mort注：

首先，这不是一篇告诉你“Haskell和C一样快”的软文，并不存在这样的说法；这只是一篇关于“ 在某些情况下你可以做到让GHC生成的机器码跑得像用C写出来的一样快，以及如何做到这一点 ”的个案研究。

以下是翻译君关于Haskell性能问题的科学解释

~~（其实只是冗长的个人吐槽而已）~~ ，科普为主，如果对Haskell已经比较了解的话请自动忽略。

事情起源于reddit上一场关于Haskell效率的讨论：

Damn Lies and Haskell Performance

该帖的楼主用Haskell、Scheme和C实现了完全相同的计算Fibonacci数的算法，然后分别使用GHC、Gambit-C和GCC对代码进行了最高优化等级的编译。最后得到的机器码，C毫无悬念地性能最高，Scheme（Gambit-C）生成的机器码比C慢一倍以上，而Haskell（GHC）生成的机器码，比Scheme（Gambit-C）甚至还要慢上一些。所以，楼主提出了这样一个问题：Haskell效率高，这种说法是真的吗？

这个问题说起来有点复杂。首先，正如许多人第一眼所看到的那样，该楼主使用的Fibonacci函数是一个最naive的纯裸递归（非尾递归调用，无memoization，是用来计算Fibonacci函数的最简单直接也是最低效的方式），这意味着每一次递归执行该函数，调用栈的开销就会增殖一倍，随着n的增加，计算量（时间空间开销）呈指数级增长，就计算较大的Fibonacci数来说，不管对于任何一种程序语言而言，该算法的时间空间代价显然是不可承受的。

```
fib :: Int -> Int

fib n =
  if n < 2 then
    n
  else
    fib (n - 1) + fib (n - 2)
```

自然，他的意图很明显：拿一个最简单的递归的极端情况作为micro-benchmark，来测试对比各种语言（当然这里主要是Haskell）的性能。正如帖子里的结果所显示的那样，GHC生成的机器码在这项benchmark上的性能远远落后于GCC生成的机器码。所以，刚开始接触Haskell的人如果有心做了类似这方面的benchmark测试对比，差不多都会问同样的一个问题：Haskell的效率可以和C媲美的江湖传闻，是真的吗？抑或只是一个天大的谎言？

正好这也是我自己在一年前的状况。那时，我拿Tak函数做了一个略不科学的测试（其实只是为了找到在Code Jam的时候适合我这种不懂算法的人拿来做递归搜索的语言(´д`;) ），最后得到的结果（按运行时间从短到长排序）是：

```
OCaml (ocamlopt) < Go (gc) < C (GCC) < Standard ML (MLton) << Haskell (GHC)
```

（注：这个结果是在所有编译器都不加优化选项的默认情况下得到的，如果对GCC加上更高级别的优化，排名应该还要再靠前些）

当时我跑了一个极大的数，在生成的机器码性能上，前四者并没有质的差别，可是当同样一个最简单的Tak函数实现（代码本身没有任何优化技巧）被移植到Haskell （GHC）上时，同其他语言生成机器码在效率上的差距就大大地被拉开了（我观察到的是：C或ML编译器生成的机器码几乎比GHC生成的机器码快了一个数量级）。这与该帖子的结论吻合。所以，至少GHC在处理一般情况下的纯递归benchmark（不包含延迟求值优势的函数，比如Tarai）上效率要低于大部

分C和ML编译器，基本上是铁板钉钉的事实。

（顺便科普一发，如果你不知道 Tak/Tarai函数 是什么：

对于递归函数的计算而言，实际上存在两种状况：一种是大部分场合下更常见的递归函数，你调用它一次，压栈，再调用，再压栈，及时求值，返回（这是所有急性求值语言的通常工作方式），对于这种绝大多数情况下的函数，惰性求值（在计算时间上）不具备任何理论上的优势；

然而，存在着另一种递归函数。 对于这些特例而言，恰恰相反，惰性求值占据了绝对的理论优势。 这一类函数，如果你用及时求值的方式花了二年半去算，算到最后才会发现，这么多层递归中的计算其实都是不必要的（它们既没有副作用，又丝毫不对函数最终的求值产生影响），在这种情形下，call-by-need机制的惰性求值语言（诸如Haskell）占据了得天独厚的优势。对于这种延迟求值优势函数的计算，Haskell可以秒杀其他一切语言。你可以自己试试看同一个Tarai函数用Haskell实现以及用C、或ML实现在效率上的天差地别。把我当时做这个micro-benchmark的代码贴在下面：

C版本：（注意到我在这里尝试了两种优化方式，但它们仍然输给了在代码写法上没有任何优化技巧的Haskell程序）

```
int tarai(int x, int y, int z) {
    if (x <= y) {
        return y;
    }
    return tarai(tarai(x - 1, y, z), tarai(y - 1, z, x), tarai(z - 1, x, y));
}

int tarai_m(int x, int y, int z) {
    if (m[x][y][z]) {
        return m[x][y][z];
    }
    if (x <= y) {
        return y;
    }
    int p1 = tarai_m(x - 1, y, z),
        p2 = tarai_m(y - 1, z, x),
        p3 = tarai_m(z - 1, x, y);
    m[p1][p2][p3] = tarai_m(p1, p2, p3);
    return m[p1][p2][p3];
}

int tarai_(int x, int y, int z) {
    while (x > y) {
        int oldx = x, oldy = y;
        x = tarai_(x - 1, y, z);
        y = tarai_(y - 1, z, oldx);
        if (x <= y) break;
        z = tarai_(z - 1, oldx, oldy);
    }
    return y;
}
```

Go版本：

```
func tarai(x, y, z int) int {
    if x <= y {
        return y
    }
    return tarai(tarai(x - 1, y, z), tarai(y - 1, z, x), tarai(z - 1, x, y))
}
```

Standard ML版本：

```
fun tarai (x, y, z) =
    if y < x then tarai (tarai (x - 1, y, z), tarai (y - 1, z, x), tarai (z - 1, x, y))
    else y

val _ = print (Int.toString (tarai (22, 10, 1)) ^ "\n")
```

OCaml版本：

```
let _ =
    let rec tarai (x, y, z) =
        if y < x
            then tarai (tarai (x - 1, y, z), tarai (y - 1, z, x), tarai (z - 1, x, y))
            else y
    in
    Printf.printf "%d\n" (tarai (22, 10, 1))
```

最后，是战斗力爆表的Haskell版本：

```
tarai :: Int -> Int -> Int -> Int

tarai x y z
    | x <= y = y
    | otherwise = tarai (tarai (x - 1) y z) (tarai (y - 1) z x) (tarai (z - 1) x y)

main = print (tarai 22 10 1)
```

注意到，Tak/Tarai函数的两种形式在定义上只存在微妙的差别（最初定义的Tarai函数是那个惰性求值占优的形式，后来的变体则不存在这个优势）；正是这点微妙的差别决定了一个函数属于前者（惰性求值不具备理论优势），而另一个函数属于后者（惰性求值占据绝对优势）。关于Tak函数，Donald Knuth曾经探讨过它的一般形式和递归终止的条件，可以参考 这篇report 。

题外话，扯远了。惰性求值在特例上的优势不是我今天想讲的话题……回到关于Haskell本身效率的讨论上）

通常，对Haskell了解不得深的人喜欢在惰性求值上大做文章，揪住它的弱点来作为黑Haskell的依据：正是这种call-by-need的惰性求值方式，造成了机器码中额外的堆开销（尤其在执行深层递归时）；惰性求值非常不利于编译器的优化；惰性求值导致了Haskell程序性能的低下。因此，惰性求值是一个根本性的设计错误，基于惰性求值的Haskell是个烂语言，云云。

换个角度，其实你可以问自己三个问题： 第一，处理这种极端状况的micro-benchmark下递归的性能，在实际的应用中真的有那么重要吗？ 第二，如果说GHC生成的机器码性能不高，那么，你非得让GHC这么做不可吗？ 是不是考虑换一种使用Haskell的方式会比较好呢？ 第三，惰性求值导致效率低下这件事情，是必然的吗？ 或者说，GHC生成机器码效率的低下，一定就是Haskell本身设计哲学的问题吗？

首先，我想阐释清楚一个事实。正如reddit那篇帖子的回复中所提到的，GHC生成的机器码效率的低下，很大程度上要归咎于其对Haskell这种惰性求值方式的处理：每次处理一个thunk，都是对内存堆（以至于执行时间上）的巨大消耗。一方面，在每进入一个递归函数的时候，需要占用额外的开销去处理thunk；另一方面，在内存中为thunk分配堆frame这件事情，本来机器指令的执行效率就要远低于使用栈（急性求值的语言如C和ML，就更易得益于硬件体系本身对栈的底层优化）。想一想，这个递归函数本身是指数时间，放到基于堆的GHC实现里，还要再乘上一个用于消耗在处理thunk和内存堆上的因数，生成的机器码效率较低是可以预见的。也就是说，惰性求值方式比之急性求值来会产生更大的额外开销（所谓的Lazy evaluation overhead），本来就是每一个Haskeller都应该了解的事实。惰性求值的这种额外开销，在重

度考察递归性能的micro-benchmark（比如Tak函数，还有上面reddit帖子里这个纯裸递归计算Fibonacci数的例子）中会体现得尤其明显。

然后，针对上面的三个问题，给出我自己的答案。

第一， 在现实的软件开发中，这种依赖深层递归 （并由此产生巨额堆分配的lazy evaluation overhead） 的极端情况，事实上并不十分常见。

如果你要做算法题，直接拿一个裸的不做任何优化手段的递归函数去死磕求值（在参加算法比赛时，一般不会有太多多余的时间耗在代码以至于硬件级别的优化上，算法和数据结构的高阶优化才是重点），那么，Haskell显然并不十分适合。在这种情况下，C，ML，甚至哪怕是Scala的编译器都能够帮你去做更多的底层优化；而Haskell的惰性求值方式所带来的堆分配overhead，很可能会是一场灾难。简而言之，就算你一定要在算法竞赛中用Haskell，那么，注意避开深层次的递归是必须的；你应该选择一个更好的算法，或者，Haskell语言中更好的部分，比如monad。

现在假设另外一种情况，如果你要做实用的软件开发，比如，拿Yesod搭一个web服务。你会需要搞出一个深层递归的东西去让它疯狂分配堆然后死耗内存吗？基本不可能。我想这也是世界上有那么多Haskellers，但是他们并不特别纠结于这类单纯的micro-benchmark的原因。递归性能仅仅是函数式语言效率的一个非常个体化的方面，不能代表（在日常的软件开发情境下的）整体效率，更不能以偏概全地拿它来代表整个并行系统的效率。

退一步说，如果你确实有必要在Haskell中进行这样的重度计算，这些惰性求值带来的额外开销是完全可以回避的。在上面这个reddit帖子里，tdammers贴出了一种使用State monad的解决方案，它能够生成高效的机器码。于是，这就是对第二个问题的解答： 如果你足够了解Haskell，你就不可能在解决惰性求值并不适合的问题上采取高度依赖求值本身效率的代码写法 ，然后拿它生成的机器码和C或ML作比较，指望它能拥有同样的性能。这只会让情况更糟。硬要拿急性求值的优点来和惰性求值的软肋放在一起比较说事，等于是让龟兔在一起赛跑，最后得出一个结论：从进化论的角度看，兔子适合生存，乌龟不适合生存。这是一种荒谬的比较学。

关于第三个问题，GHC生成机器码效率的低下，一定就是Haskell本身设计哲学的问题吗？

注意到，这件事情的本质，是GHC对惰性求值所采取的处理方式（通过在内存中分配heap frame来构造thunk）并不利于程序（在这类纯递归的micro-benchmark上）的性能提升，因为每次递归时对堆的访问均是一笔巨大的内存开销，这和编译器的底层优化存在着不可调和的矛盾；而使用栈的实现（正如C和ML的编译器所做的那样），就不存在这种lazy evaluation带来的额外开销，亦可以更多地得益于高效的机器原生栈的支持，正如那篇帖子里得分最高的一篇回复所指出的那样，现代CPU体系（如x86/64）对栈的底层支持（更短的指令序列和处理器优化，寄存器缓冲机制）使得利用本地栈的C和ML在底层程度上通常能得到高度优化的代码，Haskell（GHC）则在机器级别上缺乏这种底层优势；而在以前的Sparc体系上，由于机器本身对栈提供的优化支持并不特别明显（特别地，提到了Sparc的寄存器缓冲在递归的情形下表现糟糕），所以在那时，使用栈的C或ML比起Haskell来并不具备这种特别的优势，性能的差距当然也就没有像在今天的CPU体系上这么显而易见。

所以，GHC在这类micro-benchmark中表现出的低效率， 从本质上说，这并不是Haskell语言设计上的问题，甚至也不是惰性求值本身的问题，更多的是一个编译器的设计实现该如何去适应（不断随时代演变的）机器底层优化的问题。 你已经看到了，在reddit的原帖子里，该作者重新使用了JHC（使用C进行中间编译，编译得到的机器码自然也就使用了C的栈）进行测试，最终得到的结果是，至少在这个纯递归的micro-benchmark上，JHC完胜GHC（因为基于栈的实现带来了底层级别的优势），可以达到接近C的程度。

在开头我提到过，这不是一篇告诉你“Haskell和C一样快”的文章。理由就是，单纯地拿“一种语言的性能”同“另一种语言的性能”作比较，谈论“谁比谁快”，本来就没有任何实质意义。快的是实现该语言的某种编译器产生的机器码/字节码，而不是这种语言本身。如你已经

看到的那样，即使是同样的Haskell语言，同一段计算深层递归的代码，放到GHC和JHC上进行编译，效率会有巨大的差别；另外，即使是同样的编译器实现，采用不同的编译优化级别，或者放在不同的体系平台上（比如，指令集针对栈作特别优化的体系和无特别优化的体系）分别进行编译，性能也会产生质的差异。单就语言本身的某个特性泛泛而谈地讨论性能的优劣，缺乏科学性可言。惰性求值的设计也许确实给编译器的高效实现带来了困难（相比较能够最大限度利用机器本身对栈的优化的急性求值语言来说），但是，效率这件事情归根结底还是依赖于编译器的设计与实现：实现Haskell这样的惰性求值语言，你可以像GHC那样通过在内存里分配堆来处理thunk，这样就比较难以通过机器底层的优化来提升性能，做递归的时候由于堆造成的时间和空间消耗非常严重；你也可以像JHC或者Fay那样把Haskell编译成另一种高级语言的中间码（C或者JavaScript），然后让这些语言现成的、高效的编译器去负责底层优化的“脏”任务，这样做也就回避了直接在内存中进行堆分配造成的性能问题。

然后，下面我要翻译的这篇文章，虽然与惰性求值无关，主要是讲如何在GHC中实现代码级别的优化、以达到可以和C相提并论的效率的：

Yet Another Lambda Blog » Haskell as fast as C: A case study

在正常情况下（非重度递归中，惰性求值带来的额外开销并非影响性能的主要因素时），如同此文中这个list操作的简单例子，Haskell程序的性能是可以优化到与C接近的地步的。但是你在后面也会发现，这样的代码级别优化是有代价的，为了使在GHC上最终编译得到的性能与C接近，你需要牺牲那些“显而易见”的代码写法，转而写出不那么优雅的解决方案。这当然不是什么好事情，因为如你所知，GCC、MLton这些编译器已经可以做到非常变态的底层级别优化，所以在C和ML里，显而易见的代码通过编译器优化出来的结果大部分时候已经可以充分榨干机器的每一寸性能；而GHC目前还远远做不到这一点。你将会看到，为了让Haskell写出来的程序在GHC上具有可以与C相匹敌的性能，你需要使用一些奇技淫巧，而且这只有在你充分熟悉Haskell实现的前提下才能做得到。而在C和ML里，你并不总是需要了解编译器本身，就能写出性能远比Haskell要高得多的代码。

所以，下面的这篇文章，仅仅印证了一件事情： 你可以这么做。 不代表你应该这么去做。

关于Haskell性能能够与C匹敌的神话，目前还只是一个神话；事实状况是还有很长的一段路要走。当然，这也并不代表GHC不是一个好的编译器，只因为它的focus并不在底层优化这方面；更不代表Haskell就必须意味着计算性能的低效，想想前面那个延迟求值优势的例子（Tarai函数）。

好了前面废话略多（你丫废话真tm多。。。）原文在此：

Original Article: Haskell as fast as C: A case study by Jan Stolarek
(Chinese Translation by Mort Yao)

偶尔会有这样的人，通常是初学者，问起关于Haskell与C的性能比较方面的问题。事实上，在我初学Haskell的时候，我也曾问过同样的问题。最近几天，我尝试了从一段短小的Haskell程序中压榨出性能，最后得到的结果几乎可以与C相媲美，所以我想分享一下这个结果。这将是一个短小的个案分析，我不想在这里全面地覆盖关于Haskell和C性能比较的大课题。关于这方面已经有了许多文章，建议你去搜索Haskell-cafe的归档和其他博客。在这其中，我最为推崇的是Don Stewart的这篇博文： Write Haskell as fast as C: exploiting strictness, laziness and recursion 和这篇： Haskell as fast as C: working at a high altitude for low level performance 。

这儿是我的简短的代码：

```
sumSqrL :: [Int] -> Int
sumSqrL = sum . map (^2) . filter odd
```

它的输入是一个Int的list，从中移除所有的偶数，对剩余的奇数计算平方后进行求和。这是

一段合乎常规的Haskell代码：它用到了标准Prelude中内置的list处理函数，依赖函数组合，得到的代码既清晰可读又模块化。我们怎样让它更快呢？最简单的方法是迁移到一个更加高效的数据结构上，也就是未包装的（Unboxed）Vector：

```
sumSqrV :: U.Vector Int -> Int
sumSqrV = U.sum . U.map (^2) . U.filter odd
```

从实际的角度来看，代码写法并没有什么改变，除了类型签名和命名空间的前缀那部分，它是用来避免同Prelude中的名字冲突的。你会看到，这段代码大约比以前使用list的版本快了3倍。

我们能做得更好吗？是的。下面这段代码比使用Vector的版本还要快三倍，但这么做是有代价的。我们需要牺牲代码的模块性与优雅性：

```
sumSqrPOp :: U.Vector Int -> Int
sumSqrPOp vec = runST $ do
  let add a x = do
        let !(I# v#) = x
            odd#     = v# `andI#` 1#
        return $ a + I# (odd# *# v# *# v#)
  foldM` add 0 vec
```

这段代码同样使用了未包装的Vector。用来fold Vector的 `add` 函数，获取一个累加器 `a`（在调用 `foldM'` 期间被初始化为0）和一个Vector的元素作为参数。为了检查该元素的奇偶性，该函数需要首先将其解包，然后将其除最低有效位之外的所有位均置零。若这个Vector元素为偶数，那么 `odd#` 将包含零；若其为奇数，那么 `odd#` 将包含1。通过将该Vector元素的平方与 `odd#` 相乘，我们避免了一个条件判断分支指令，作为代价，我们付出的是可能不必要的对偶数元素的乘法和加法运算。

让我们来看看这些函数实际编译成Core代码是怎样的。 `sumSqrV` 会是这样：

```
$wa =
  \vec >
    case vec of _ { Vector vecAddressBase vecLength vecData ->
    letrec {
      workerLoop =
        \index acc ->
          case >=# index vecLength of _ {
            False ->
              case indexIntArray# vecData (+# vecAddressBase index)
              of element { __DEFAULT ->
              case remInt# element 2 of _ {
                __DEFAULT ->
                  workerLoop (+# index 1) (+# acc (*# element element));
                0 -> workerLoop (+# index 1) acc
              }
              };
            True -> acc
          }; } in
    workerLoop 0 0
    }
```

而 `sumSqrPOp` 则编译为：

```
$wsumSqrPrimOp =
  \ vec ->
    runSTRep
      ( (\ @ s_X1rU ->
          case vec of _ { Vector vecAddressBase vecLength vecData ->
          (\ w1_s37C ->
```

```
            letrec {
              workerLoop =
                \ state index acc ->
                  case >=# index vecLength of _ {
                    False ->
                      case indexIntArray# vecData (+# vecAddressBase index)
                      of element { __DEFAULT ->
                      workerLoop
                        state
                        (+# index 1)
                        (+# acc (*# (*# (andI# element 1) element) element))
                      };
                    True -> (# state, I# acc #)
                  }; } in
            workerLoop w1_s37C 0 0)
        })
      )
```

我稍微清理了一下代码，好让它们看起来更加可读。在第二个版本中有一些ST monad引入的噪声，但除去这些以外，两段代码实际上非常相似。它们的差异在于， `workerLoop` 是如何在最内层嵌套的case表达式中被调用的。第一个版本进行一个条件性的、对两个可能的 `workerLoop` 之一的调用；而第二个版本执行的是无条件调用。这看起来并没多大差别，然而实际结果却显示出了巨大的差异：第二个版本代码的性能可以与C相媲美，而第一个版本的代码却要慢上3倍。

让我们来看看LLVM后端生成的汇编代码。 `sumSqrV` 的主循环部分编译为：

```
LBB1_4:
    imulq    %rdx, %rdx
    addq     %rdx, %rbx
.LBB1_1:
    leaq     (%r8,%rsi), %rdx
    leaq     (%rcx,%rdx,8), %rdi
    .align  16, 0x90
.LBB1_2:
    cmpq     %rax, %rsi
    jge      .LBB1_5
    incq     %rsi
    movq     (%rdi), %rdx
    addq     $8, %rdi
    testb    $1, %dl
    je       .LBB1_2
    jmp      .LBB1_4
```

而 `sumSqrPOp` 的主循环则编译为：

```
.LBB0_4:
    movq     (%rsi), %rbx
    movq     %rbx, %rax
    imulq    %rax, %rax
    andq     $1, %rbx
    negq     %rbx
    andq     %rax, %rbx
    addq     %rbx, %rcx
    addq     $8, %rsi
    decq     %rdi
    jne      .LBB0_4
```

无需成为一个汇编专家，你也能看得出第二个版本生成的机器码明显更加紧凑。

我在前面承诺过要拿它和C来作比较。代码在此：

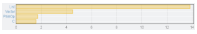
```
long int c_sumSqrC( long int* xs, long int xn ) {
  int index   = 0;
```

```
  int result  = 0;
  int element = 0;
 Loop:
  if (index == xn) goto Return;
  element = xs[index];
  index++;
  if ((0x1L &amp;amp; element) == 0) goto Loop;
  result += element * element;
  goto Loop;
 Return:
  return result;
}
```

你可能会质问我，为什么要在这里使用goto语句。理由是，这个“奇数平方累加函数”的例子原本是从Richard Waters的这篇论文“ Automatic transformation of series expressions into loops （序列表达式到循环的自动变换）”里借用来的，我试图在这里忠实地模拟他的融合（fusion）框架所生成的结果。

翻译君注：这里提到的融合，意即 stream fusion ，是指一种在list操作过程中、通过某个基于函数的变换来消除中间数据结构（例如中间list）的优化手段，可以极大地加速list操作的性能。 在Haskell中 有一套相应的库可以用来完成借助于stream fusion的list操作。我把这个词暂时叫作“流融合”。（实在是没找到这方面对应的中文资料）

我使用了一套测试基准来对比上述提到的四种实现：基于list的实现，基于vector的实现，基于vector并采用了 foldM +primops的实现，和C语言的实现。我在Haskell中使用FFI去调用这个C的实现，故而可以基于相同的基准来进行科学的分析。这里是一个包含了1百万个元素的list/vector的最终结果比较：



C的版本仍然比基于primops的实现快了大约8%。我认为这已经是一个非常可观的结果了，鉴于那个基于Vector库的版本慢得是它的3倍的前提下。

一个简要的总结

Vector库在其内部采用了流融合（stream fusion）的实现，用以优化其对vectors进行操作的代码。在我前面提到过的Don Stewart的博文中，也探讨了一些关于流融合的问题，但如果你想要了解更多的知识，你也许会对这两篇论文感兴趣： Stream Fusion. From Lists to Streams to Nothing at All 和 Haskell Beats C Using Generalized Stream Fusion 。我的 sumSqrPOp 函数，尽管几乎可以做到和C一样快，事实上是相当丑陋的，我当然不会推荐任何人去写出这样的Haskell代码。你可能已经注意到，尽管 sumSqrPOp 实现方式的高效来源于在机器代码的循环内部对条件指令的避免，那个C语言的实现版本实际上在循环里用到了条件指令以确定某元素的奇偶性。有趣的是，这个条件判断在GCC编译器的优化过程中被自动消除了。

正如你所看到的这样，在Haskell中写出与C同样高效的代码是有可能的。坏处是，为了得到这种高效的代码，你将不得不牺牲函数式编程范式的优雅与抽象性。我希望在将来的某一天，Haskell能够拥有一个全能的融合（fusion）框架，比起今天的框架来能够做到更多的优化，使得我们能够在保持Haskell代码优雅性的同时也拥有机器码的高性能。毕竟，如果gcc能够避免不必要的条件判断指令，在GHC中应当也同样能够做到。

一个简短的附录

如需dump GHC产生的Core信息，在编译时使用 -ddump-simpl 命令行参数。同时建议使用 -dsuppress-all 参数，它将在输出时跳过来自类型的全部信息——这将使Core更加简单可读。

如需dump GHC产生的汇编代码，使用 `-ddump-asm` 参数。当使用LLVM作后端编译时，则需使用 `-keep-s-files` 参数。

如需反汇编产生的目标文件（如编译后的C文件），可使用 `objdump -d` 命令。

更新——有关Reddit上的讨论

在reddit上展开了关于该帖子的讨论，我想要在这里着重回应一下评论中出现的一些意见。

Mikhail Glushenkov指出下面的Haskell代码其实能够产生和我的 `sumSqrPOp` 函数相同的结果：

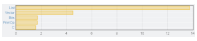
```
sumSqrB :: U.Vector Int -> Int
sumSqrB = U.sum . U.map (\x -> (x .&. 1) * x * x)
```

我承认我没有注意到这个简单的解决方案；我当初应该选择一个更好的问题（不存在类似这样的简单解法）来说明。

围观群众表示希望看到一个与常规C代码的性能比较结果，因为我在前面所展示的C代码很明显并不是常规的写法。那么，这里是我所能想到的最常规的一段C代码（虽然这并不代表它是最高效的）：

```
long int c_sumSqrC( long int* xs, long int xn ) {
  long int result = 0;
  long int i = 0;
  long int e;
  for ( ; i < xn; i++ ) {
    e = xs[ i ];
    if ( e % 2 != 0 ) {
      result += e * e;
    }
  }
  return result;
}
```

它的性能看起来和从前一模一样。（“Bits”代表Mikhail Glushenkov的解决方案，而“C”代表新的C代码）：



有人建议我采用如下的C代码：

```
for(int i = 0; i < xn; i++) {
    result += xs[i] * xs[i] * (xs[i] & 1);
}
```

作者声称这段代码运行起来比我所用的版本更快。不过我没能在我的机器上重现这一点——我得到的结果反而是更慢了（2.7ms相比之1.7ms——在1,000,000个元素的情形下）。也许这是因为我用的是GCC 4.5，而最新版本是GCC 4.8的缘故。

最后，关于通过FFI来调用C代码所引入的额外开销部分，存在着一些质疑。在我最初想到通过FFI来为C代码作benchmark测试的时候，我也意识到了这一点。在经过一些试验之后，我发现这带来的额外开销是如此微不足道，以至于可以被忽略。想了解更多信息，请参见这篇 帖子 。