GHC 源码剖析：代码生成

陈逸凡，吴昊泽

2017 年 5 月 30 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 [1]。

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 具有显式的堆栈空间，所有的求值和内存分配都由语言显示指定：用堆对象表达运行中产生的待求值的式子，包含指针指向函数和函数需要的参数和其他外部绑定；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。
- 所有参数都是原子（atom），这样避免了如何将一个未求值的表达式作为参数传给一个函数的问题。转化为将一个指向该表达式的指针传给函数，这样就能保持参数栈大小一致。
- 所有右值被提升为 let 绑定。这样能显式地为右值的表达式创建堆对象。
- 未完全填充的值构造器被通过 lambda 表达式填充。这样使得带值构造器的值在内存中的表示是一致的。
- 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。

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 push-enter 与 eval-apply[2]

在虚拟机器上执行 STG 代码时，存在两种不同的方法来处理求值前后值放置的位置。主要原因是参数及自由变量赋值的存储位置与返回值的存储位置同时放置在栈中时无法处理求值顺序与应用顺序不一致的问题。所以要么把参数赋值按函数的参数列表顺序压入栈中，而把函数返回值放置到另一个栈中；要么把参数赋值存放到堆中，而把函数返回值压入栈中。前者称为 push-enter 模式，后者称为 eval-apply 模式。

两种方法相比，push-enter 会导致栈增长规模不可预测，必须经常检查，而且栈上内容除对象指针外还有函数指针和字面值对于垃圾回收非常不友好；eval-apply 模式下，栈上只有对象指针，指针类型参数和字面值参数都在堆上，一致类型的内容既有利于避免错误，也有利于提高性能，当然也有不足，这一方法中参数栈是否填满是不可预测的。

GHC 曾经采用的是 push-enter 方法 [1]，但现在出于性能考量采用的是 eval-apply 模式 [2]。为了进一步提高 eval-apply 的性能，GHC 还采用了一个非常巧妙的称为 **“pointer tagging”** 的技巧。由于对象在堆中是对齐的，所以栈上存储的堆对象指针低 2 位或低 3 位是无用的，可以被利用来存储额外信息（包括参数数量、值构造器类型等），在很多方面优化了 GHC 生成的代码的性能。

3 GHC 的代码生成流程

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 只将代码编译到 C 语言的一个子集，再调用 C 编译器生成可执行文件；二是因为，Cmm 语言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。

参考文献

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

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