Faster Laziness Using Dynamic Pointer Tagging

Simon Marlow[1] Alexey Rodriguez Yakushev[2]
Simon Peyton Jones[1]

[1]Microsoft Research Cambridge

[2]Institute of Information and Computing Sciences
Utrecht University

International Conference on Functional Programming 2007

Implementation of lazy evaluation

- Consider:

```
map :: (a->b) -> [a] -> [b]
map f xs = case xs of
                 []       -> []
                 (x:xs') -> f x : map f xs'
```

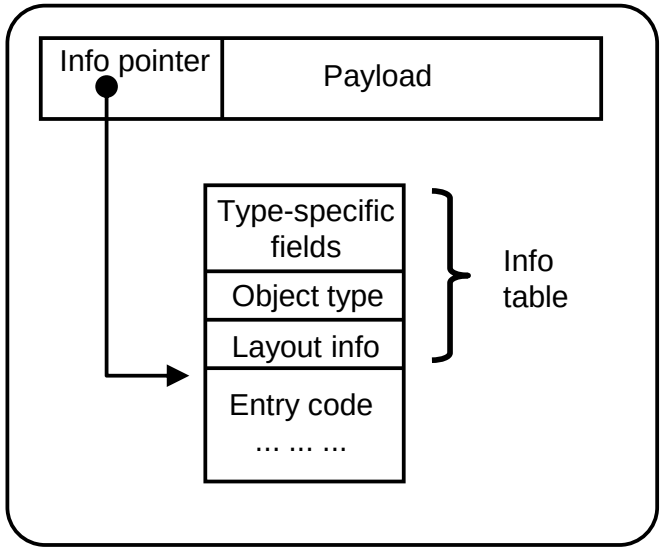
- Before pattern matching `xs`, it must be evaluated to WHNF.
- There are two ways to implement the evaluation of `xs`:
  - Tag-ful:
    - if `xs` is already a constructor, continue with case execution,
    - otherwise, jump to the closure code to evaluate it
  - Tagless: jump to the code of `xs` regardless of its evaluatedness.
- The tagless scheme is simple and uniform.
- This is the evaluation scheme that the Spineless Tagless G-Machine (STG machine) uses.

The tagless scheme is expensive on modern CPU architectures

- The tagless scheme uses too many indirect jumps.
- These jumps cause mispredicted branches, each having a penalty of 10 or more cycles depending on the architecture.
- Some programs exhibit up to 20% of stalled cycles due to branch mispredictions.
- In this talk, we will revisit the tagless design decision that is used in the STG machine.

In GHC, closures have a uniform representation

- The first field of a heap closure points to its entry code.
  - Jumping to the entry code of a closure evaluates it and returns the result in `R1`.
  - Constructor entry code returns immediately.
- The info table contains information such as the closure type (thunk, constructor, etc), the constructor tag (for constructors), etc.
- The payload varies per closure type (constructor fields, or the free variables of a thunk for example).



Tagless code for scrutinee evaluation

- A case expression is split into two blocks of code, scrutinee evaluation and branch selection.
- In the code below, `R1` points to the `xs` closure.

```
map_entry()
{
    ...
    // Put the case continuation on the stack
    Sp[-8] = map_ret;
    Sp = Sp - 8;

    // Indirect jump to entry code of xs
    jump R1[0];
}
```

Tagless continuation code for the case expression

- Invariant: on return `R1` points to a constructor, so we use the constructor tag to select a case branch.

```
data { word32[] = {...} } // Info table
map_ret()
{
    info = R1[0];
    tag = info[-8];
    if (tag == 0) jump tag_0;
    // Extract the fields for Cons:
    x    = R1[4];
    xs'  = R1[8];
    ...
 tag_0: // Nil constructor
    ...
}
```

- Note: if `xs` was an evaluated constructor it would have returned immediately.

The problems with tagless

- Closure evaluation uses two indirect jumps
  - one to enter the closure,
  - and another to return to the evaluation site.
- Idea:

  If the closure is already evaluated, a conditional jump might be much faster.
- Of course this will only be effective if there is a high portion of scrutinised closures that are already evaluated.

Frequency of already evaluated closures in case expressions

- At runtime, 61.89% of case expressions scrutinise an already evaluated closure.

Program	Evaluated scrutinee (%)
anna	65.1
cacheprof	72.8
constraints	54.5
fulsom	41.5
integrate	67.6
mandel	73.9
simple	78.6
sphere	72.8
typecheck	56.5
wang	41.9
(81 more)	...
Min	0.20
Max	99.00
Average	61.86

- A constructor fastpath is worthwhile.

Optimisation Scheme 1: Semi-tagging

- For semi-tagging, we guard the indirect jump with a test against the evaluatedness of the closure.

```
map_entry()
{   ...
    Sp[-8] = map_ret;
    Sp = Sp - 8;
    // ------------------------------------
    // Indirect jump to entry code of xs
    jump R1[0];
}
```

Optimisation Scheme 1: Semi-tagging

- For semi-tagging, we guard the indirect jump with a test against the evaluatedness of the closure.

```
map_entry()
{   ...
    Sp[-8] = map_ret;
    Sp = Sp - 8;
    // ------------------------------------
    // grab the info pointer from the closure
    info = R1[0];
    // grab the type field from the info table
    type = info[-12];
    // if it's a constructor, jump to the cont.
    if (type <= MAX_CONSTR_TYPE) jump map_ret;
    jump info;
}
```

- The continuation for `map` is exactly as before.
- See paper for improvements on this scheme.

Semi-tagging results

- This small modification has dramatic results, semi-tagging reduces the average runtime in 7.2% on an Opteron processor, while increasing code size in 1.5%.
- The total number of branches goes up by 15%, but even then the total number of branch mispredictions is nearly halved (46.3%)!
- The speedup is due to the reduced amount of mispredicted branches: in one of our tests, branch mispredictions account for 91% of the difference in cycles.

Our contribution: Dynamic pointer tagging

- This scheme uses the spare bits of a pointer to encode evaluatedness information.
- Much more efficient than the semi-tagging scheme (fewer memory accesses).
- Most pointer tagging and untagging comes for free.
- The scheme is dynamic: if a thunk becomes evaluated the pointer to it will eventually become tagged.

Optimisation Scheme 2: Dynamic pointer tagging

- Pointer tagging makes the evaluatedness test cheaper by encoding evaluatedness in the lowest bits of the closure pointer.

unevaluated or unknown	evaluated constructor
000	non-zero

- If the number of constructors is small enough, the tag bits can even encode the tag of the constructor.

unevaluated or unknown	evaluated constructor			
	tag 0	tag 1	tag 2	...
000	001	010	011	...

- If the number of constructors is too high, the following encoding is used:

unevaluated or unknown	evaluated
000	001

- How many bits is enough?

Are three bits enough to encode datatype constructors?

- Constructors encountered in case expressions, classified by family size.

Family Size	Distribution (%)	Cumulative (%)
1	42.5	42.5
2	52.4	94.9
3	1.2	96.1
4	0.5	96.6
5	1.7	98.3
6	0.9	99.2
7	0.0	99.2
> 7	0.8	100.0

- Two bits are enough to encode 96.1% of inspected constructors in our benchmarking suite.
- And three bits are enough to encode 99.2% of them.
- Not very surprising: datatypes like lists, `Bool`, `Int` and `Char` are very common.

Pointer tagging: evaluating the scrutinee

- With pointer-tagging, the evaluatedness test does not dereference the info closure and pollute the cache with the info table.

```
map_entry
{
    ...
    Sp[-8] = map_ret;
    Sp = Sp - 8;

    // if it's a constructor, jump to the
    // continuation
    if (R1 & TAG_MASK != 0) jump map_ret;
    jump R1[0];
}
```

- Note: it is safe to leave a constructor pointer untagged.
- But constructor code should return an untagged pointer.

Branch selection in a case expression

- Invariant: on return `R1` is a tagged pointer to a constructor.
- Lists have two constructors, so the constructor tag appears in the pointer.
- Again, this avoids loading the info table into the cache.

```
data { word32[] = { ... } }
map_ret
{
    tag = R1 & TAG_MASK;
    if (tag == 1) jump nil_alt;
    // else, tag == 2
    // Code for (x:xs') alternative
    ...
 nil_alt:
    ...
}
```

Cheap untagging

- In the cons case, rather than masking out the tag:

```
// Code for (x:xs') alternative
// Extract the fields from offsets 4 and 8
R1  = R1 & ~TAG_MASK
x   = R1[4];
xs' = R1[8];
...
```

- We do untagging subtracting the constructor tag directly:

```
// Code for (x:xs') alternative
// Extract the fields from offsets 4 and 8
x   = R1[4 - 2];
xs' = R1[8 - 2];
...
```

Cheap tagging

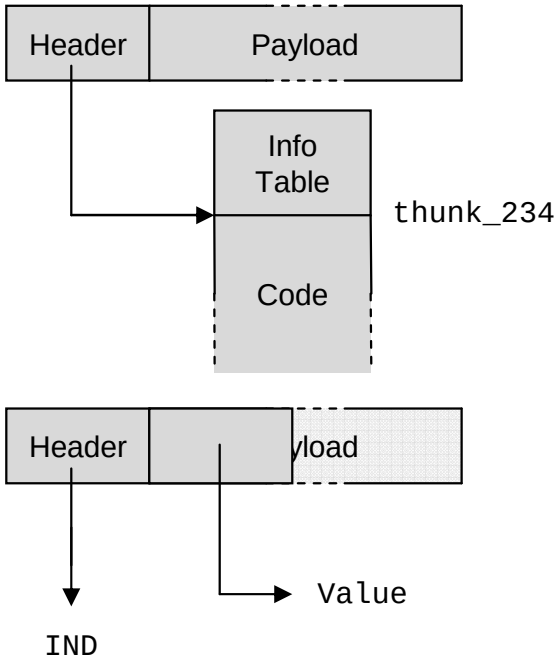
- Dynamic closures are tagged at creation, consider `length [x]`.

```
f_entry
{
  ... increment Hp and check for heap overflow ...
  Hp[-8] = cons_info;
  Hp[-4] = x;
  Hp[ 0] = Nil_closure + 1;
  R2 = Hp - 8 + 2;
  jump length_info;
}
```

- For static closures the story is similar, see paper.
- In the RTS we do need to mask out the tag bits, since we cannot know what are the tag bits statically.

Propagation of tag bits from indirections

- When a thunk is evaluated, it is overwritten by an indirection to the actual value:



- A recently evaluated thunk can be considered non-evaluated because the tag bits are in the indirection.
- The GC already removes indirections as part of its traversal, so it was modified to propagate the pointer tags of indirectees.
- This makes pointer tagging dynamic, since the tag of a closure pointer is changed at runtime.

Results

- Pointer tagging reduced runtimes by 13.7% on average on the AMD Opteron.
- Compared to semi-tagging, this is an improvement of 5.5%.
- Compared to semi-tagging, the amount of branch mispredictions was not changed.
- The improvement over semi-tagging comes from a reduction in cache miss activity (L1: 4.5% , L2: 20.1%).

Tagging pointers to functions

- When an unknown function is applied to some arguments, a generic apply code sequence checks that:
  - the function closure is evaluated, and that
  - the function closure has the right arity.
- We tag pointers to functions with their arity:

unevaluated or unknown	evaluated constructor arity 1	arity 2	...
000	001	010	...

- Using the tag bits we can have a fast path for the case when the function arity matches the number of arguments.
- This optimisation gives an additional 1-2% performance benefit.

Conclusions

- The tagless lazy evaluation scheme is too expensive in modern CPUs.
- We propose a novel and efficient solution: dynamic pointer tagging.
- Relevant for anyone building a compiler for a lazy functional language.
- Impressive gains for a mature optimising compiler (14%).
- Now in GHC 6.8!