Pattern Matching Compilation in L

Maxim Sokhatsky, Cloudozer LLP

07/07/2014

1 Overview

Here is the description of Patterm Matching Compilation scheme used in L. Obviously compiler should optimize matching tree for number of comparisons, the processing speed. Next, if possible, optimize the size of decision tree, the code size. It is well known that producing smallest decision tree is NP-hard. That is why was proposed a several heuristics:

- Relevance (Baudined, MacQueen, 1985)
- Arity Factor (Baudinet, MacQueen, 1985)
- Small Defaults (Baudinet, MacQueen, 1985)
- Fewer Child Rules (Maranget, 1992, Ramsey, Fernandez, 1995)
- Left to Right (Sestoft, 1996, Moscow ML, OCaml, MLKit)
- Small Branching Factor (SML/NJ)
- Large Branching Factor (Cardelli, 1984)
- Leaf Edges
- Artificial Rule
- Right to Left Top-down

In this document we will generalize all heuristics and also provide extra information about binary strings and named tuples pattern maching.

2 Normalization

Here is example of patterns set which should be matched along x with minimal set of atomic operations like SWITCH. You may specify patterns of any type. Consider the type of x an alberaic sum type of all patterns. Subsums of same type patterns is being processed separetly, producing different decision tree for each type.

```
case x of
    [1,2|tail] -> tail;
    [2,y] -> 2;
    [x,1,y|tail] -> 3;
    {{1,{_,2}},x,y} -> 4;
    {_,x,1,y} -> 5;
    {_,_} -> 6 * x + y;
    [{1,2,_},{x,y}] -> 7;
    _ -> :ok end.
```

First of all patterns should be filtered by the type of x. L compiler should compile only patterns that fit its type.

Table 1: Types Structure

Type	Tag
`(1,(2, T))`	C
`(2,(y, N))`	C
`(x,(1,(y, T)))`	C
`(1,(_, 2)),x,y)`	T3
`(_, x, 1, y)`	T4
`(_, _)`	T2
`((1,2,_),((x,y),N))`	T2

All cells in the matrix coule be **vars**, **wildcards**, **constants** or type **knots**. Type knots like `cons` or `tuple` encoded with type tag information, thus type knots is allowed to use in SWITCH along with constants. Constants are encoded with `int`, `atom`, `nil` and other value types.

Table 2: Encoded Input AST

Type Constructors	Tag
cons(int(1),cons(int(2),var(tail)))	C
cons(var(x),cons(var(y),nil()))	C
cons(var(x),cons(int(1),cons(var(y),var(tail))))	C

In runtime the value will holds the type index nunber, while the type information will be erased. E.g. some set of patterns may fall the same type, like in example assuming x is list of deep 2. The first four patterns has the same type `(_,(_,_))` – list of length two.

3 Decision Trees

After normalized M-arity matrix with N rules we can apply selected algorithm directly. We call decision tree efficient if it examines each matrix cell at most once. In this sense MacQueen (1985), Laville (1991), Maranget (1992), Sestoft (1996) are efficient; while Augustsson (1985), Wadler (1987), Maranget (1994) are not.

Usually each heuristic is dealing with only one column. Depending on the function result over column (e.g. equality), we can apply another heuristic on the same step, in other words we can compose heuristics.

Table 3: Heuristics

Heuristic	Occurance in Column
Constructors	N
Constants	N or OccuranceMap
Vars	-N
Wildcards	-N

As we can do SWITCH operation only over **constants** and type **knots** (encoded in memory with type tag) we take constants and knots number of occurance with positive sign. The **vars** and **wildcards** (which could be treated as vars that are not binded in matched expression) are counted with negated weight during heuristic decision.

In some cases base heuristics could be extended with additional information. E.g. you can observe not only the number of knots, but also count the sum of their arity.

```
case l of
    {1,2,3} -> 1
    {1,x,4} -> 2
    {1,x,5} -> 3,
         x  -> 4
```

Table 4: Matching Target AST Instructions

Instructions	Returns
bind(string,any)	void
tag(any)	int
element(int,tuple/N)	any
switch(any,tuple(any,list(tuple/2)),list(tuple/2))	tuple(any,list(tuple/2))
read(int,binary)	binary
scan(int,binary)	binary

Most Constants or Constructors Desision Tree:

```
switch(element(1,L),
    {4,[bind(x,L)]},
    [{1,switch(element(2,L),
        {4,[bind(x,L)]},
        [{3,switch(element(3,L),
            {4,[bind(x,L)]},
            [{2,{1,[]}}])},
         {4,{2,[bind(x,element(2,L))]}},
         {5,{2,[bind(x,element(2,L))]}}])}]).
```

Consider having simple case with vars and constants. We will use here only SWITCH target LLVM instruction and ELEMENT runtime tuple extraction.

Left to Right Desision Tree:

```
switch(element(1,L),
    {4,[bind(x,L)]},
    [{1,switch(element(2,L),
        switch(element(3,L),
            {4,[bind(x,L)]},
            [{4,{2,[bind(x,L)]}},
             {5,{3,[bind(x,L)]}}]),
        [{3,switch(element(3,L),
            {4,[bind(x,L)]},
            [{3,{1,[]}}])},
             {4,{2,[bind(x,element(2,L))]}},
             {5,{3,[bind(x,element(2,L))]}}]}])}]).
```

4 Binary Strings

Matching binary string with variable segment size is slightly different from matching algebraic data types.

```
case MPLS of
    <Value:20/bit,Exp:3/bit,S:1/bit,TTL:8/bit)> -> {Value,Exp,S,TTL} end.
```

```
case IPv6ExtHeader of
    <> end.
```

case Binary of <42:8, X1> -> 1; <Sz:8, V:Sz, X2> -> 2; <_:8, X3:16, Y:8> -> 3 end. "'
To unvail the power of binary segments pattern maching, we should extend the target language [SWITCH,ELEMENT,TAG,BIND] with [READ,SIZE,SCAN] operations. READ operation scans the bit string of variable size and SIZE operation is a guard that does range checks.

```
p1 = {size>=8, sw(42,read(0,8)), bind(X1,read(8,_))}
p2 = {size>=8, bind(Sz,read(0,8)), size>=8+Sz, sw(V,read(8,Sz)), bind(X2,read(8+Sz,_))}
p3 = {size>=8, bind(X3,read(8,24)), size>=32, bind(Y,read(24,32))}
```

One can extend the operation set with SCAN operation, which scans the bit string till a given byte or pattern. And you can also specify scan sections multiple times. The only rule is that rest/binary sections should be separated with constant bitstring.

```
case Scan of
    <_:8, X3:16, S1/binary, 0:8, S2/binary, 0:8, Rest/binary> -> 3 end.
```

Multiple scans binding is another possible evolution of bitstring pattern matching. Often used in video decoders, we need a case to fold several sequential null-terminated strings into one list of binded variables.

```
case MultipleScan of
    <_:8, X3:16, (Strings/scan, 0:8):N, Rest/binary> -> 3 end.
```

Here Strings will be binded with an list of null-terminated strings, and N will be binded with number of its occurance.

```
    [ io:format(”String ~p~n”,[lists:nth(1,Strings])
      || S <- lists:seq(1,N) ]
```

5 Named Tuples

Named Tuples could be easily transformed to normalized by constructing smallest union the tuple keys.

```
case NamedTuple of
    {n=7,a=5} -> 1;
    {a=5,s=0} -> 2;
    {s=0,n=1} -> 3 end.
```

Type	Flattened Matrix	Tag
(n=7,a=5,_)	[7 5 *]	1
(_, a=5,s=0)	[* 5 0]	1
(n=1,_, s=0)	[1 * 0]	1

6 List Matching Sample

The test coverage should include lists, deep lists and tuples custom heuristic cases. Here is example of deep list pattern matching.

```
caser x y ->
    case x of
        [1,2|tail] -> tail;
        [1,4]      -> 11;
        [2,3]   -> 20;
        [x,y,2] -> 21;
        [x,a,3|tail]    -> 31;
        [_,_,3,a,b,c,5] -> 32;
        [_,_,3,x,b,t,6] -> 33 end.
```

```
{switch,
    {tag,{var,x}},
    {fail},
    [{{cons,2},
      {switch,
          {shift,1},
          {switch,
              {shift,3},
              {fail},
              [{{int,3},
                {switch,
                    {shift,7},
                    {fail},
                    [{{int,6},
                      {ret,
                          {int,9,33},
                          {binds,[{t,{read,7}},{b,{read,6}},{x,{read,5}}]}}},
                     {{int,5},
                      {ret,
                          {int,8,32},
                          {binds,
                              [{c,{read,7}},{b,{read,6}},{a,{read,5}}]}}}]}},
               {{int,2},
                {ret,{int,6,21},{binds,[{y,{read,3}},{x,{read,1}}]}}}]},
          [{{int,2},{ret,{int,5,20},{binds,[]}}},
           {{int,1},
            {switch,
                {shift,2},
                {fail},
                [{{int,2},{ret,{var,3,tail},{binds,[]}}},
                 {{int,4},{ret,{int,4,11},{binds,[]}}}]}}]}}]}
```

Example:

```
case x of
1  {x1, x2, {x3, {x4, 1},        x6}}
2  {y1, y2, {y3, 2,              y5}}
3  {z1, z2, {z3, {z4, {z5, 3}}}, z7}} -> z1
4  {o1, o2, 4}
5  {p1, p2, {p3, 5}}
```

Extractors:

```
    1   2    3
             1   2               3
                 1     2
             1   2
```

Optimal Decision Tree should be Length 4 (reads):

```
{switch,{read,3},
  [{{int,4},{binds,[{o1,{read,1}},{o2,{read,2}}]}},
   {{tuple,2},
      {switch,{read,2},
        [{int,5},{binds,[{p1,{read,1}},{p2,{read,2}},{p3,{read,[3,1]}}]}],
         {fail}}},
   {{tuple,3},
      {switch,{read,2},
        [{{int,2},{binds,[{y5,{read,[3,3]}},{y3,{read,[3,1]}}]}},
         {{tuple,2},
            {switch,{read,2},
              [{{int,1},{binds,[]}},
               {{tuple,2},
                  {switch,{read,2},
                    [{int,3},{binds,[]}],
                     {fail}}}],
               {fail}}}],
         {fail}}}],
   {fail}}}.
```

7 Algorithm

The alrorithm uses Left to Right strategy for Lists and it is possible to plug any heuristics for Tuples matching. Type Constructors and Constant Values comes withing the same container and so do Variables and Wildcards. On the low level switch among Constructors and Constant Values should be handled separately, here its unified for clarity.

```
compile([],_,_) -> [];
compile(List,Exp,J) ->
```

```
Column = heuristic(List),
Fold = folda(Column,List),
io:format("~p Fold: ~p~n",[J, Fold]),
[ Ctors, Cvars ] =  [ group({Name},Fold) || Name <- [ctor,cvar] ],
case {Ctors,Cvars} of
     {[],   Cvars} -> compile(binds(Cvars,J+1),Exp,J+1);
     {Ctors,Cvars} -> switch({shift,J},
          [ case length(Rows) of
                1 -> {_,_,E,B} = hd(Rows), {C,{ret,E,{binds,B}}};
                _ -> {C,compile(exclude(Rows,Column),Exp,J+1)} end || {C,Rows} <- Ctors ],
            case length(Cvars) of
                0 -> {fail};
                1 -> [{_,L}] = Cvars,
                     {_,_,E,B} = hd(L),
                     {ret,{E,{binds,B}}};
                _ -> compile(binds(Cvars,J),Exp,J+1) end) end.
```

8 Literature

1 . Peter Sestoft. ML Pattern Match Compilation. 1996

2 . Luc Maranget. Compiling Pattern Matching. 2008

3 . SPJ, Philip Wadler. Implementation of Functional Languages. 1958.