Status report on the ETOS Erlang to Scheme compiler

http://www.iro.umontreal.ca/~etos

http://www.iro.umontreal.ca/~gambit

http://www.iro.umontreal.ca/~feeley

Marc Feeley, Patrick Piché, Sylvain Beaulieu,
Martin Larose, Mario Latendresse

Université de Montréal

ETOS goals

- **Conform to Standard Erlang (final draft 0.6, june 1998)**
- **Generate fast code**
- **Reuse Gambit-C Scheme compiler technology**
  - Mature compiler
    - dynamic module loading
    - FFI (Foreign Function Interface)
    - Unicode support
  - Generates fast portable ANSI-C code (Unix, Windows, Mac)
  - Specially modified for ETOS

Summary

- **Performance comparison**
- **Gambit-C optimisations**
- **Pattern matcher**
- **Future work**

ETOS 2.3 compared to 1.4 and others

- **ETOS 2.3 + Gambit-C 3.1f + gcc 2.8.1**
- **JAM/BEAM 47.4.1, Hipe 0.27**

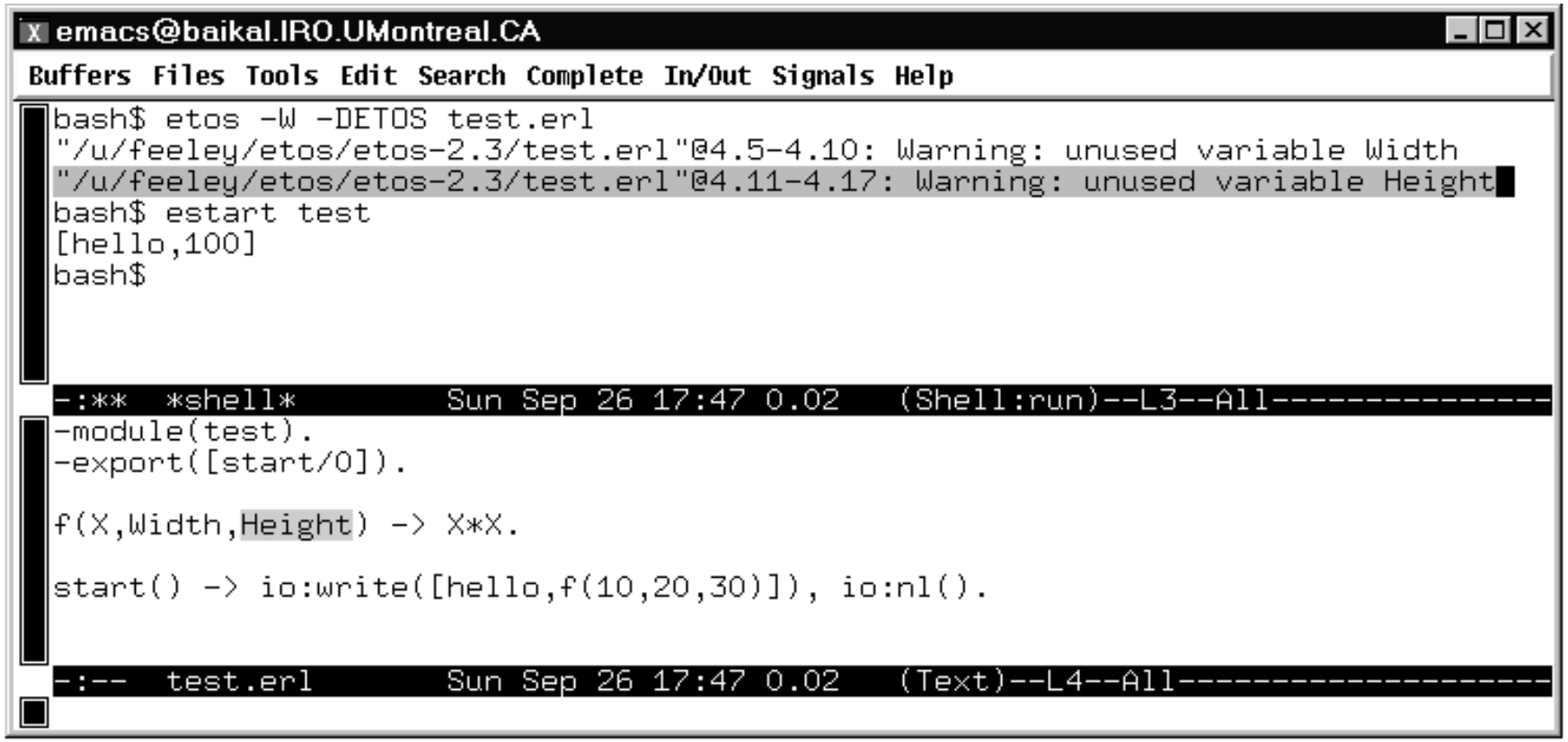
	Run time relative to ETOS			
	UltraSparc 143MHz			Pentium 400
	Hipe	BEAM	Jam	BEAM
fib	1.65	3.27	7.24	3.27
huff	1.35	4.19	12.48	6.28
length	1.55	4.31	15.66	3.61
smith	2.20	3.56	7.55	7.38
tak	1.56	6.29	8.94	7.89
barnes	16.01	20.62	18.18	19.62
nrev	.89	1.47	5.13	2.15
qsort	1.11	5.21	11.10	7.07
ring	.57	.76	1.15	.70
stable	.91	.61	1.15	.54

- **In general, v2.3 faster than v1.4 (better inlining & better intermodule calls by Gambit-C)**
- **Slower in a few cases (support for dynamic module loading causes more intermodule calls)**
- **Processes are now better (still not great due to remaining intermodule calls to kernel)**

What is Standard Erlang?

- **A "Cleaner" Erlang:**
  - Character type and support for Unicode
  - New constructs: **`all_true`**, **`some_true`** and **`try`**
  - Order of evaluation = left to right
  - Recognizer BIFs begin with "**`is_`**" prefix
  - Function type

Using the ETOS compiler



Compilation of an Erlang module (part 1)

Erlang source "`m.erl`":

```
-module(m).
-export([f/1]).
f(X) -> other:h(g(X)).
g(Y) -> Y+1.
```

Scheme code generated by ETOS front-end:

```
(let ()
  (include "/u/feeley/etos/etos-2.3/rt-gambit.scm")
  (letrec ((module_info/1 (lambda (v0) ...))
           (module_info/0 (lambda () ...))
           (f/1 (lambda (v1) (other:h/1 (g/1 v1))))
           (g/1 (lambda (v2) (erl-+/2 v2 1))))
    (erl-install-module
     "m"
     (vector (vector "f" 1 f/1)
             (vector "module_info" 0 module_info/0)
             (vector "module_info" 1 module_info/1))
     (lambda ()
       (set! m:f/1 f/1)
       (set! m:module_info/0 module_info/0)
       (set! m:module_info/1 module_info/1)
       (if (erl-fn-unbound? "other" "h" 1)
           (set! other:h/1 (erl-undefined-fn "other" "h"))))
     (lambda ()
       (set! m:f/1 (erl-undefined-fn "m" "f"))
       (set! m:module_info/0 (erl-undefined-fn "m" "module_info"))
       (set! m:module_info/1 (erl-undefined-fn "m" "module_info"))))))
```

Exported function table

Module installation

Module deinstallation

Compilation of an Erlang module (part 2)

Gambit-C expands this program to:

```
(let ((module_info/1 (lambda (v0) ...))
      (module_info/0 (lambda () ...))
      (f/1 (lambda (v1)
             (other:h/1
              (if (and (##fixnum? v1)
                       (##fixnum.< v1 (##fixnum.+ v1 1)))
                  (##fixnum.+ v1 1)
                  (erl-generic-+/2 v1 1))))))
  (erl-install-module
   "m"
   (vector (vector "f" 1 f/1)
           (vector "module_info" 0 module_info/0)
           (vector "module_info" 1 module_info/1))
   (lambda ()
     (set! m:f/1 f/1)
     (set! m:module_info/0 module_info/0)
     (set! m:module_info/1 module_info/1)
     (if (##unbound? other:h/1)
         (set! other:h/1
               (lambda x (erl-undefined-fn-handler x 'other 'h)))))
   (lambda ()
     ...)))
```

- **`erl-+/2`** inlined
- **`g/2`** inlined and removed
- This is compiled to C code (not shown!)
- **`gcc`** compiles this to the shared library "**`m.o1`**"

Intermodule jumps in C

- **Both Erlang and Scheme require constant space tail-calls**
- **Jumping from point "A" to "B" is a problem because A and B are possibly in different C functions (given that modules are compiled separately)**
- **Gambit-C uses a trampoline technique**
  - Each control point is represented by a structure containing a pointer to the C function hosting that control point
  - A dispatcher function passes control from one host C function to the next

Simplified example

```
/*-------------------- runtime library --------------------*/
ctl_pt *ret;        /* return address */
ctl_pt *erlang_add; /* initialized to &tbl_erlang[1] */
ctl_pt *m_f;        /* initialized to &tbl_m[0] */
ctl_pt *other_h;    /* initialized to &tbl_other[1] */

main () { ... dispatcher (m_f); } /* call m:f/1 */

void dispatcher (ctl_pt *pc) { while (1) pc = pc->host (pc); }

/*---------------------- module "m" ----------------------*/
ctl_pt tbl_m[] = { {host_m}, {host_m} };

ctl_pt *host_m (ctl_pt *pc)
{ jump:
  switch (pc - tbl_m) {
    case 0: ... ret = &tbl_m[1]; pc = erlang_add; goto jump;
    case 1: ... pc = other_h; goto jump;
  }
  return pc;
}

/*-------------------- module "erlang" --------------------*/
ctl_pt tbl_erlang[] = { {host_erlang}, {host_erlang} };

ctl_pt *host_erlang (ctl_pt *pc)
{ jump:
  switch (pc - tbl_erlang) {
    case 0: ...
    case 1: ... pc = ret; goto jump;
  }
  return pc;
}
```

Example’s details

- **Remote computed jumps are slow (1 fn return, 1 fn call, 2 `switch`, 1 `goto`)**
- **Local computed jumps are faster (1 `switch`, 1 `goto`)**
- **Local direct jumps are fastest (1 `goto`)**
- **Virtual machine registers are cached in local variables for fast access (read on entry, write on exit), which makes remote jumps even slower**

Specialized code for gcc

module "m"

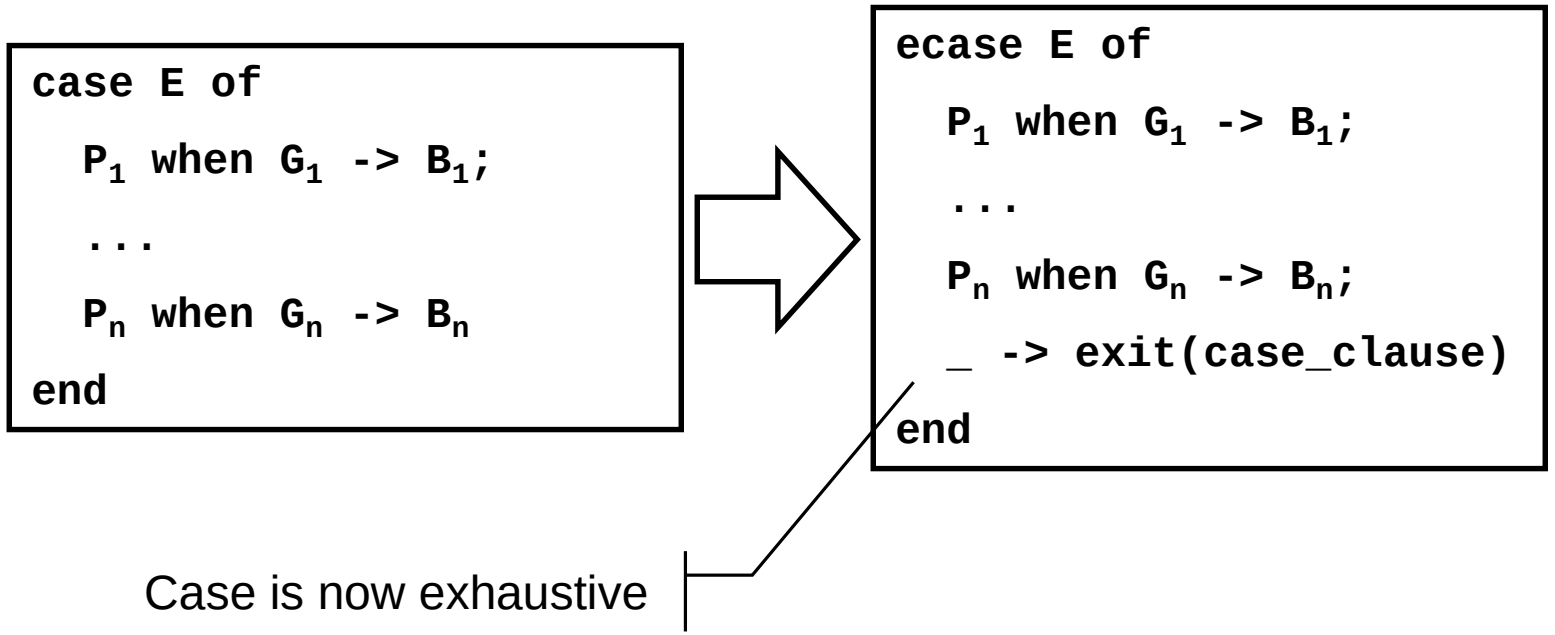
```
ctl_pt tbl_m[] =
{ {host_m,0}, {host_m,0} }; /* label field initialized later */

ctl_pt *host_m (ctl_pt *pc)
{ static void *lbls[] = { &&label0, &&label1 };
  if (pc == NULL) return lbls; /* to initialize label field */
  goto *pc->label;
  label0: ... ret = &tbl_m[1]; pc = erlang_add;
          if (pc->host != host_m) return pc;
          goto *pc->label;
  label1: ...
  }
}
```

- **gcc's "computed `goto`" can replace the `switch` statements**
- **Remote computed jump is 2 times faster**
- **Local computed jump is 3 times faster**
- **gcc's computed goto only works within one function (Mercury beware!)**

Pattern matching based on exhaustive case (ecase)

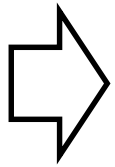
- **A new pattern matching construct**



Function definition

- **Function definition with ecase**

```
fun (P11,..,P1n) when G1 -> B1;
    ..
    (Pm1,..,Pmn) when Gm -> Bm
end
```

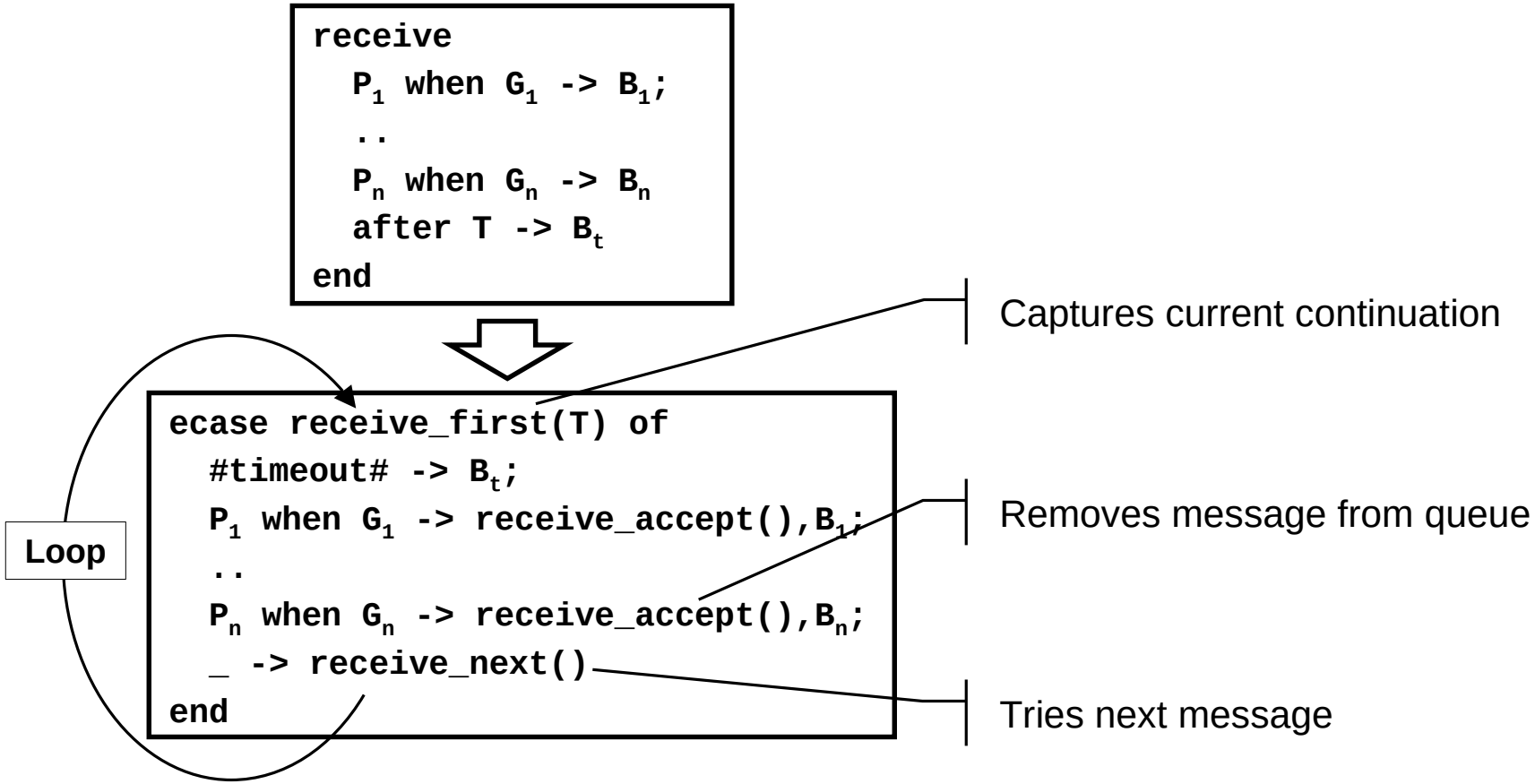


```
fun(A1,..,An) ->
  ecase {A1,..,An} of
    {P11,..,P1n} when G1 -> B1;
    ..
    {Pm1,..,Pmn} when Gm -> Bm;
    {_,..,_} -> exit(lambda_clause)
  end
end
```

Exhaustive case allows to skip artificial structure tests

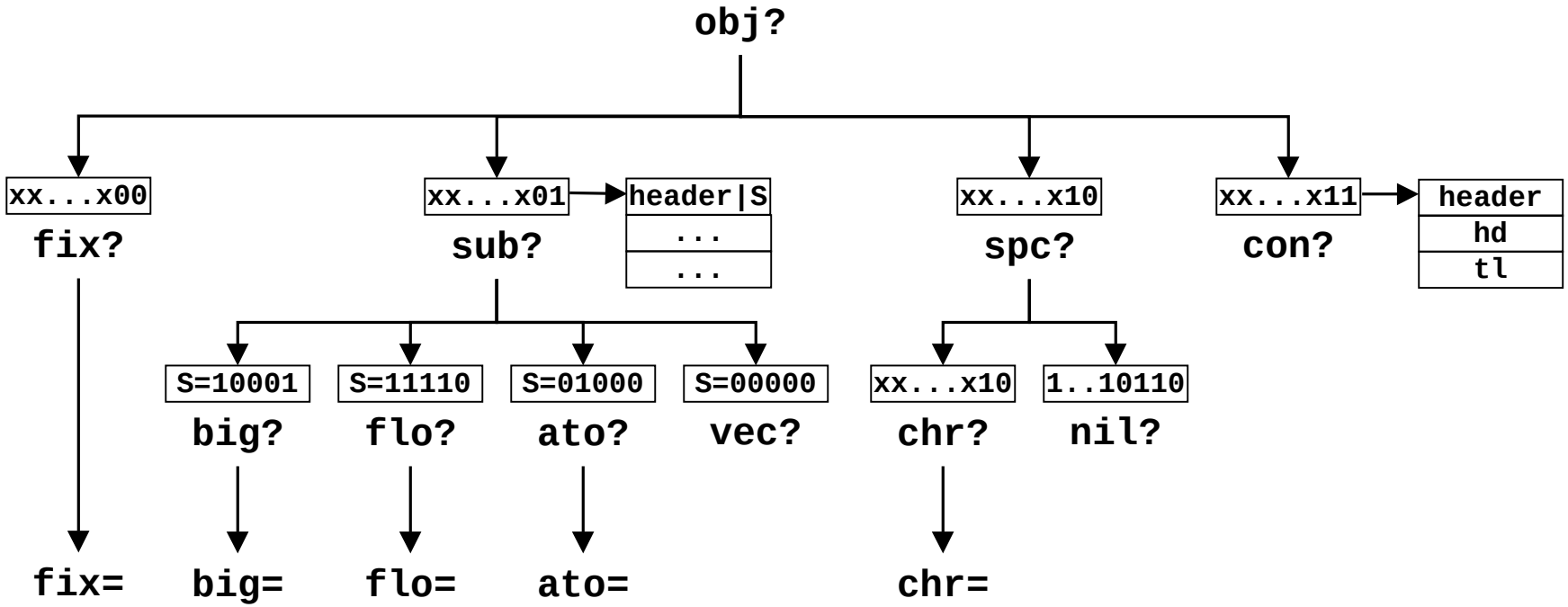
Implementation of receive

- **call/cc mechanism allows simple transformation of receive into ecase**



Type test expressions

- **Type test tree based on Gambit-C data representation**



Test operators

- **Operators working on tests**
  - **Boolean**
    - **all, and, one, or, not, true, false**
  - **Predictive**
    - **kt(T)** **{T known to be true}**
    - **kf(T)** **{T known to be false}**

Test prediction

- **Relationships in data type hierachy helps predict tests**

```
PREDICT(TEST) =>
if(kt(<one child>)) TRUE
else
  if(!kt(<parent>))
    if(kf(<parent>)) FALSE else TEST
  else
    if(kt(<one brother>)) FALSE
    else
      if(kf(<one brother>)) TRUE else TEST
```

Example

```
ecase X of
  [$a|$b] -> 1;
  [H|_] when is_char(H) -> 2;
  _ -> 3
end
```

```
(if (con? ~x)
    (let ((h (erl-hd ~x))
      (if (chr= h #\a)
          (if (chr= (erl-tl ~x) #\b)
              1
              2)
          (if (chr? h) 2 3)))
    3)
```

chr? predicted to be true
(the child test **chr=** is known to be true)

Sestoft algorithm

- **Perform tests left to right, top down**
- **Accumulate positive and negative information**
- **Predict tests using this information**

Baudinet-MacQueen algorithm

- **Heuristic approach to choose test order**

- **Suggests three heuristics:**
  - relevance
    - First perform tests that discriminate first clause
  - branching factor
    - First perform tests that discriminate most clauses
    - Note: Modified for use with dynamic types
  - arity factor
    - First perform tests that span less new tests

ETOS pattern matcher

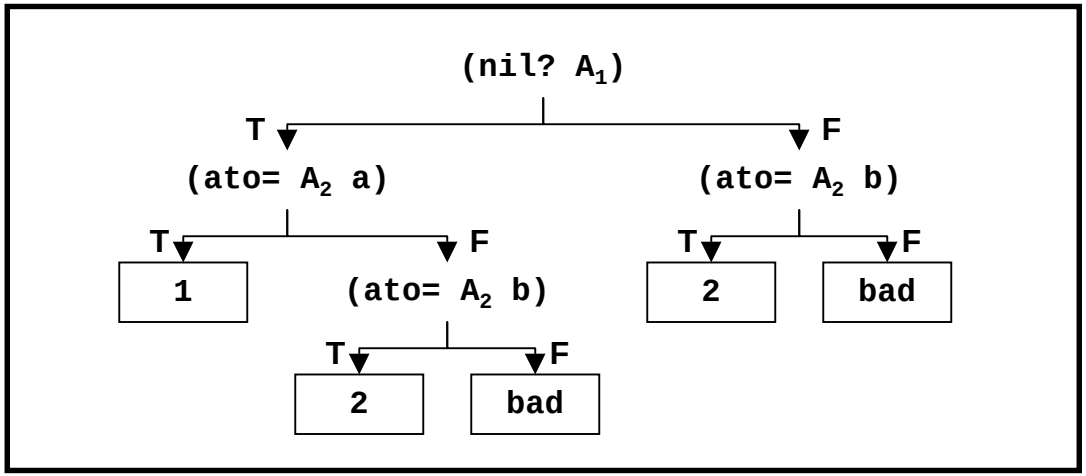
- **Currently, Sestoft algorithm implemented and tested**
- **Implementation suited for easy upgrade to MacQueen**
- **Work currently in progress:**
  - Development of better heuristics (example follows)
  - Use of hash tables for selection in vast atomic clauses

Pattern matching example

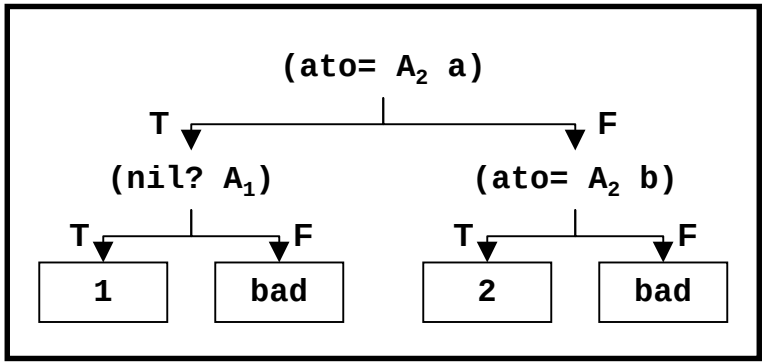
Erlang expression

```
ecase {A1,A2} of
  {[],a} -> 1;
  {_,b} -> 2;
  {_,_} -> bad
end.
```

Sestoft and Baudinet-MacQueen test tree



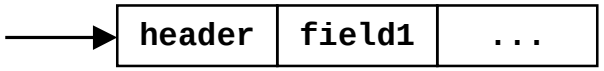
Optimal test tree

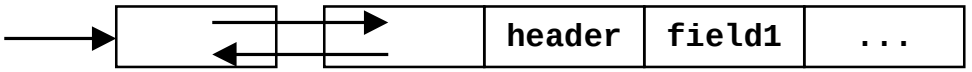


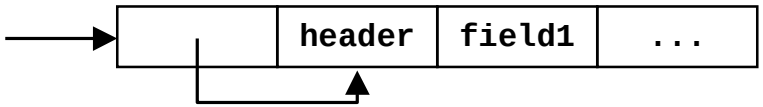
- Sestoft: Left to right, top down
- BM: Unable to choose
- New heuristic: 2nd clause relevance

Future: real-time GC

- **Experimented with various GC algorithms**
  - Stop & Copy
    - Gambit-C ’s standard GC
    - 2 semispaces
    - "direct" access to objects
  - Mark & Compact
    - using non-movable handles
  - Mark & Compact Real-Time
    - handles allow fast relocation
  - Brooks' Real-Time
    - movable handle
    - 2 semispaces
  - Brooks' Real-Time + Generational
    - In design







Real-time GC overhead (Gambit-C 2.7)

	Alloc MB/s	Time secs	M&C	M&C R-T	Brooks R-T
boyer	3.14	16.46	.91	.87	1.67
compiler	.91	49.99	1.19	1.49	1.41
puzzle	.92	21.88	1.20	1.31	1.25
browse	3.95	33.36	1.25	1.70	1.48
conform	2.88	25.52	1.29	1.81	1.72
traverse	5.61	10.93	1.37	1.68	1.62
peval	5.86	35.40	1.52	1.81	1.68
fft	12.58	5.05	1.53	1.56	1.02
maze	16.18	11.58	1.65	2.43	2.34
simplex	15.34	10.91	1.70	2.18	1.76
earley	8.42	36.13	1.87	2.22	1.52
dderiv	22.76	39.39	2.04	3.00	1.83
destruc	19.35	15.29	2.11	2.67	1.77
cpstak	46.94	13.95	2.41	2.44	1.53
fibfp	47.05	11.80	2.43	2.95	1.29
deriv	27.43	32.69	2.50	3.11	1.82
mbrot	60.20	13.03	2.87	3.64	1.33
divrec	54.92	16.66	3.15	3.88	1.49
sumfp	71.12	85.82	3.21	4.28	1.38
diviter	123.03	7.44	6.51	8.12	1.69

- M&C = 2.14 * S&C (on average)
- M&C R-T = 1.24 * M&C (on average)
- Brooks R-T = 1.58 * S&C (on average)

Real-time GC pauses

•M&C R-T on a 133Mhz DEC Alpha 21064/ Digital UNIX v4.0 (≈ 100MHz Pentium)

	Avg(ms)	Max(ms)	% GC
compiler	3.69	11	4
puzzle	2.12	8	4
conform	2.86	5	5
peval	2.67	5	8
boyer	2.83	6	9
browse	2.86	5	10
traverse	3.57	12	15
simplex	3.01	6	20
destruc	2.85	6	21
earley	3.47	10	22
dderiv	2.88	6	24
fft	3.31	6	24
deriv	2.85	6	29
maze	3.12	15	35
cpstak	2.39	5	40
divrec	2.89	6	45
diviter	2.89	6	48
fibfp	3.31	6	52
mbrot	3.35	7	55
sumfp	3.45	7	57

•Avg = 2.1 to 3.7 ms / Max = 5 to 15 ms

Future: native code generation

- **Pro: would allow faster intermodule calls**
- **con: low portability and need to redo C compiler's optimizations**
- **Approach**
  - use intermediate "RTL" code (RISC style register machine)
  - expand RTL instructions to native code
  - use a generic instruction scheduler
- **Easy to port to a new machine (≈ 2 weeks) and reasonable performance**

Gambit-RTL performance

- **Prototype implementation of Gambit-RTL compared to Gambit-C 3.0 (below 1 is good for Gambit-RTL)**

	Alpha	MIPS	PPC	USPARC	PII	M68K
tak	.56	.69	.91	.70	.33	.26
fib	.75	.71	.85	.46	.53	.22
mazefun	1.07	.66	1.53	.36	.37	.35
assq	1.10	.73	.85	.60	.29	.98
nrev	.81	.65	3.18	.81	.35	.41
sort	1.67	.80	3.27	.65	.44	.70
takl	1.53	.83	1.22	.92	.48	.49