GHC 源码剖析：代码生成

陈逸凡，吴昊泽

2017 年 6 月 1 日

1 概述

在前一篇提到 GHC 在经过简化和优化后得到一个语法更为简洁的中间语言 Core。Core 以 system F 为类型系统原型，将多态函数转换为在 dictionary 中查表得到对应的单态函数的函数，而且将 GHC 的初始类型的计算暴露出来。这样的语言很适合进行优化，但距离在一个命令式的随机访问内存机器上运行的代码还有很大差距。如何将一个惰性求值的函数式语言转换到一个立即求值的命令式语言上，成为代码生成阶段最大的问题。

2 Spineless Tagless G-machine

2.1 STG 设计思想

为了填补源语言与目标语言在执行方式上的巨大差距，Simon Peyton Jones 在 1992 年提出了 Spineless Tagless G-machine 这一抽象模型 [3]。

STG 包括一个抽象机器模型以及在这个抽象机器上执行的语言。STG 具有显式的堆和栈，和一组抽象寄存器：用堆对象表达运行中产生的待求值的式子，包含多个指针分别指向函数入口（entry code）、函数需要的参数和其他外部绑定（info table）；当且仅当 let 语句创建新的堆对象，当且仅当 case 语句对堆对象指定的表达式求值。这样就能用命令式的顺序执行方式表达惰性求值的语义。

2.2 STG 的语法

以下为 ghc/compiler/stgSyn/StgSyn.hs 文件中对 STG 抽象语法的定义（为简短考虑，删除大部分注释）。bndr 和 occ 在实际编译过程中，都是带定位信息的 Identifier 类型。

```
-- | A top-level binding.
data GenStgTopBinding bndr occ
  = StgTopLifted (GenStgBinding bndr occ)
  | StgTopStringLit bndr ByteString

data GenStgBinding bndr occ
  = StgNonRec bndr (GenStgRhs bndr occ)
  | StgRec    [(bndr, GenStgRhs bndr occ)]

data GenStgExpr bndr occ
  = StgApp
        occ             -- function
        [GenStgArg occ] -- arguments; may be empty
  | StgLit      Literal
  | StgConApp   DataCon
                [GenStgArg occ] -- Saturated
                [Type]          -- See Note [Types in StgConApp] in UnariseStg
  | StgOpApp    StgOp           -- Primitive op or foreign call
                [GenStgArg occ] -- Saturated.
                Type            -- Result type
                                -- We need to know this so that we can
                                -- assign result registers
  | StgLam
        [bndr]
        StgExpr    -- Body of lambda
  | StgCase
        (GenStgExpr bndr occ)  -- the thing to examine
        bndr        -- binds the result of evaluating the scrutinee
        AltType
```

```
      [GenStgAlt bndr occ]
                  -- The DEFAULT case is always *first*
                  -- if it is there at all
  | StgLet
      (GenStgBinding bndr occ)    -- right hand sides (see below)
      (GenStgExpr bndr occ)       -- body
  | StgLetNoEscape
      (GenStgBinding bndr occ)    -- right hand sides (see below)
      (GenStgExpr bndr occ)       -- body
  | StgTick
    (Tickish bndr)
    (GenStgExpr bndr occ)       -- sub expression
```

可以看出，STG 的定义是非常简洁直接的。有以下一些特性使得这个语言从 lambda 演算更偏向在一般硬件上执行的语言，暴露出更多的硬件特性。

- 大部分类型信息已经去除，因为内部类型安全已经由类型检查所保证，而传入数据的类型也由与外部交互的函数所检查。
- 所有表达式通过 ANF（administrative normal form）[1]。也就是说所有子表达式都被转化为 let 绑定并赋予了“名字”，这样也就能方便为所有子表达式的 thunk [2]分配内存了。
- 未完全填充的 lambda 表达式、值构造器和初始操作符（primitive operator）被通过 eta-expansion[3] 填充为参数完全填满的方式。这解决了部分应用（partial application）的问题：原本部分应用的操作符或结构体在经过 eta-expansion 后，被调用时一定都是参数补全的（saturated）。。
- 模式匹配被简化，一次只匹配一层值构造器（弱首范式求值）。同样有助于使模式匹配更统一更好实现。
- 算数表达式内部被拆箱计算，这样能加快计算过程。

[1] 关于 ANF 概念的具体定义，见维基百科A-normal form.

[2]thunk 可以认为是未求值的表达式

[3]关于 eta-expansion 定义，见 HaskellWiki Eta conversion

2.3 STG 运行时的操作语义

STG 语言本身只是表达了各个计算的“组装”方式。在抽象机器上运行时，实际上是要对入口的表达式做“求值”。求值时，代码指针跳到被求值式子的代码的入口，消耗参数，执行代码后，将返回值弹出。而在执行 let 语句时，则创建一个堆对象，包含一个指向函数入口的指针，然后将堆对象自身地址返回。

2.4 处理参数不匹配情形：push-enter 与 eval-apply[4]

前面提到 Haskell 作为一一个高阶非严格求值的语言，具有部分应用的语法特性，也就是一个多参函数在接受少于参数数量的参数后，会返回一个接受剩余参数并返回结果的函数。在经过处理后，一些部分应用函数会被编译器通过 eta-expansion 转换为参数补全的形式，部分应用时会返回一个函数指针。

但这样的转换仅仅对编译器能够知道函数位置和实际参数数量的函数起作用。对于下面这样的函数：

```
unknownApp :: (a -> b) -> a -> b
unknownApp f x = f x
```

编译器并不能确认 f x 是否是参数完全的，所以需要特殊的机制来处理这样的问题，GHC 中称之为 generic apply。

在 generic apply 的实现上，有两种不同的方法。一种被称为 push-enter，是 GHC 早期使用的技术。在这种方式下，会先把参数全部压入栈中，再直接调用目标函数，由函数决定接下来是直接返回结果还是将返回的函数应用到剩余参数中，或是返回参数不足的闭包。

而现在 GHC 所使用的是 eval-apply 方式。在生成的代码中，会先求函数的参数数量（这会被记录在函数的信息表中），和被传入的参数数量比较。若相等，则直接调用函数；若传入参数不足，则在栈上分配一个部分应用的闭包记录下被调用的函数和已经传入的参数，并将地址压回栈上；若传入参数过多，则会先将等于期望数量的参数传给函数，并将剩余参数以 continuation 方式放置，接下来先调用当前函数，执行完毕后应当返回一个

函数指针，再将之前压入的剩余参数作为被返回的函数的参数，调用该函数。

经过测试，后一种调用方式效率更高，而且能利用更多优化措施，所以在 GHC 中采用了后一种方式来做代码生成。

3 Cmm 语言：一种抽象的汇编语言 [2]

GHC 在优化生成 Core 中间语言后，先将 Core 进一步扫描转化为 STG 语言，再按上述的方法，将 STG 翻译为一个接近汇编语言风格的 Cmm 语言，再将 Cmm 语言翻译成目标码，包括 LLVM Bytecode、x86 汇编、C 语言等。之所以不直接生成汇编码，一是由于历史原因，最早的 GHC 将代码编译到 C 语言，再调用 C 编译器生成可执行文件，以利用已有的成熟的 C 编译器的优化能力，这也是当时当时函数式语言实现者都在尝试的做法；二是因为，Cmm 语言介于 C 和汇编语言之间，机器无关却又较好地描述了堆栈式机器的通用性质，具有相当的可移植性，对新平台做代码生成时，只需要修改 Cmm 语言到机器语言的翻译过程就可以了。

Cmm 语言使用数组来模拟堆栈操作，绕过了普通的 C 语言的参数传递机制，可以分配任意多本地变量。由于 Cmm 已经是完全的命令式严格求值语言，所以可以用常见的命令式语言的优化方式来对其做优化，包括控制流优化、重复子代码消除、代码结构优化等等。

在这之后，将 Cmm 代码生成到目标码，再插入运行时代码（包括任务调度、垃圾回收等），才能得到最终的可执行文件。

4 Pointer Tagging[1]

Pointer Tagging 也是 Haskell 代码生成中的重要一步。它的作用主要发生在运行时。由于在机器生成的代码需要按字对齐，所以指针的最后两到三位都是 0。于是可以利用这 2 3 bit 的空间来存储一些有效信息，包括函数的参数数量、生成的堆对象是否需要被垃圾回收器回收等等。这些工作只需要少量的标记，所以压缩在这几位中，能带来减少许多内存访问的操作、提高垃圾回收的效率等许多好处。

这对生成的代码的影响，就是每次解引用时，都需要对地址做一些处理。在代码生成时，就是简单的为所有解引用操作套上一个宏来处理，再在后续的预处理程序中改成普通代码。据测试，这一技术能显著提高 GHC 生成代码的效率。

参考文献

[1] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In ICFP '07: Proceedings of the ACM SIGPLAN international conference on Functional programming. ACM Press, October 2007.

[2] Dino Oliva, T. Nordin, and Simon Peyton Jones. C⁻: A portable assembly language. In Proceedings of the 1997 Workshop on Implementing Functional Languages, volume 1467, page 1 – 19. Springer Verlag, January 1997.

[3] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. Journal of Functional Programming, 2:127–202, July 1992.

[4] Simon Peyton Jones. How to make a fast curry: push/enter vs eval/apply. pages 4–15, September 2004.