module instance*, which can be used in a qualified function call just like any module name.

2.1 Defining a parameterized module

In order to allow us to declare an abstract module, we extend the module declaration syntax so that we may specify a list of variables, as follows:

$$\texttt{-module(}\textit{name}\texttt{, [V}_1\texttt{, ..., V}_n\texttt{]).}$$

This will make the variables $\texttt{V}_1, \ldots, \texttt{V}_n$ available over all functions in the module, i.e., the scope of their declaration is the whole module. There are no restrictions whatsoever on what kind of values the variables may hold – in particular, a variable may be bound to an instance of another abstract module, as we will see later.

[1]The interaction between anonymous function closures and code versioning is a complicated and interesting matter, but lies outside the scope of this paper.

2.2 Creating a module instance

Just like a lambda expression becomes instantiated to a "fun-value" (function closure), we need to be able to instantiate abstract modules. For this purpose, the compiler automatically creates a function named `new/`n in each abstract module, where n is the number of variables in the module declaration. Thus, to create a module instance `M`, we call the corresponding `new` function:

$$\texttt{M = }\textit{name}\texttt{:new(A}_1\texttt{, ..., A}_n\texttt{)}$$

where *name* is the name of the abstract module, and $\texttt{A}_1, \ldots, \texttt{A}_n$ are the actual parameter values to be bound to the formal parameters $\texttt{V}_1, \ldots, \texttt{V}_n$. It is important to note that except for `new/`n and the standard functions `module_info/1` and `module_info/2`,[2] *no other functions in an abstract module may be called directly by the module name alone.*

The representation of the value `M` is not defined here. Ideally, a new built-in opaque datatype `module` should be added to the ERLANG language for representing module instances.

2.3 Calling a function in an abstract module

If an instance `M` of an abstract module has been created as described above, that value can be used in a function call, just as if it was the name of an ordinary module:

$$\texttt{M:f}_i\texttt{(A}_1\texttt{, ..., A}_{k_i}\texttt{)}$$

where k_i is the arity of the i:th function $\texttt{f}_i$ exported by the abstract module, and $\texttt{A}_1, \ldots, \texttt{A}_{k_i}$ are the arguments. This is done by extending the functionality of the `:` operator to accept both atoms (module names) and module instances as the left-hand side operand. The built-in functions `apply/3`, `spawn/3`, etc., are similarly extended.

Note that this allows existing code to use abstract module instances *without modification*: where previously an atom (a module name) was expected as a reference to a callback module, as in e.g. the standard library `gen_server` module, a module instance could be passed instead.

As mentioned previously, when a function is called by explicitly specifying the target module, using the `:` operator, the semantics of ERLANG dictate that the latest loaded version of the code for that module is used (in other words, the binding is done at the time of the call). This property should hold also when calling instances of abstract modules, i.e., the latest version of the abstract module is always used. This way, abstract modules may be used where function closures are not suitable.

2.4 A simple example

Figure 2 shows the code for two modules `main` and `print`, where `print` is parameterized with respect to the variable `Name`. The function `main:start/0` creates two instances of `print` and calls the function `message/1` for both instances. The resulting output is:

```
Humpty: 'Hello!'
Dumpty: 'Hi!'
```

[2]In an Object-Oriented language, these would be called the *static* functions.

```
% File: main.erl
-module(main).
-export([start/0]).
start() ->
  M1 = print:new("Humpty"),
  M2 = print:new("Dumpty"),
  M1:message("Hello!"),
  M2:message("Hi!"),
  ok.

% File: print.erl
-module(print, [Name]).
-export([message/1]).
message(Text) ->
  io:fwrite("~s: '~s'", [Name, Text]),
  ok.
```

Figure 2: A simple example

In other words, whenever the instance `M1` is called, the value `"Humpty"` is used for `Name`, and whenever `M2` is called, the value `"Dumpty"` is used.

2.5 Generic server example

Figure 3 shows the implementation of a "generic server" module `server`, parameterized with respect to the variables `Name` and `Mod`. The function `start/1` starts a server process for a particular instance of `server`, returning the new process identifier. The server registers itself under the given name (an atom), and also includes the name in each reply to any received request. The argument to `start/1` is passed to the `init/1` function of the callback module to produce the initial state.

The callback module must implement at least the three functions `init/1`, `handle/2`, and `cleanup/1`. The first takes the data given to the `start/1` function and produces the initial server state; the `handle/2` function takes a request and the current state and produces a result and a new state, and the `cleanup/1` function takes the final state and handles any necessary cleanup before the server process terminates.

Note that as the server is implemented, the callback module `Mod` can be either an atom, naming an ordinary module, *or* an instance of an abstract module – the code for the server itself does not depend on this. Furthermore, the server code is very clean, since the `Mod` variable does not need to be passed around explicitly as an extra parameter, or hidden as a component of the state value. Also note that to start the server, as in:

```
Process = Server:start(...)
```

it is not necessary to know anything about the values of the `Name` and `Mod` parameters – these are only needed when the server module instance is created, which could occur at a completely different point in the program, as in:

```
Server = server:new(my_server, CallbackModule)
```

Figure 4 shows a possible implementation of a callback module (named `callback`) for such a generic server. It provides the necessary functions, which implement a simple counter with the operations `inc(N)`, `dec(N)`, `reset`, and `read`. The three first operations modify the counter value and return `ok`, while the last returns the current value without changing it. The `init/1` function simply sets the initial counter value, and the cleanup function does not have to do anything.

```
% File: server.erl
-module(server, [Name, Mod]).
-export([start/1]).

start(Arg) ->
  spawn(fun () -> init(Arg) end).

init(Arg) ->
  register(Name, self()),
  loop(Mod:init(Arg)).

loop(State) ->
  receive
    {request, Sender, R} ->
      {A, State1} = Mod:handle(R, State),
      Sender ! {reply, Name, A}
      loop(State1);
    halt ->
      Mod:cleanup(State),
      ok
  end.
```

Figure 3: A generic server

To make things more interesting, the `callback` module is also parameterized with respect to a module `Log`, which must provide a function `event/1`, and is used for keeping a log over the sequence of received requests. We simply assume that there is an implementation of such a logger available in a module named `log`.

To create a callback instance, we thus do the following:

```
CallbackModule = callback:new(log)
```

The value `CallbackModule` is then passed to `server:new/2` as described above to create an instance of the `server` module, which in its turn is used to actually start the server process.

What happens now is that the `log` module will be called upon each received request, but the client who issues the requests to the server does not specify this in any way. Furthermore, the server code does not pass any explicit "extra information" to the callback module, and the name of the logger is not hidden in the server state or in any global variable. Nor is the code for the callback module cluttered by code for digging out the logger module reference from the state parameter, or from any other location: the only traces are the declaration of the `Log` variable in the module header, and the uses of `Log` in the code.

2.6 A source of errors eliminated

In current ERLANG programs that reference callbacks by name, it is in general impossible to automatically detect if a particular atom or string literal in the source code will be used as the name `M` of a module in a call `M:f(...)`. As a consequence, the set of modules called by a program cannot be exactly determined by debugging tools. Sometimes, module names are even constructed at run time by string operations. This makes debugging of such applications very difficult. If a module name has been misspelt, or the target

```
% File: callback.erl
-module(callback, [Log]).
-export([init/1, handle/2, cleanup/1]).

init(Value) ->
  Log:event({init, Value}),
  Value.

handle({inc, N}, Value) ->
  Log:event({inc, N}),
  {ok, Value + N};
handle({dec, N}, Value) ->
  Log:event({dec, N}),
  {ok, Value - N};
handle(reset, _Value) ->
  Log:event(reset),
  {ok, 0};
handle(read, Value) ->
  Log:event(read),
  {Value, Value}.

cleanup(Value) ->
  Log:event({stopped, Value}),
  ok.
```

Figure 4: A callback implementation

module has been renamed but the code not updated to reflect this, the result will be a run-time error when the name is eventually used in a call.

By replacing all such so-called *meta-calls* with calls to abstract modules, for example by changing

```
Module = foobar
```

to

```
Module = foobar:new()
```

we can make sure that *all* references to modules by name are detectable as such (because of the context), and so can be found by automatic tools.

3. THE RELATION TO OBJECT-ORIENTED PROGRAMMING

In Object-Oriented Programming [2, 4, 7], an *object* is a piece of data, associated with a number of operations on that data. Depending on the programming language involved, there may be different facilities available for creating this association, but in general, the following holds:

- Each object belongs to a *class*, such that all objects of the same class have the exact same set of data fields and associated operations.
- A *class definition* describes both the individual data fields and the implementations of the operations (or *methods*), that every object belonging to the class will have.
- A new object is created by *instantiating* the class; usually by calling a *constructor method*, passing values to be used at object creation time. When an object is no longer needed, a corresponding *destructor method* is often called to perform any necessary cleanup operations before the object can be completely discarded. (Constructor and destructor methods are often automatically generated by the compiler.)
- Each created object instance has a distinct *identity*; even if two objects are of the same class, and were created by calling the same constructor methods with identical arguments, they remain separate objects.
- The data fields of a particular object instance constitute the *state* of the object. While in some languages fields can be declared as *constant*, and then cannot be changed after object instantiation, in the general case all objects have modifiable state.
- When a method is called, apart from its normal arguments it must also be told *which object instance* to operate on. This mechanism is typically hidden by the language, but in effect, the object reference is passed to the method as an extra argument. For example, the syntax `myObject.print(x)` may be rewritten internally to code resembling `print(x, myObject)`. The extra "invisible" parameter to such methods is usually referred to as the *this reference*.
- Usually, *inheritance* is also supported by the language, so that a newly created class can be declared as an extension of a previously existing class, by adding new data fields and functions, or new implementations of existing functions. Objects belonging to the new class automatically belong also to the previous class (the *superclass*), and recursively, to all of its superclasses. When a method is called, the nearest implementation in the hierarchy is used, overriding those higher up.

It is easy to see that an abstract module is very similar to a class. Both are instantiated to yield values that are used for calling functions (methods). Also, both need to pass a *this* reference, as we shall see in the following section. However, there are some important differences:

- A module instance has no modifiable state. Although an abstract module can have "accessor functions" that return the values of the module parameters, these values cannot be changed once instantiated.
- Instances of abstract modules do not have identity. Two instances created using the same constructor (the `new(...)` function) and arguments are equivalent.
- In an Object-Oriented language, the arguments passed to the constructor functions are typically not immediately available over the whole class definition, but only within the constructor method. It is the responsibility of the programmer to write code that explicitly transfers the arguments to data fields (which may then be accessed by the other methods) when the constructor is executed. The constructor code may also perform other initialization tasks that are executed for each new object instance.

In contrast, when an abstract module is instantiated, there is no programmer-specified initialization to be performed, and no need to write explicit constructor code. Furthermore, since there is no state, and no identity, there is no need for destructor functions.

Finally, we note that the issue of inheritance is orthogonal to the topic of this paper, and we will not pursue the matter further here, except remark that inheritance could most probably also be added to the ERLANG language, if desired.

4. THE 'THIS' REFERENCE

When a function is called through a module instance `M`, the code must be able to access the values that were given to the constructor function `new(...)` when `M` was created. Obviously, these values must be somehow embedded in `M`, but they also need to be transfered to the called function. (This is exactly the same issue as with the values of the free variables in function closures.) There are two main ways of doing this:

- Each value can be extracted from `M` before the call and be passed separately as an extra parameter to the function. (This is known as "closure conversion".)
- A reference to `M` itself can be passed to the function as a single extra parameter. The values are then extracted from `M` as needed, in the code of the called function. An important advantage of this method is that the function will always implicitly have access to the value `M` itself, which often turns out to be useful.

The latter method is also the one used in Object-Oriented languages, as previously described. By convention, the programmer can usually access the reference directly by the name `this`.

In Erlang, it would not only be convenient to be able to access the "this" reference, but actually necessary, in order to support dynamic code replacement. To give an example, in the "generic server" example in Figure 3, each time after a request has been handled, a tail-recursive[3] call to `loop` is executed, in order to handle further requests. Since this call is not qualified with the module name (i.e., it is a "local" call), it will not be redirected when a new version of `server` is loaded.

This is usually not acceptable, since a server process may run for a very long time, and it should be possible to update the code without restarting the process. Therefore, such loops are typically implemented by writing the tail-recursive call as a fully qualified call, as in `server:loop(State1)`. But in our example, this is not possible, because `server` is an abstract module, and the only function which can be called directly is `server:new(...)`.

Thus we see that in order to support dynamic code replacement in our server implementation, we need to be able to use the "this" reference. Adopting the convention that in every function of an abstract module, the variable `THIS` is always implicitly bound to the current module instance, we can write the tail recursive call as `THIS:loop(State1)`.

5. ABSTRACT MODULES AND ERLANG BEHAVIOURS

A "behaviour" is the name in the ERLANG community for what is otherwise more commonly known as an "interface", e.g. in Java [5]. Behaviour declarations are also currently the only constructs that are statically type-checked by the ERLANG compiler.

[3]Last call optimization is a required feature of ERLANG implementations.

5.1 How behaviours work

A module is declared as implementing a particular behaviour `B` by adding the declaration `-behaviour(B)`. The definition of the behaviour `B` is found by executing the function call `B:behaviour_info(callbacks)`, which returns a list of function names (represented as 2-tuples). For example, the standard library behaviour `application` is defined by the following code:[4]

```
-module(application).
-export([..., behaviour_info/1, ...]).
...
behaviour_info(callbacks) ->
    [{start,2},{stop,1}].
...
```

i.e., a module declared with `-behaviour(application)` must define the two functions `start/2` and `stop/1`.

The compiler checks that the module which contains the declaration also defines the corresponding functions, and issues a warning otherwise. Thus, the behaviour-defining module `B` must be available in the system for this check to be performed. (However, it is not a compile-time error if `B` is not present, or the check fails in any other way; thus a module can always be compiled independently of any other modules, if necessary, for example during bootstrapping.) Currently, the compiler also warns if a module contains more than one behaviour declaration, but there is no real reason for this, except that behaviours are still being regarded as something of an "advanced feature".

5.2 The Behaviour behaviour

It can be immediately noted that the act of defining a behaviour is in itself a recurring pattern, and could be defined as follows:

```
-module(behaviour).
-behaviour(behaviour).
-export([behaviour_info/1]).

behaviour_info(callbacks) ->
    [{behaviour_info,1}].
```

All behaviour-defining modules, such as `application` above, and even the module `behaviour` itself, as shown, could then include the declaration `-behaviour(behaviour)`.

5.3 Abstract modules and behaviours

Today, the use of callback modules (by name) is relatively common in ERLANG programs, but is perceived as a rather advanced programming trick. With abstract modules, it becomes more natural to use such "plug-ins", both because one can easily define a variable over a whole module (no need to explicitly pass extra parameters for the callbacks), and because the problem of passing "additional information" along with the callback is solved in a transparent way.

However, this also means that there will be an increased need to specify the requirements on a particular plug-in in a more formal way. The most basic specification is of course the interface, i.e., the "behaviour".

Looking back at our "generic server" example, the module `callback` in Figure 4 uses a logger module which is expected

[4]It is not uncommon that a behaviour-defining module also contains other exported functions, and thus serves as both a definition and a working component.

to contain a function `event/1`. This simple behaviour could be defined as follows:

```
-module(logger).
-behaviour(behaviour).
-export([behaviour_info/1]).

behaviour_info(callbacks) ->
    [{event,1}].
```

The documentation of the `callback` module could then specify that the module parameter `Log` must implement the `logger` interface. This will provide the author of such a logger module with a way to automatically check that his code at least implements all the required functions. We can expect that if abstract modules are introduced in ERLANG, the number of such mini-behaviours will grow quite large, so that any medium-sized application will typically define a number of behaviours for its internal use, and a few more for interoperability with other programs.[5]

Obviously, the checking of behaviour declarations is quite limited. Because ERLANG has no static typing, it is not possible to specify the types of the function parameters in a behaviour and have these automatically checked. Nor is it possible to declare that the type of a particular module parameter `M` is "module that implements behaviour `B`" and have the compiler check that `M` is only used to call functions defined in `B`. However, to get the assurance that one has remembered to implement all the necessary functions when creating a callback module is a very good start.

5.4 Behaviour of an abstract module

Of course, we want to be able to use behaviour declarations also in abstract modules. This requires no extra machinery. Looking at Figure 3, the `server` module is parameterized with respect to a callback module `Mod`. The expected interface of `Mod` could be specified as:

```
-module(server_callback).
-behaviour(behaviour).
-export([behaviour_info/1]).

behaviour_info(callbacks) ->
    [{init,1},{handle,2},{cleanup,1}].
```

We could then add the declaration:

```
-behaviour(server_callback)
```

to the module `callback` in Figure 4, like this:

```
% File: callback.erl
-module(callback, [Log]).
-behaviour(server_callback).
-export([init/1, handle/2, cleanup/1]).

init(Value) -> ..

handle(Reqest, Value) -> ...

cleanup(Value) -> ...
```

The fact that `callback` is an abstract module (with parameter `Log`) does not affect the check for functions `init/1`, `handle/2`, and `cleanup/1`.

[5]In comparison, the current Erlang/OTP distribution, containing more than a thousand modules, defines a total of 6 behaviours.

6. RAVIOLI CODE WARNING

In the Object Oriented Programming world, "ravioli code" is what you get when you have factored your program into too many small chunks of code, so that it becomes impossible to keep track of where the actual work is being done. (It's "all in the sauce", which of course is hard to get a good grip of.) With abstract modules, these kinds of programs become possible also in ERLANG. In the worst case, a program could start with instantiating hundreds of abstract modules, finally creating an "application" module instance `M` and calling `M:start/0`, and it could then take weeks to understand (even for the original author) which parts of the code are actually being called, from where, and at what time.

This kind of overuse of parameterizing modules should be avoided. The strategy to be used mainly depends on the programming problem: some problems map easily onto a set of functions without any real need for parameterizing the module itself – in those cases, the temptation to create an abstract module should be resisted. In other cases, parameterizing the module solves the problem in a very elegant manner. Use abstraction with discretion.

7. RELATED WORK

Module systems come in many different forms and flavours. A module system usually serves several purposes, such as code organization, separate compilation, abstraction (by hiding implementation details), code reuse (by creating suitably abstract components), and name space separation.[6] In Object-Oriented languages, modules are typically identified with classes [8, 5]. In others, such as Ada [9] or ML [10], a module (or *package*) is any collection of programming language entities. In some systems, modules can be nested.

If the language is statically typed and supports separate compilation, the compiler must be able to typecheck the uses of other modules when a single module is being compiled. This can be done by consulting the actual source code of the referenced modules, or by using *interface files* that contain only the interface specifications and can be delivered separately from the source code. (In some cases, the compiled object code contains all necessary interface information, e.g. in Java [5].) In some systems, the static typechecking does not allow mutually recursive dependencies between modules.

Parameterized modules are also known as *generic* modules. In Standard ML [10], parameterized modules are called *functors*. (In mathematics, a functor is a mapping from one category to another, or sometimes any function that operates on other functions.)[7] Module systems that allow abstraction with respect to module references are sometimes called *higher-order*. A *higher-order functor* is a functor whose components may themselves be functors, allowing for example *partial* instantiation of a functor. This is usually not possible in statically typed systems like Standard ML.

Some languages do not allow abstract modules to be instantiated at run-time, but only at compile-time or link-time, as in Ada [9] or Modula-3 [3]. In these cases, abstract modules are more like *templates*, as found in C++ [11], creating one separate, slightly different copy of the code for each instance. In languages that do allow run-time instanti-

[6]A good overview of module systems can be found in [12].

[7]Rather unfortunately, the term has recently also been used in the area of Object-Oriented programming to denote objects of a class that mainly contains functions.

ation, e.g. Standard ML, the implementations typically use the same techniques that are used for objects in Object-Oriented languages. (Most programmers make only limited use of functors in ML because of the resulting performance penalty.) Techniques like *specialization* or *partial evaluation* [6] can be used to generate more efficient code when only a few instances with partially static parameters are being created.

8. IMPLEMENTATION NOTES

- We have made a proof-of-concept prototype implementation, to show that the ideas presented in this paper work as supposed, and do not interfere with existing syntax. We hope to extend this to a full implementation in the near future.
- The current implementation of parameterized calls on the form `M:f(...)` in Erlang/OTP is such that existing compiled code does not need to be recompiled in order to handle the case where `M` is an instance of an abstract module, as long as the runtime system is updated.
- A new instruction needs to be added to the BEAM abstract machine code external format, in order to support more efficient calls using abstract modules, and the compiler must be extended to generate such instructions, when both the function name and the arity (but not the module) are known. By using caching techniques, as those used in certain Object-Oriented systems, the overhead of looking up the target address can be made very small.
- The current implementation of calls `M:f(...)` is very inefficient. With the new compilation scheme, even though handling of abstract modules is added, the net result should be an overall speedup. Calling an abstract module is expected to be only slightly slower than calling a function closure, so that code can be efficient even when abstract modules are heavily used.
- It is likely that compiled code for abstract modules needs to be marked in some way, so that tools like a debugger can know that the functions of such modules cannot be called without a reference to a "current module instance". We suggest that the compiler automatically sets the module attribute `abstract = true` for this purpose.

9. CONCLUDING REMARKS

We have described a straightforward way to extend the ERLANG programming language with abstract modules, a construct which solves several programming problems encountered in ERLANG programs of today. The technique is not new – the same thing is done in Object Oriented Programming to create instances of classes, the main difference being that a module instance in ERLANG does not have a state component, since there are no destructive updates. The concept of abstract modules can be traced back to languages like Ada [9] and Standard ML [10], but the system we suggest here owes more to dynamically typed Object Oriented languages such as Smalltalk [7]. Adding parameterized modules to ERLANG fits in well with the existing but little used language feature called *behaviours* (interface specifications) which offer a limited but very useful form of compile time checking. We expect that user-defined behaviours will become a very common thing if parameterized modules are added.

10. ACKNOWLEDGMENTS

The author would like to thank Erik Stenman, Tobias Lindahl and Per Gustafsson, as well as the anonymous referees, for their comments on previous versions of this paper, and Mikael Pettersson for discussions on efficient implementation and assistance with the prototype.

11. REFERENCES

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[2] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, second edition, 1997.

[3] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *SIGPLAN Notices*, 27(8):15–42, Aug. 1992.

[4] O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, Sept. 1966.

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

[6] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[7] A. C. Kay. The early history of Smalltalk. *ACM SIGPLAN Notices*, 28(3):69–75, Mar. 1993. The Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II).

[8] B. Meyer. *Eiffel, The Language*. Prentice Hall, Englewood Cliffs, 1992.

[9] Military Standard. Reference manual for the Ada programming language. Technical Report ANSI/MIL-STD-1815A-1983, United States Government Printing Office, 1983.

[10] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[11] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[12] M. Zenger. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, Ecole Polytechnique Federale de Lausanne, Switzerland, draft version, 2003.