



Notes on Haskell Debugging



Felis sap... 

函数式编程、编程语言、编程 话题的优秀回答者

已关注

圆角骑士魔理沙、祖与占、题叶、酿酿酿酿酿泉、草莓大福等 57 人赞了该文章

本文是 Deep Dark Haskell 系列的第二集（[前作](#)），汇总一下我所知道的所有 Haskell debugging 方法和注意事项。有一部分是跟 profiling 相关的，不过性能问题也是问题嘛，就一块算在内了。

使用条件编译

传统 C/C++ 项目中常见的一个 idiom 是使用宏来区分 debug/release 代码。通过修改编译选项，即可激活/关闭 debug 相关的 code path（比如各种 pre/post condition 的 assert 等）。在 Haskell 中也可以这样做。

一个例子是 [vector](#)，它在 Cabal 配置中提供一系列 flag，对应不同的 check（比如 bound check），可以在 Cabal 的 configure 阶段决定是否打开，而在 vector 的源代码中包含了对应检查的 C 预处理器宏。对于依赖 vector 的项目而言，在经过充分测试，有信心不会产生 out-of-bound 等错误之后，发布 release 版本的 binary 时，即可将 vector 中所有的 check flag 关掉以保证性能。

另外，Haskell 提供了原生的 [assert](#)，在 ghc 编译选项中带 -O 时会自动关闭。

关于 logging

不论语言，一个很原始也很有用的 debugging 手段就是打日志。Haskell 中应用最广的 logging 框架是 [fast-logger](#) 和 [monad-logger](#)，前者实现了针对多核运行时优化的 logging，后者则在前者基础上实现了 mtl-style 的 MonadLogger class，可以很方便地在 mtl stack 中增加



默认支持的 logging 后端是 stdout 或者文本文件，不过 fast-logger 和 monad-logger 中我们都可以接上自己的后端，比如用 redis 专门做日志服务器。

关于 trace 和 unsafePerformIO

在 Haskell 中，IO 类型是带传染性的，这使得一般的 print 调试法用起来不那么方便：如果临时起意需要在纯函数中输出一点东西，却不想让函数被 IO 类型玷污，怎么办？可以使用 `trace`，其作用是被求值时输出一段 String 并返回原来的参数。`trace` 的实现基于 `unsafePerformIO`，其类型签名说明了一切：`IO a -> a`，能够让一段假装自己 pure 的代码执行任意 IO action。除了 debugging 的以外，有许多场合（比如优化性能、实现全局变量、etc）也用到了这个组合子。

`trace`、`unsafePerformIO` 等组合子的存在说明 Haskell 的所谓类型安全、引用透明等特性实际上只是一个君子协定而已——能玩坏她的手段还不只于此。为了安全地使用这类在 pure code 中嵌入 IO action 的组合子，需要对 Haskell 的编译和运行机制有一些基本了解，避免一些陷阱：

- 嵌入了 `unsafePerformIO` 的 pure code，在 ghc 的眼中与同类型的 pure code 别无二致，而 ghc 针对一般的 pure code 有许多优化措施：`inlining`、`let-floating` 等等，这使得假装 pure 的 IO action 执行时机可能与我们期望的相左——`inlining` 使 IO 可能被重复执行，而 `let-floating` 使 IO 可能被只执行一次。所以应该尽量将此类代码集中到少数 module 中，使用 `OPTIONS_GHC` pragma 单独增加编译选项控制优化，使用 `NOINLINE` pragma 显式禁止 `inline`，等等。
- Haskell 默认规约策略为 lazy evaluation，一个含 `unsafePerformIO` 调用的表达式，在被规约到 WHNF (Weak Head Normal Form) 时，相应的 IO action 才会被执行。至于什么时候一个表达式会被规约到 WHNF 呢？一个不精确的答案是“发生模式匹配时”，而精确的答案需要阅读 ghc 的 STG dump 才能确认，后面会提到怎么做。

在有余力实现妥善的 logging 时，请务必用 logging 代替 trace，毕竟为了解决旧的 bug 而引入新的 bug 总是不合算的。

ghci 断点调试

Haskell 代码的执行方法有 2 种：编译成 object code，作为原生代码链接执行，或编译成 byte-code，通过 ghci 解释执行。如果采用后一种，那么可以使用 ghci 的断点调试功能，设置断点并单步执行。详细方法参考 [4.5. The GHCi Debugger](#)。

在 Haskell 中获取 call stack

从 ghc 8 开始，我们可以在 Haskell 中获取 call stack 并进行处理。只要在函数类型签名中挂上 `HasCallStack` constraint，在该函数出错时，出错信息将自动附上 call stack。同时，`GHC.Stack` 提供了 `withFrozenCallStack`，可以在我们自己的异常处理代码中冻结当前 call stack 并获取其内



阅读 STG dump

ghc 将 Haskell 源代码编译到原生代码的 pipeline 中，经历了多种不同中间表示的转换，可以通过各种 `-ddump-xx flag` 导出到文本来查看（详细参考 [6.13. Debugging the compiler](#)）。阅读这些中间表示可以深入学习 Haskell 的实现原理，同时对调试性能问题很有帮助，比如直接确认一些优化措施是否的确反映到了生成的代码中（`inlining`、`unboxing`、`let-floating`、`worker/wrapper` 变换，等等）。

在几种中间表示中，推荐阅读 STG。STG 在 ghc pipeline 中发挥承上启下的作用：STG 之前是 Core，这是一种带显式类型标注的“精简版”Haskell，而 STG 是特殊形式的 Core，通过 ANF 变换将 evaluation 顺序全部显式表明，STG 之后就是 C-like 的 Cmm，此时基本无法看出 Cmm dump 与原始 Haskell 之间的关联。

阅读 STG dump 并不需要许多预备的阅读材料：先看 [How to make a fast curry: push/enter vs eval/apply](#)（ghc 使用 eval/apply 机制），然后看 GHC API 文档中的 `StgSyn` 模块里，STG 的 AST 是如何定义的，结合 `-ddump-stg` 导出的 STG dump 进行阅读。下面是一个简单的例子：

Fact.hs 内容（包含一个简单的阶乘函数）

```
{-# OPTIONS_GHC -Wall -O2 -ddump-to-file -ddump-stg #-}

module Fact where

fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

运行 ghc Fact.hs 之后，查看 Fact.dump-stg。其中 fact 相关内容：

```
Fact.$wfact [InlPrag=[0], Occ=LoopBreaker]
  :: GHC.Prim.Int# -> GHC.Prim.Int#
[GblId, Arity=1, Caf=NoCafRefs, Str=<S,1*U>, Unf=OtherCon []] =
  \r [ww_s296]
    case ww_s296 of ds_s297 {
      __DEFAULT ->
        case -# [ds_s297 1#] of sat_s298 {
          __DEFAULT ->
            case Fact.$wfact sat_s298 of ww1_s299 {
              __DEFAULT -> *# [ds_s297 ww1_s299];
            };
          };
    };
```



```
Fact.fact [InlPrag=INLINE[0]] :: GHC.Types.Int -> GHC.Types.Int
[GblId,
  Arity=1,
  Caf=NoCafRefs,
  Str=<S(S),1*U(1*U)>m,
  Unf=OtherCon []] =
  \r [w_s29a]
    case w_s29a of {
      GHC.Types.I# ww1_s29c [Occ=Once] ->
        case Fact.$wfact ww1_s29c of ww2_s29d {
          __DEFAULT -> GHC.Types.I# [ww2_s29d];
        };
    };
};
```

结合 StgSyn 的类型定义，可以看出哪些门道呢？

- 与 Core 相比，STG 的 AST 上标注了许多用于 codegen 的额外信息，同时不再保证每个 binder 都带上类型标注；
- 整个 STG program 是一系列 StgTopBinding 的列表；
- 从 GenStgBinding 可以看出，STG 中的 binding 清晰地分为非递归和递归的情形，对应 ML-family 语言中的 let 和 letrec。所以 Haskell 实际上也是有 letrec 的（摊手）
- 一个 binding 将一系列 GenStgRhs 绑定到一系列变量。所以函数定义变成了 lambda 表达式绑定到函数名的形式；
- 绑定的 right hand side 中，表达式是不能任意嵌套的，我们已经可以从中看出 heap object 的创建和求值时机。比如要将一个函数或者 data constructor 给 apply 到一系列表达式的话，应用对象是 GenStgArg，对应 fast curry paper 中的 atom (literal/variable)，我们需要通过一系列 nested let 来将表达式绑定到变量，才能 apply。此时，一个 let expression 会创建一个 thunk 所对应的 heap object，将 heap object 的指针往 let expression 内部传递，而一个 case expression 会将一个 thunk 求值到 WHNF；
- case expression 必须是 exhaustive 的，覆盖所有 case。同时第一个 alternative 必须是内置的 __DEFAULT，原因是为美观考虑；
- 关于 # 结尾的常量、算符和 data constructor：GHC 的整数分为 boxed/unboxed integer，后一种就是对应 host platform 上的 machine int，而前一种则对应有一个 garbage-collected heap object。每个 heap object 的内存布局包含一个指向 info table 的指针以及 payload，其中 payload 部分保存着原本的 int。所有 kind 为 * 的 Haskell 类型，在内存中都使用类似的表示方式。# 结尾的常量和算符对应原生的常量和算符，像 -# [ds_s297 1#] 代表 (ds_s297 - 1)，GHC.Types.I# [ww2_s29d] 指的是将 ww2_s29d 代表的 unboxed int 重新装箱，case w_s29a of { GHC.Types.I# ww1_s29c .. } 指的是将 w_s29a 代表的 boxed int 求值到 WHNF，将求值后获得的 unboxed int 绑定到 ww1_s29c 继续之后的计算。
- 源代码中只有一个 fact 函数，这里为什么产生了 \$wfact 这个额外的函数？这是 ghc 中



了，所以 ghc 生成了一个对 unboxed int 进行快速阶乘的函数 \$wfact (worker)，而原本的 fact 函数只是一个调用 \$wfact，并负责参数与结果的 unboxing/boxing 处理的 wrapper 函数。这样一来，我们需要额外付出的开销只有开始计算时的一次 unboxing 和计算结束时的一次 boxing，中间和 C 是一样快的。为什么不在所有的地方都操作 unboxed 的值？考虑到 ghc 基于 boxing 的编译策略，unboxed 类型值的抽象能力是严重受限的，使用不便。

如果我们把 fact 的类型签名改得更加 polymorphic 一些，我们就能直观地感受到 Haskell 中的抽象并不免费，而伴随着性能开销这件事情了：

```
fact :: (Eq a, Num a) => a -> a
fact 0 = 1
fact n = n * fact (n - 1)
```

生成的 STG 代码：

```
Fact.fact [Occ=LoopBreaker]
  :: forall a. (GHC.Classes.Eq a, GHC.Num.Num a) => a -> a
[GblId,
 Arity=3,
 Caf=NoCafRefs,
 Str=<S(C(C(S))L),U(C(C1(U)),A)><S,U(A,C(C1(U)),C(C1(U)),A,A,A,C(U))><L,U>,
 Unf=OtherCon []] =
  \r [$dEq_s2eL $dNum_s2eM ds_s2eN]
    let {
      sat_s2e0 [Occ=Once] :: a_a2cU
      [LclId] =
        \u [] GHC.Num.fromInteger $dNum_s2eM lv1_r2dH;
    } in
    case GHC.Classes.== $dEq_s2eL ds_s2eN sat_s2e0 of {
      GHC.Types.False ->
        let {
          sat_s2eS [Occ=Once] :: a_a2cU
          [LclId] =
            \u []
              let {
                sat_s2eR [Occ=Once] :: a_a2cU
                [LclId] =
                  \u []
                    let {
                      sat_s2eQ [Occ=Once] :: a_a2cU
                      [LclId] =
                        \u [] GHC.Num.fromInteger $dNum_s2eM lv11_r2
```



```
    } in  GHC.Num.* $dNum_s2eM ds_s2eN sat_s2eS;  
    GHC.Types.True -> GHC.Num.fromInteger $dNum_s2eM lv11_r2eF;  
};
```

啊咧咧，没法自动进行 worker/wrapper 优化了。这段 STG 中有意思的地方：为什么 fact 好像多出了两个参数呢？新的两个参数名 `$dEq_s2eL`，`$dNum_s2eM`，从名字我们就能猜到是什么了——type class dictionary。GHC 中实现 type class 机制很简单粗暴，dictionary passing，每个 instance 实现都是一个 dictionary（没错，也是 heap object），然后函数参数包含 dictionary 来支持不同 instance。

使用 gdb 调试

你没有看错，ghc 编译的原生代码是可以使用 gdb 调试的！需要记住的几点：

- 编译选项记得带上 `-debug -g`。`-debug` 是 linker flag，链接 debugging rts，这个版本的 rts 是单线程的，并且带有更多的 sanity check，包含更多 symbol，方便调试。`-g` 会启用 DWARF 信息生成，其应用参考 [DWARF - GHC](#)
- Haskell 并不使用原生栈和调用约定。STG 编译到 Cmm 时，会使用一系列的虚拟寄存器（见 [Commentary/Rts/HaskellExecution/Registers - GHC](#)），这些寄存器使用 x86/x64 的通用寄存器是存不下的，所以它们实际保存在 BaseReg 结构体中，而 BaseReg 的位置保存在 `ebx` (x86) / `r13` (x64) 中。要查看 STG 虚拟寄存器的值，在 `{GHC_INSTALL_DIR}/lib/ghc-{GHC_VERSION}/include/DerivedConstants.h` 中可以获取 BaseReg 中不同 field 的 offset，然后从 `ebx/r13` 获取的地址加上 offset 读取即可。
- Haskell 也不直接使用原生 thread，而是先维护一个 thread pool，每个 worker thread 称为 Capability，然后每个 Capability 维护一个 Haskell thread 的队列，每个 Haskell thread 对应一个 TSO (Thread State Object)，每个 Haskell thread 有独立的栈，对应 STG 栈的一个片段，TSO 中的栈顶指针指向这个片段。调试链接了 threaded runtime 的程序之前，可以从 `includes/rts/storage/TSO.h` 看起。
- 关于整个 Haskell heap 的内存布局，参考 [Commentary/Rts/Storage/HeapObjects - GHC](#)

profiling

Cost-Centre Profiling 是 ghc 内置的 profiling 机制：在 Haskell 源代码中，可以使用 `SCC` pragma 将一个函数归类到一个 cost centre，然后给运行时传 `-p` 的选项，即可生成一个 profile 报告，列举每个 cost centre 的运行时间和内存分配数据。使用 `-fprof-auto` 编译选项，可以给每个函数自动插入 `SCC` 标记。



Cost-Centre 的用途是识别系统中的性能瓶颈。而针对另一种场景——比较同一功能的不同实现的性能，推荐使用 [criterion: a Haskell microbenchmarking library](#) 手动实现，criterion 的特色功能是自动反复执行 benchmark 并对数据进行统计分析生成报表，增加结果的可靠程度。

GHC eventlogs

ghc 的运行时还有一个 [event log](#) 机制，支持记录多种事件（如 gc 触发、线程 / spark 创建、进程调用等，详见 [GHC.RTS.Events](#)）。可以使用 [ghc-events](#) 库解析 event log 文件，或者使用 [threadscope](#) 可视化查看 event log。在调试 Haskell 程序的空间问题时，event log 非常有用。

编辑于 2017-04-24

「真诚赞赏，手留余香」

赞赏

还没有人赞赏，快来当第一个赞赏的人吧！

Haskell

GHC（编程套件）

函数式编程

▲ 赞同 57



● 3 条评论

➤ 分享

★ 收藏



文章被以下专栏收录



不动点高校现充部

一切与编程语言理论、函数式编程相关的杂谈。

已关注

推荐阅读



想象中的Haskell - Compiling

学 Haskell 果然是要趁早

几年前，length 的类型是介个样子
`length :: [a] -> Int` 现在你打开 GHCi 捏，是介个样子滴，（搜 FTP，并没有真相 `length :: Foldable t => t a -> Int` 当然了，这其实并不会...



Haskell开发环境配置



H

3 条评论

[⇌ 切换为时间排序](#)

nyycbd

1 年前

虽然不懂haskell，但是封面很好



草莓大福

1 年前

求 Reference (厚脸求



Felis sapiens (作者) 回复 草莓大福

1 年前

主要就是 ghc commentary 和文档

