Compiling Erlang to Scheme

Marc Feeley and Martin Larose

Université de Montréal
C.P. 6128 succursale centre-ville
Montréal H3C 3J7, Canada
{feeley,larosem}@iro.umontreal.ca

Abstract. The programming languages Erlang and Scheme have many common features, yet the performance of the current implementations of Erlang appears to be below that of good implementations of Scheme. This disparity has prompted us to investigate the translation of Erlang to Scheme. Our intent is to reuse the mature compilation technology of current Scheme compilers to quickly implement an efficient Erlang compiler. In this paper we describe the design and implementation of the **Etos** Erlang to Scheme compiler and compare its performance to other systems. The Scheme code produced by Etos is compiled by the Gambit-C Scheme to C compiler and the resulting C code is finally compiled by `gcc`. One might think that the many stages of this compilation pipeline would lead to an inefficient compiler but in fact, on most of our benchmark programs, Etos outperforms all currently available implementations of Erlang, including the Hipe native code compiler.

1 Introduction

Erlang is a concurrent functional programming language which has been mostly developed internally at Ericsson for the programming of telecom applications. The language is not purely functional because of its support for concurrent processes and communication between processes. Scheme shares many similarities with Erlang: "mostly" functional programming style, mandatory tail-call optimization, dynamic typing, automatic memory management, similar data types (symbols, lists, vectors, etc). Section 2 and Sections 3 briefly describe these languages (a complete description can be found in [3] and [16, 7]).

There is growing interest in Erlang in industry but due to its "in-house" development there is a limited choice of compilers. As the implementors of these compilers freely admit [1], "Performance has always been a major problem". On the other hand there are many implementations of Scheme available [18] and the good compilers appear to generate faster code than the Erlang compilers available from Ericsson (for example Hartel *et al.* [13] have shown that the "pseudoknot" benchmark compiled with Ericsson's BEAM/C 6.0.4 is about 5 times slower than when compiled with the Gambit-C 2.3 Scheme compiler).

Because of the strong similarity between Erlang and Scheme and the availability of several good Scheme compilers, we have begun the implementation of an Erlang to Scheme compiler called “Etos”. Our goal is to reduce development efforts by exploiting the analyses and optimizations of the Gambit-C Scheme to C compiler. It is reasonable to believe that most of the Gambit-C technology can be reused because the similarities between the languages outweigh the differences (infix vs. prefix syntax, pattern matching vs. access functions, `catch/throw` vs. `call/cc`, and concurrency). When we started this project it was not clear however if the many stages of the compilation pipeline would allow efficient code to be generated. In the rest of the paper we explain the major design issues of an Erlang to Scheme compiler and how these are solved in Etos 1.4, and show that its performance is very good when compared to other Erlang compilers.

2 Scheme

This section briefly describes Scheme for those unfamiliar with the language.

Scheme is a lexically scoped dialect of Lisp (invented by Sussman and Steele in 1975 [19] and enhanced regularly since then) which is both small and expressive. It is an expression-based language with garbage-collection and so promotes the functional programming style (but side-effects on variables and data-structures are permitted). The language requires that tail-recursion be implemented properly [6]. Several builtin data types are available, all of which are first-class and have indefinite extent: boolean, character, string, symbol, list, vector (one dimensional array), procedure (of fixed or variable arity), port (file handle), number (unlimited precision integers and rationals (i.e. exact numbers), and floating point and complex numbers). Procedures are closed in their definition environment (i.e. they are “closures” containing a code pointer and environment) and parameters are passed by value. An anonymous procedure is created by evaluating a `lambda` special form (see example below). Scheme is one of the few languages with first-class continuations which represent the “rest of a computation” and a construct, `call/cc`, to transform the current (implicit) continuation into a user-callable procedure. All arithmetic functions are generic, e.g. the addition function can be used to add any mix of number types.

3 Erlang

Erlang, like Scheme, is a garbage-collected expression-based language that is lexically scoped (but with unusual scope rules as explained in Section 8), properly tail-recursive, dynamically typed and which uses call-by-value parameter passing. The data types available are: number (floating-point numbers and unlimited precision integers), atom (like the Scheme symbol type), list, tuple (like the Scheme vector type), function, port (a channel for communicating with external processes and the file system), pid (a process identifier), reference (a globally unique marker), and binary (an array of bytes). Integers are used to represent characters and lists of integers are used to represent strings. Erlang's arithmetic operators are generic (any mix of numbers) and the comparison operators can compare any mix of types.

Erlang's syntax is inspired by Prolog (e.g. `[x,y,z]`, `[]` and `[H|T]` denote lists, variables begin with an uppercase letter and atoms with lowercase, pattern matching is used to define functions and take apart data). Erlang does not provide full unification as in Prolog (i.e. a variable is not an object that can be contained in data). Note also that a guard can be added to a pattern to constrain the match (third clause in the example below). The only way to bind a variable is to use pattern matching (in function parameters, the `case`, `receive`, and *pattern=expr* constructs). In particular in *pattern=expr* the expression is evaluated and the result is pattern matched with the pattern, variables bound in the process have a scope which includes the following expressions.

The language was designed to write robust concurrent distributed soft real-time applications in telephony. Local and remote processes are created dynamically with the `spawn` function, and interacted with by sending messages (any Erlang object) to their mailbox which sequentializes and buffers incoming messages. Messages are drained asynchronously from the mailbox with the `receive` construct, which extracts from the mailbox the next message which matches the pattern(s) specified by the `receive` (a timeout can also be specified).

Exceptions are managed using the forms `throw` *expr* and `catch` *expr*. Evaluating a `throw` *X* transfers control to the nearest dynamically enclosing `catch`, which returns *X*. Predefined exceptions exist for the builtin functions.

Erlang supports a simple module system, which provides namespace management. Each module specifies its name, the functions it exports and the functions it imports from other modules. The form `lists:map` indicates the function `map` in the module `lists`.

Here is a small contrived example of an Erlang function definition showing off some of the features of Erlang:

```
f(green,_)              -> 1.5;               % ignore second parameter
f([H|_],Y)              -> T=Y+1, {H,T*T};    % return a two tuple
f(X,Y) when integer(X)  -> lists:reverse(Y);  % X must be an integer
f(X,Y)                  -> lists:map(fun(Z) -> [Z,X+Z] end, Y).
```

This is roughly equivalent[1] to the following Scheme definition:

```
(define f
  (lambda (x y) ; parameters of f are x and y
    (cond ((eq? x 'green) 1.5) ; return 1.5 if x is the symbol green
          ((pair? x)
           (let ((t (+ y 1))) ; bind t to y+1
             (vector (car x) (* t t))))
          ((integer? x)
           (reverse y)) ; y better be a list
          (else
           (map (lambda (z) ; pass an anonymous procedure to map
                  (list z (+ x z))) ; create a list
                y))))) ; y better be a list of numbers
```

[1] There are subtle differences such as `(integer? 2.0)` is true in Scheme, but `2.0` is not an integer in Erlang.

4 Portability vs Efficiency

Early on we decided that portability of the compiler was important in order to maximize its usefulness and allow experiments across platforms (different target machines but also different Scheme implementations). Etos is written in standard Scheme [7] and the generated programs conform fairly closely to the standard.

It is clear however that better performance can be achieved if non-standard features of the target Scheme implementation are exploited. For example, the existence of fast operations on fixed precision integers, i.e. fixnums, is crucial to implement Erlang arithmetic efficiently. Fixnums are not part of the Scheme standard but all of the high performance Scheme compilers have some way to manipulate them. To exploit these widespread but not truly standard features, the generated code contains calls to Scheme macros whose definition depends on the target Scheme implementation. The appropriate macro definition file is supplied when the Scheme program is compiled. Not all Scheme implementations implement the same macro facilities, but this is not a problem because each macro file is specific to a particular Scheme implementation. This approach avoids the need to recompile the Erlang program from scratch when the target Scheme implementation is changed. For example, the Erlang addition operator, which is generic and supports arguments of mixed float and unlimited precision integer types, is translated to a Scheme call of the `erl-add` macro. The macro call `(erl-add x y)` may simply expand to a call to a library procedure which checks the type of `x` and `y` and adds them appropriately or signals a run time type error, or if fixnum arithmetic is available, it may expand to an inline expression which performs a fixnum addition if `x` and `y` are fixnums (and the result doesn't overflow) and otherwise calls the generic addition procedure.

Using a macro file also allows to move some of the code generation details out of the compiler and into the macro file, making it easy to experiment and tune the compiler. For example the representation of Erlang data types can easily be changed by only modifying the macro definitions of the operations on that type.

5 Direct Translation

We also wanted the translation to be direct so that Erlang features would map into the most natural Scheme equivalent. This has several benefits:

- Erlang and Scheme source code can be mixed more easily in an application if the calling convention and data representation are similar. Special features of Scheme (such as first-class continuations and assignment) and language extensions (such as a C-interface and special libraries) can then be accessed easily. For this reason, adding extra parameters to all functions to propagate an exception handler and/or continuation would be a bad idea.
- A comparison of compiler technology between Erlang and Scheme compilers will be fairer because the Scheme compiler will process a program with roughly the same structure as the Erlang compiler.
- The generated code can be read and debugged by humans.

When a direct translation is not possible, we tried to generate Scheme code with a structure that would be compiled efficiently by most Scheme compilers. Nevertheless there is often a run time overhead in the generated Scheme code that makes it slower than if the application had been written originally in Scheme. For example, Erlang's "`<`" operator is generic (it works on numbers as well as lists and other data types) but in most application programs it is only used to compare numbers. The code generated by Etos can't use Scheme's "`<`" primitive directly because it works on numbers only.

6 Data Types

The most important Erlang data types have a direct equivalent in Scheme, as explained in this section.

6.1 Numbers

Scheme numbers are organized into a class hierarchy: integer $\subseteq$ rational $\subseteq$ real $\subseteq$ complex. Independently of their class, numbers have an "exactness". For instance `2.0` denotes the inexact number 2 and `1/2` denotes the exact number 0.5. Scheme exact integers correspond to Erlang integers. In both Scheme and Erlang, integers can be of limited range. The Erlang specification requires at least 24 bit integers but all available compilers support unlimited precision integers by using a bignum representation when the integers are larger than can fit in a fixnum. Scheme inexact reals correspond to Erlang floats.

An unfortunate consequence of this representation is that testing for an Erlang integer or float translates into two tests in standard Scheme (i.e. `(and (integer? x) (exact? x))` tests if `x` is an exact integer). The test `(integer? x)` is typically quite expensive because it must return true on both exact integers and on inexact reals which happen to have a null fractional part. Again, some non-standard features can help to do this quicker, for example in Gambit-C: `(or (##fixnum? x) (##bignum? x))`.

6.2 Atoms

Scheme symbols can be used to represent Erlang atoms. Both can contain arbitrary characters and symbols can be compared for equality efficiently with the `eq?` predicate (which is simply a pointer comparison in many implementations of Scheme). The Scheme procedures `string->symbol` and `symbol->string` are equivalent to the Erlang built-in functions `list_to_atom` and `atom_to_list` except that the former deals with strings (which is a separate data type in Scheme).

One complication is that Scheme is a case-insensitive language and Erlang is case-sensitive. Variable names and symbols in the source of Scheme programs are stripped of their case. A simple solution for converting Erlang function and variable names is to prefix uppercase letters with an escape character (i.e. `^`), so that the Erlang variable `ListOfFloats` becomes `^list^of^floats` in Scheme.

Atoms are handled differently. The only way to force a particular case for symbols in Scheme is to use the procedure `string->symbol`. This means that Erlang constants containing atoms (e.g. the constant list `[1,tWo]`) must be created at run time using `string->symbol`. This is done by storing the objects created into global variables once in the initialization phase of the Scheme program and references to these globals replace references to the constants. Constants not containing atoms get converted to Scheme constants. For example, the Erlang call `f([1,tWo],[3,4])` gets converted to:

```
(define const1 (string->symbol "tWo")) ; global definitions
(define const2 (list 1 const1))
... (f const2 '(3 4)) ...
```

Alternative representations for atoms which were rejected are:

- Strings: no special treatment for uppercase letters is needed but the equality test is much more expensive.
- Symbols with escape character for uppercase letters: requires an unnatural and inefficient translation of `list_to_atom` and `atom_to_list`.

Gambit-C provides a (non-standard) notation for symbols that preserves case (e.g. `|tWo|`) so it was possible to reference atoms literally in code and constants.

6.3 Lists

Both languages handle lists similarly. In Scheme, lists are made up of the empty list (i.e. `()`) and pairs created with the `cons` primitive or the variable arity `list` primitive. The primitives `car` and `cdr` extract the head and tail of a list.

6.4 Tuples

Scheme vectors are the obvious counterpart of tuples. Vectors are constructed either with the variable arity `vector` primitive (Erlang's `{...,...}`), the `list->vector` primitive (Erlang's `list_to_tuple`), or the `make-vector` primitive (which creates a vector of length computed at run time).

A minor incompatibility is that tuples are indexed from 1 (with the `element` builtin function) and Scheme vectors are indexed from 0 (using `vector-ref`).

A more serious problem is that lists and vectors are the only compound data structures in standard Scheme. Since the Erlang data types port, pid, reference, and binary don't have a direct counterpart in Scheme, they must be implemented using lists or vectors. We have used vectors to implement these data types (as well as tuples and functions) because their content can be accessed in constant time. The first element of the vector is a symbol which indicates the type and the data associated with the type is in the remaining elements. Thus the tuple `{1,2,3}` is represented by the Scheme vector `#(tuple 1 2 3)`. Note that with this representation, tuple indexing does not require a run time decrement of the index to access an element. However, an Erlang type test translates to two Scheme tests. Thus `(and (vector? x) (eq? (vector-ref x 0) 'tuple))` tests if `x` is a tuple (we need not worry about the `vector-ref` being out of bound because empty vectors are never created by Etos).

A more compact representation which is based on the ability to test object identity with `eq?` is to use no tag for tuples and a special tag for non-tuples:

```
(define pid-tag (vector 'pid))
(define make-pid (lambda (...) (vector pid-tag ...)))
(define pid?
  (lambda (x)
   (and (vector? x)
        (> (vector-length x) 0)
        (eq? (vector-ref x 0) pid-tag))))
```

This representation was not used because type testing (a frequent operation in pattern matching) is more expensive in this representation. One more test is required for non-tuples as shown above and many more tests for tuples (we must check that the first element is not one of the tags `pid-tag`, etc).

6.5 Functions

Scheme procedures are the obvious counterpart of Erlang functions. Erlang functions are of fixed arity so the variable arity mechanism of Scheme is not necessary. Both Erlang and Scheme can create and call functional objects.

Unfortunately, this direct representation does not support error detection. Erlang's general function calling mechanism needs to ensure that the function that is being called is of the appropriate arity, and signal a run time error if it isn't. Because there is no standard way in Scheme to extract the arity of a procedure or to trap the application of a procedure to the wrong number of arguments, functional objects are represented as a tagged vector which contains the function's arity and the corresponding Scheme closure.

Toplevel functions of a module contain the arity information in their name so no arity test is needed when they are called. For example the function `bar` of arity 2 in module `foo` is translated to a Scheme lambda-expression of arity 2 bound to the global variable `foo:bar/2` (a valid variable name in Scheme). A call such as `foo:bar(1,2)` is then translated to a Scheme call to `foo:bar/2` which is guaranteed to be bound to a procedure of arity 2.

6.6 Ports, Pids, References and Binaries

The remaining Erlang data types can be represented with tagged Scheme vectors as shown above. Ports, which allow interaction with external processes (such as device drivers written in C), will clearly have to be built with some implementation specific extension to Scheme (i.e. a foreign function interface). There are no raw binary array data types in standard Scheme so a space inefficient vector based representation must be used. Scheme strings can't be used because there is no constraint on the size of characters and the `integer->char` procedure may not implement a natural encoding (such as ASCII). A compact representation is possible in Gambit-C by using bytevectors (arrays of 8, 16 and 32 bit integers).

7 Front End

To ensure compatibility with existing Erlang compilers, Etos' parser specification was derived from the one for the JAM interpreter and processed by our own Scheme parser generator [8, 5]. The original parser constructs a parse tree built of tuples. Because Etos needs to attach semantic information on the nodes of the parse tree, a conversion phase was added to extend the tree nodes with additional fields. This conversion also computes the bound variables at each node and performs constant propagation and constant folding. Constant propagation and folding are mainly needed to avoid allocation of structures which are constant, such as in the definition `f(X) -> Y={1,2}, [X,Y,3,4].` which gets compiled as though it were: `f(X) -> [X|[{1,2},3,4]].` The list `[{1,2},3,4]` is represented internally as the Scheme constant list `(#(tuple 1 2) 3 4)`.

Following this, the free variables before and after each node are computed. This is done as a separate pass because the bound variable analysis requires a left-to-right traversal of the parse tree, whereas the free variable analysis requires a right-to-left traversal. The free variables are needed to efficiently translate `case`, `if`, and `receive` expressions, which is explained in the next section.

8 Binding and Pattern Matching

8.1 Binding in Erlang

Erlang's approach for binding variables is a relic of its Prolog heritage. Binding is an integral part of pattern matching. Once it is bound by a pattern matching operation, a variable can be referenced in the rest of a function clause but can't be bound again (unless it has become an "unsafe" variable, see below). For example, in `f({A,B}) -> [X,X,X]=A, B+X.` the function `f` will pattern match its sole argument with a two-tuple. In the process, the variables `A` and `B` get bound to the first and second element respectively. After this, `A` is referenced and pattern matched with a list containing three times the same element. Note that the first occurrence of `X` binds `X` to the first element of the list and the remaining occurrences reference the variable.

8.2 Binding in Scheme

In Scheme the basic binding construct is the lambda-expression and binding occurs when a procedure is called, as in `((λ (x) (* x x)) 3)`. Here the variable `x` is bound to `3` when the closure returned by evaluating the lambda-expression is called with 3. Scheme also has the binding constructs `let`, `let*` and `letrec` but these are simply syntactic sugar for lambda-expressions and calls. For example the previous expression is equivalent to `(let ((x 3)) (* x x))`.

Erlang syntactic categories:

$\langle\texttt{const}\rangle$: constant
$\langle\texttt{ubvar}\rangle$: unbound variable
$\langle\texttt{bvar}\rangle$: bound variable
$\langle\texttt{exp1}\rangle$, $\langle\texttt{exp2}\rangle$: arbitrary expressions
$\langle\texttt{pat1}\rangle$, $\langle\texttt{pat2}\rangle$: arbitrary patterns
$\langle\texttt{fn}\rangle$: function name

Expression translation:

$E(\langle\texttt{const}\rangle, k) = \texttt{(}k\ C(\langle\texttt{const}\rangle)\texttt{)}$
$E(\langle\texttt{bvar}\rangle, k) = \texttt{(}k\ N(\langle\texttt{bvar}\rangle)\texttt{)}$
$E(\langle\texttt{pat1}\rangle\texttt{=}\langle\texttt{exp1}\rangle, k) = E(\langle\texttt{exp1}\rangle, \texttt{(}\lambda\ \texttt{(v1)}\ \texttt{(}P(\langle\texttt{pat1}\rangle, \texttt{(}k\ \texttt{v1)}, \texttt{(erl-exit-badmatch)})\ \texttt{v1))})$
$E(\langle\texttt{exp1}\rangle\texttt{,}\langle\texttt{exp2}\rangle, k) = E(\langle\texttt{exp1}\rangle, \texttt{(}\lambda\ \texttt{(v1)}\ E(\langle\texttt{exp2}\rangle, k)\texttt{)})$
$E(\langle\texttt{exp1}\rangle\texttt{+}\langle\texttt{exp2}\rangle, k) = E(\langle\texttt{exp1}\rangle, \texttt{(}\lambda\ \texttt{(v1)}\ E(\langle\texttt{exp2}\rangle, \texttt{(}\lambda\ \texttt{(v2)}\ \texttt{(}k\ \texttt{(erl-add v1 v2))))})$
$E(\langle\texttt{fn}\rangle\texttt{(}\langle\texttt{exp1}\rangle\texttt{)}, k) = E(\langle\texttt{exp1}\rangle, \texttt{(}\lambda\ \texttt{(v1)}\ \texttt{(}k\ \texttt{(}N(\langle\texttt{fn}\rangle)\texttt{/1 v1)))})$

Pattern matching translation:

$P(\langle\texttt{ubvar}\rangle, s, f) = \texttt{(}\lambda\ \texttt{(}N(\langle\texttt{ubvar}\rangle)\texttt{)}\ s\texttt{)}$
$P(\langle\texttt{bvar}\rangle, s, f) = \texttt{(}\lambda\ \texttt{(v1)}\ \texttt{(if (erl-eq-object? v1}\ N(\langle\texttt{bvar}\rangle)\texttt{)}\ s\ f\texttt{))}$
$P(\texttt{[]}, s, f) = \texttt{(}\lambda\ \texttt{(v1)}\ \texttt{(if (erl-nil? v1)}\ s\ f\texttt{))}$
$P(\texttt{[}\langle\texttt{pat1}\rangle\texttt{|}\langle\texttt{pat2}\rangle\texttt{]}, s, f) = \texttt{(}\lambda\ \texttt{(v1)}$
$\qquad\texttt{(if (erl-cons? v1)}$
$\qquad\quad\texttt{(}P(\langle\texttt{pat1}\rangle, \texttt{(}P(\langle\texttt{pat2}\rangle, s, f)\ \texttt{(erl-tl v1))}, f)\ \texttt{(erl-hd v1))}$
$\qquad\quad f\texttt{))}$

Auxiliary functions:

$C(const)$: translate an Erlang constant to Scheme
$N(name)$: translate an Erlang variable or function name to Scheme

Note:

$\texttt{v}n$ stands for a freshly created variable which will not conflict with other variables.

Fig. 1. Simplified translation algorithm for a subset of Erlang.

8.3 Translation of Binding and Pattern Matching

To translate an Erlang binding operation to Scheme it is necessary to nest the evaluation of the "rest of the function clause" inside the binding construct. This can be achieved by performing a partial CPS conversion, as shown in Figure 1.

The translation function E has two parameters: the Erlang expression to translate (e) and a Scheme lambda-expression denoting the continuation which receives the result of the Erlang expression (k). E returns a Scheme expression.

E makes use of the function P to translate pattern matching. P's arguments are: the pattern to match and the success and failure Scheme expressions. P returns a one argument Scheme lambda-expression which pattern matches its argument to the pattern, and returns the value of the success expression if there is a match and returns the value of the failure expression otherwise.

When an Erlang function is translated, E is called on each function clause to translate the right hand side with the initial continuation $\texttt{(}\lambda\ \texttt{(x) x)}$ (i.e. the identity function). Note that the continuation k and all lambda-expressions generated in the translation are always inserted in the function position of a call.

This implies that in the resulting Scheme code all the lambda-expressions generated can be expressed with the `let` binding construct (except for those generated in the translation of functional objects, which is not shown). To correctly implement tail-calls, an additional translation rule is used to eliminate applications of the identity function, i.e. $(\texttt{(}\lambda\ \texttt{(x) x)}\ Y) \rightarrow Y$.

The translation algorithm is not a traditional CPS conversion because function calls remain in direct style (i.e. translated Erlang functions do not take an additional continuation argument). This partial CPS conversion is only used to translate Erlang binding to Scheme binding. A remarkable property of function E is that it embeds k in the scope of all Scheme bindings generated in the translation of e. This is important because k may have free variables which must resolve to variables bound in e. This is achieved by inserting k inside $E(e, k)$ at a place where the variables are in scope. Similarly, P always embeds s (the success expression) in the scope of all Scheme bindings generated. This is useful to handle expressions such as `Z=(X=1)+(Y=2), X+Y+Z` which reference variables bound inside previous expressions (here `X` and `Y`).

Now consider the Erlang expression `[X|Y]=foo:f(A), X+bar:g(Y)`. This is translated to the following Scheme expression (if we assume that `A` is bound):

```
(let ((v7 ^a))
  (let ((v5 (foo:f/1 v7)))
    (let ((v6 v5))
      (if (erl-cons? v6)
          (let ((^x (erl-hd v6)))
            (let ((^y (erl-tl v6)))
              (let ((v1 v5))
                (let ((v2 ^x))
                  (let ((v4 ^y))
                    (let ((v3 (bar:g/1 v4)))
                      (erl-add v2 v3)))))))
          (erl-exit-badmatch)))))
```

There are many useless bindings in this code. In the actual implementation, the translator keeps track of constants, bound variables and singly referenced expressions and propagates them to avoid useless bindings. With this improvement the code generated is close to what we would expect a programmer to write:

```
(let ((v5 (foo:f/1 ^a)))
  (if (erl-cons? v5)
      (erl-add (erl-hd v5) (bar:g/1 (erl-tl v5)))
      (erl-exit-badmatch))))))
```

8.4 Translation of Conditionals

The `case`, `if`, and `receive` constructs perform conditional evaluation. The `case` construct evaluates an expression and finds the first of a set of patterns which matches the result, and then evaluates the expression associated with that pattern. The `receive` construct is like `case` except that the object to be matched is implicit (the next message in the process' mailbox), no error is signaled if no pattern matches (it simply moves to the following message in a loop until a match is found), and a timeout can be specified. The `if` construct is a degenerate `case` where each clause is only controlled by a boolean guard (no pattern matching is done).

These conditional constructs must be handled carefully to avoid code duplication. Consider this compound Erlang expression containing `X*Y` preceded by a `case` expression: `case X of 1->Y=X*2; Z->Y=X+1 end, X*Y`.

This `case` expression will select one of the two bindings of `Y` based on the value of `X`. After the `case`, `Y` is a bound variable that can be referenced freely. On the other hand `Z` is not accessible after the `case` because it does not receive a value in all clauses of the `case` (it is an "unsafe" variable after the `case`).

The `case` construct could be implemented by adding to the translation function E a rule like Figure 2a. Note that the continuation k is inserted once in the generated code for each clause of the `case`. This leads to code duplication which is a problem if the `case` is not the last expression in the function body and the `case` has more than one clause. If the function body is a sequence of n binary `case` expressions, some of the code will be duplicated 2^n times.

This code explosion can be avoided by factoring the continuation so that it appears only once in the generated code. A translation rule like Figure 2b would almost work. The reason it is incorrect is that k is no longer nested in the scope of the binding constructs generated for the `case` clauses, so the bindings they introduce are not visible in k.

A correct implementation has to transfer these bindings to k. This can be done by a partial lambda-lifting of k as shown in Figure 2c. The arguments of the lambda-lifted k (i.e. `vk`) are the result of the `case` (i.e. `vr`) and the set of bound variables that are added by the clauses of the `case` and referenced in k (i.e. AV). Each clause of the case simply propagates these bindings to `vk`. AV can be computed easily from the free variables (it is the difference between the set of free variables after the `case` and the set of free variables after the selector expression). The lambda-lifting is partial because `vk` may still have free variables after the transformation.

This lambda-lifting could be avoided by using assignment. Dummy bindings to the variables AV would be introduced just before the first pattern matching operation. Assignment would be used to set the value of these variables in the clauses of the `case`. This solution was rejected because many Scheme systems treat assignment less efficiently than binding (due to generational GC and the assignment conversion traditionally performed to implement `call/cc` correctly).

In the actual implementation of the pattern matching constructs, the patterns are analyzed to detect common tests and factor them out so that they are only executed once (using a top-down, left-right pattern matching technique similar to [4]). For example the translation of the following `case` expression will only contain one test that `X` is a pair:

```
case X of  [1|Y]->...;  [2|Z]->...  end
```

```
E( case ⟨exp0⟩ of      , k) = E(⟨exp0⟩, (λ (v0)
    ⟨pat1⟩ -> ⟨exp1⟩;                (P(⟨pat1⟩,
    ⟨pat2⟩ -> ⟨exp2⟩                    E(⟨exp1⟩, k),   ;;; duplication of k
   end                                  (P(⟨pat2⟩,
                                           E(⟨exp2⟩, k),   ;;; duplication of k
                                           (erl-exit-case-clause))
                                         v0))
                                      v0)))
```

a) Inefficient translation of the case construct.

```
E( case ⟨exp0⟩ of      , k) = E(⟨exp0⟩, (λ (v0)
    ⟨pat1⟩ -> ⟨exp1⟩;                (let ((vk k))   ;;; k not in right scope
    ⟨pat2⟩ -> ⟨exp2⟩                   (P(⟨pat1⟩,
   end                                    E(⟨exp1⟩, vk),
                                          (P(⟨pat2⟩,
                                             E(⟨exp2⟩, vk),
                                             (erl-exit-case-clause))
                                           v0))
                                        v0))))
```

b) Incorrect translation of the case construct.

```
E( case ⟨exp0⟩ of      , k) = E(⟨exp0⟩, (λ (v0)
    ⟨pat1⟩ -> ⟨exp1⟩;                (let ((vk (λ (vr AV...) (k vr))))
    ⟨pat2⟩ -> ⟨exp2⟩                   (P(⟨pat1⟩,
   end                                    E(⟨exp1⟩, (λ (vr) (vk vr AV...))),
                                          (P(⟨pat2⟩,
                                             E(⟨exp2⟩, (λ (vr) (vk vr AV...))),
                                             (erl-exit-case-clause))
                                           v0))
                                        v0))))
```

Where AV... is the set of bound variables that are added by the clauses of the `case` and referenced in k.

c) Correct translation of the case construct.

Fig. 2. Translation of the `case` construct.

9 Errors and `catch/throw`

The traditional way of performing non-local exits in Scheme is to use first-class continuations. A `catch` is translated to a call to Scheme's `call/cc` procedure which captures the current continuation. This "escape" continuation is stored in the process descriptor after saving the current escape continuation for when the `catch` returns. A `throw` simply calls the current escape continuation with its argument. When control resumes at a `catch` (either because of a normal return or a `throw`), the saved escape continuation is restored in the process descriptor.

10 Concurrency

First-class continuations are also used to implement concurrency. The state of a process is maintained in a process descriptor. Suspending a process is done by calling `call/cc` to capture its current continuation and storing this continuation in the process descriptor. By simply calling a suspended process' continuation, the process will resume execution.

Three queues of processes are maintained by the runtime system: the ready queue (processes that are runnable), the waiting queue (processes that are hung at a `receive`, waiting for a new message to arrive in their mailbox), and the timeout queue (processes which are hung at a `receive` with timeout). The timeout queue is a priority queue, ordered on the time of timeout, so that timeouts can be processed efficiently.

There is no standard way in Scheme to deal with time and timer interrupts. To simulate preemptive scheduling the runtime system keeps track of the function calls and causes a context switch every so many calls. When using the Gambit-C Scheme system, which has primitives to install timer interrupt handlers, a context switch occurs at the end of the time slice, which is currently set to 100 msecs.

11 Performance

11.1 Benchmark Programs

To measure the performance of our compiler we have used mostly benchmark programs from other Erlang compilers. We have added two benchmarks (`ring` and `stable`) to measure the performance of messaging and processes.

- `barnes` (10 repetitions): Simulates gravitational force between 1000 bodies.
- `fib` (50 repetitions): Recursive computation of 30th Fibonacci number.
- `huff` (5000 repetitions): Compresses and uncompresses a 38 byte string with the Huffman encoder.
- `length` (100000 repetitions): Tail recursive function that returns the length of a 2000 element list.
- `nrev` (20000 repetitions): Naive reverse of a 100 element list.
- `pseudoknot` (3 repetitions): Floating-point intensive application taken from molecular biology [13].
- `qsort` (50000 repetitions): Sorts 50 integers using the Quicksort algorithm.
- `ring` (100 repetitions): Creates a ring of 10 processes which pass around a token 100000 times.
- `smith` (30 repetitions): Matches a DNA sequence of length 32 to 100 other sequences of length 32. Uses the Smith-Waterman algorithm.

– `stable` (5000 repetitions): Solves the stable marriage problem concurrently with 10 men and 10 women. Creates 20 processes which send messages in fairly random patterns.
– `tak` (1000 repetitions): Recursive integer arithmetic Takeuchi function. Calculates `tak(18,12,6)`.

11.2 Erlang Compilers

Etos was coupled with the Gambit-C Scheme compiler version 2.7a [10]. We will first briefly describe the Gambit-C compiler.

The Gambit programming system combines an interpreter and a compiler fully compliant to R^{4}RS and IEEE specifications. The Gambit-C compiler translates Scheme programs to portable C code which can run on a wide variety of platforms. Gambit-C also supports some extensions to the Scheme standard such as an interface to C which allows Scheme code to call C routines and vice versa.

The Gambit-C compiler performs many optimizations, including automatic inlining of user procedures, allocation coalescing, and unboxing of temporary floating point results. The compiler also emits instructions in the generated code to check for stack overflows and external events such as user or timer interrupts. The time between each check is bound by a small constant, which is useful to guarantee prompt handling of interprocess messages.

Gambit-C includes a memory management system based on a stop and copy garbage collector which grows and shrinks the heap as the demands of the programs change. The user can force a minimum and/or maximum heap size with a command line argument. Scheme objects are encoded in a machine word (usually 32 bits), where the lower two bits are the primary type tag. All heap allocated objects are prefixed with a header which gives the length and secondary type information of the object. Characters and strings are represented using the Unicode character set (i.e. 16 bit characters). Floating point numbers are boxed and have 64 bits of precision (like the other Erlang compilers used).

The implementation of continuations uses an efficient lazy copying strategy similar to [15] but of a finer granularity. Continuation frames are allocated in a small area called the "stack cache". This area is managed like a stack (i.e. LIFO allocation) except when the `call/cc` procedure is called. All frames in the stack cache upon entry to `call/cc` can no longer be deallocated. When control returns to such a frame, it is copied to the top of the stack cache. Finally, when the stack cache overflows (because of repeated calls to `call/cc` or because of a deep recursion), the garbage collector is called to move all reachable frames from the stack cache to the heap.

We have compared Etos version 1.4 [9] with three other Erlang compilers:

– Hipe version 0.27 [17], an extension of the JAM bytecode compiler that selectively compiles bytecodes to native code;
– BEAM/C version 4.5.2 [14], compiles Erlang code to C using a register machine as intermediate;
– JAM version 4.4.1 [2], a bytecode compiler for a stack machine.

11.3 Execution Time

The measurements were made on a Sun UltraSparc 143 MHz with 122 Mb of memory. Each benchmark program was run 5 times and the average was taken after removing the best and worse times.

The Scheme code generated by Etos is compiled with Gambit-C 2.7a and the resulting C code is then compiled with `gcc` 2.7.2 using the option `-O1`. The executable binary sets a fixed 10 Mb heap.

Program	Etos (secs)	Time relative to Etos		
		Hipe	BEAM/C	JAM
`fib`	31.50	1.15	1.98	8.33
`huff`	9.74	1.48	5.01	24.81
`length`	11.56	2.07	3.44	34.48
`smith`	10.79	2.17	3.37	13.06
`tak`	13.26	1.12	4.37	11.09
`barnes`	9.18	2.08	–	4.07
`pseudoknot`	16.75	2.37	–	3.18
`nrev`	22.10	.84	1.83	10.98
`qsort`	14.97	.96	3.88	15.38
`ring`	129.68	.30	.31	1.92
`stable`	21.27	1.16	.64	2.43

Fig. 3. Execution time of benchmarks

The results are given in Figure 3. They show that Etos outperforms the other Erlang compilers on most benchmarks. If we subdivide the benchmarks according to the language features they stress, we can explain the results further:

- `fib`, `huff`, `length`, `smith` and `tak`, which are integer intensive programs, take advantage of the efficient treatment of fixnum arithmetic in Gambit-C and from the inlining of functions. Etos is up to two times faster than Hipe, 5 times faster than BEAM/C, and 35 times faster than JAM.
- On the floating point number benchmarks, `barnes` and `pseudoknot`, Etos is also faster than the other Erlang implementations. In this case Etos is a little over two times faster than Hipe. These programs crashed when compiled with BEAM/C.
- List processing is represented by `nrev` and `qsort`. On these programs Hipe is a little faster than Etos (4% to 16%), which is still roughly two to four times faster than BEAM/C. Etos' poor performance is only partly attributable to its implementation of lists:
  1. Gambit-C represents lists using 3 word long pairs as opposed to 2 words on the other systems. Allocation is longer and the GC has more data to copy.

2. Gambit-C guarantees that interrupts are checked at bound intervals [11] which is not the case for the other systems. For example, the code generated by Gambit-C for the function `app` (the most time consuming function of the `nrev` benchmark) tests interrupts twice as often as Hipe (i.e. on function entry and return).
3. The technique used by Gambit-C to implement proper tail-recursion in C imposes an overhead on function returns as well as calls between modules. For `nrev` the overhead is high because most of the time is spent in a tight non-tail recursive function. Independent experiments [12] have shown that this kind of program can be sped up by a factor of two to four when native code is generated instead of C.

– Finally `ring` and `stable` manipulate processes. Here we see a divergence in the results. Hipe is roughly three times faster than Etos on `ring`. Etos performs slightly better than Hipe on `stable` but is not as fast as BEAM/C. We suspect that our particular way of using `call/cc` to implement processes (and not the underlying `call/cc` mechanism) is the main reason for Etos' poor performance:
  1. When a process' mailbox is empty, a `receive` must call the runtime library which then calls `call/cc` to suspend the process. These inter-module calls are rather expensive in Gambit-C. It would be better to inline the `receive` and `call/cc`.
  2. Scheme's interface to `call/cc` (which receives a closure, which will typically have to be allocated, and must allocate a closure to represent the continuation) adds considerable overhead to the underlying `call/cc` mechanism which requires only a few instructions.

12 Future Work

Etos 1.4 does not implement Erlang fully. Most notably, the following features of Erlang are not implemented:

1. Macros, records, ports, and binaries.
2. Process registry and dictionary.
3. Dynamic code loading.
4. Several built-in functions and libraries.
5. Distribution (all Erlang processes must be running in a single user process).

We plan to add these features and update the compiler so that it conforms to the upcoming Erlang 5.0 specification.

An interesting extension to Etos is to add library functions to access Gambit-C's C-interface from Erlang code. Interfacing Erlang, Scheme and C code will then be easy.

The Gambit-C side of the compilation can also be improved. In certain cases the Scheme code generated by Etos could be compiled better by Gambit-C (its optimizations were tuned to the style of code Scheme programmers tend to write). It is worth considering new optimizations and extensions specifically designed for Etos's output. In particular, a more efficient interface to `call/cc` will be designed. Moreover we think the performance of Etos will improve by a factor of two on average when we start using a native code back-end for Gambit. We are also working on a hard real-time garbage collector and a generational collector to improve the response time for real-time applications.

13 Conclusions

It is unfortunate that inessential mismatches between Erlang and Scheme cause many small difficulties in the translation of Erlang to standard Scheme. Specifically the translation would be easier and more efficient if Scheme: was case-sensitive, did not separate the numeric class (integer, ...) and exactness of numbers, allowed testing the arity of procedures or a way to trap arity exceptions, had the ability to define new data types, had a raw binary array data type, a foreign function interface and a more efficient interface to `call/cc`. Fortunately, mature implementations of Scheme, and Gambit-C in particular, already include many of these extensions to standard Scheme so in practice it is not a big problem because such features can be accessed on an implementation specific basis by using a file of macros tailored to the Scheme implementation.

When coupled with Gambit-C, the Etos compiler shows promising results. It performs very well on integer and floating point arithmetic, beating all other currently available implementations of Erlang. Its performance on list processing and process management is not as good but we think this can be improved in a number of ways. This is a remarkable achievement given that the front-end of the compiler was implemented in less than a month by a single person. It shows that it is possible to quickly reuse existing compiler technology to build a new compiler and that a compiler with a deep translation pipeline (i.e. Erlang to Scheme to C to machine code) need not be inefficient. Of course Etos' success is to a great extent due to the fact that Scheme and Erlang offer very similar features (data types, functional style, dynamic typing) and that Scheme's `call/cc` can be used to simulate Erlang's escape methods and concurrency. Our work shows that Scheme is well suited as a target for compiling Erlang.

Acknowledgements

This work was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada and the Fonds pour la formation de chercheurs et l'aide à la recherche.

References

1. J. L. Armstrong. The development of erlang. In *Proceedings of the International Conference on Functional Programming*, pages 196–203, Amsterdam, June 1997.

2. J. L. Armstrong, B. O. Däcker, S. R. Virding, and M. C. Williams. Implementing a functional language for highly parallel real-time applications. In *Proceedings of Software Engineering for Telecommunication Switching Systems*, Florence, April 1992.
3. J. L. Armstrong, S. R. Virding, C. Wikström, and M. C. Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.
4. Lennart Augustsson. Compiling Pattern Matching. In Jean-Pierre Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy, France, LNCS 201*, pages 368–381. Springer Verlag, 1985.
5. D. Boucher. Lalr-scm. Available at `ftp.iro.umontreal.ca` in `pub/parallele/boucherd`.
6. W. Clinger. Proper Tail Recursion and Space Efficiency. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 174–185, Montréal, June 1998. ACM Press.
7. W. Clinger and J. Rees [editors]. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.
8. D. Dubé. SILex, user manual. Available at `ftp.iro.umontreal.ca` in `pub/parallele`.
9. M. Feeley. Etos version 1.4. Compiler available at `ftp.iro.umontreal.ca` in `pub/parallele/etos/etos-1.4`.
10. M. Feeley. Gambit-C version 2.7a, user manual. Compiler available at `ftp.iro.umontreal.ca` in `pub/parallele/gambit/gambit-2.7`.
11. M. Feeley. Polling efficiently on stock hardware. In *Proceedings of the Functional Programming and Computer Architecture Conference*, pages 179–187, Copenhagen, June 1993.
12. M. Feeley, J. Miller, G. Rozas, and J. Wilson. Compiling Higher-Order Languages into Fully Tail-Recursive Portable C. Technical Report 1078, Département d'informatique et de recherche opérationelle, Université de Montréal, 1997.
13. P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. Van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, J. Kamperman, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.
14. B. Hausman. Turbo Erlang: approaching the speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.
15. R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. *ACM SIGPLAN Notices*, 25(6):66–77, 1990.
16. IEEE Standard for the Scheme Programming Language. IEEE Standard 1178-1990, IEEE, New York, 1991.
17. E. Johansson, C. Jonsson, T. Lindgren, J. Bevemyr, and H. Millroth. A pragmatic approach to compilation of Erlang. UPMAIL Technical Report 136, Uppsala University, Sweden, July 1997.
18. The Internet Scheme Repository. `http://www.cs.indiana.edu/scheme-repository`.
19. Gerald Jay Sussman and Guy Lewis Steele Jr. SCHEME, an interpreter for extended lambda calculus. AI Memo 349, Mass. Inst. of Technology, Artificial Intelligence Laboratory, Cambridge, Mass., December 1975.