Compiling Programs in GHC: The Core and STG

petercommand

Overview

- How Haskell program is compiled in GHC
- GHC Core/STG
- STG Abstract Machine

GHC Compilation Process

- Haskell Source -> (Parser + Renamer + TypeChecker + Desugarer) ->
- Core -> (CorePrep + CoreToStg) ->
- Stg -> (CodeGen)
- Cmm

What is Core?

- A minimalistic language that Haskell compiles to
- Explicitly Typed
- Haskell without syntactic sugar
- Haskell -> desugar -> Core
- type CoreProgram = [CoreBind]

The Core Tree

```
type CoreProgram = [CoreBind]

type CoreBind = Bind CoreBndr

type CoreBndr = Var

data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]
```

- Later bindings in the list can refer to previous ones, but not vice versa

CoreSyn.hs

```
type CoreExpr = Expr Var

data Expr b  -- "b" for the type of binders,
  = Var  Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Type  Type
```

```
type Arg b = Expr b

type Alt b = (AltCon, [b], Expr b)

data AltCon = DataAlt DataCon | LitAlt  Literal | DEFAULT

data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]
```

Var Id — From Var.hs

```
type Id = Var -- Term level identifier

data Var = Id {   varName     :: !Name,
                  realUnique :: {-# UNPACK #-} !Int,
                  varType     :: Type,
               … } | …
```

Lambda Expr in Core

```
type CoreExpr = Expr Var

--                      argument    body

data Expr b = Lam b                 (Expr b)

            | …
```

- Includes types abstraction (Big lambdas)

Case Expression in Core

```
case x of y
    (Cons w ws) -> m
    DEFAULT -> n
```

- The DEFAULT case is always added in GHC, and removed if deemed unnecessary

Case Expr in Core

```
data Expr b = Case  (Expr b) b Type [Alt b]

            | …

type Alt b = (AltCon, [b], Expr b)
```

- Always strict on the scrutinee (whereas the “case” in Haskell is not always strict)

Lazy case example (Haskell):
```
case undefined of
    a -> case_body
```

Core Example — Strict

test :: (a, b) -> a

test (x, y) = x

becomes the following code in Core (ghc-core output)

```
test :: forall a_amS b_amT. (a_amS, b_amT) -> a_amS

test =  \ (@ a_ane) (@ b_anf)

              (ds_dpp :: (a_ane, b_anf)) ->

   case ds_dpp of _ { (x_an1, y_an2) -> x_an1 }
```

Core Example — Lazy

```
test :: (Num a) => (a, a) -> ((a, a) -> a) -> a

test x f = case x of x -> f x -- test x f = f x

test :: forall a_amU.  Num a_amU => (a_amU, a_amU) ->
                  ((a_amU, a_amU) -> a_amU) -> a_amU

test = \ (@ a_aos) _  (x_ao0 :: (a_aos, a_aos))
           (f_ao1 :: (a_aos, a_aos) -> a_aos) ->
                f_ao1 x_ao0
```

Let Expr in Core

```
data Expr b = Let   (Bind b) (Expr b) | …

data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]
```

Non-recursive Let

```
let x = 1
    y = 2
in x + y
```

Non-recursive Let

```
let x = 1
in let y = 2
   in x + y
```

Recursive Let

- A let is considered a recursive let if any RHS of the let bindings references a LHS of the let bindings

fix f = let x = f x in x

STG

- Spineless
- Tagless
- G-machine

STG

- Spineless Tagless G-Machine
- Very similar to Core
- More annotations
- RHS are Closures (with and without arguments) and Constructors

STG

- Constructor, FFI, and primitive operators are always saturated (through eta transform)
- Binary applications are combined ex: ((f a) b)
- Closures are decorated with free variables (pretty print does not print this by default) (-ddump-stg)
- A tree of [StgBinding]

Eta Transformation

```
map f => \x -> map f x

(+) => \x -> \y -> (+) x y
```

- Arguments generated through eta transform in ghc have names like eta_XXX in Core/Stg

GenStg* Type Synonyms

```
type StgBinding  = GenStgBinding  Id Id

type StgArg      = GenStgArg Id

type StgLiveVars = GenStgLiveVars Id

type StgExpr     = GenStgExpr Id Id

type StgRhs      = GenStgRhs Id Id

type StgAlt      = GenStgAlt Id Id
```

STGSyn.hs

- Both bindr and occ are bound to Id

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

STGSyn.hs

data GenStgRhs bndr occ

= StgRhsClosure { … } — Closure

| StgRhsCon { … } — Constructor

Update Flag

- Used to see if a thunk only needs to be evaluated once
- Does not apply to functions / constructors

STGSyn.hs

```
data GenStgRhs bndr occ

    = StgRhsClosure

        [occ] -- Free Variables

        !UpdateFlag -- ReEntrant | Updatable | SingleEntry
-- ReEntrant and SingleEntry are non-updatable closures

        [bndr] -- Arguments (can be empty)

        (GenStgExpr bndr occ)   -- body | …
```

STGSyn.hs

```
data GenStgRhs bndr occ =
  StgRhsCon
           DataCon          -- constructor
           [GenStgArg occ]  -- args
    | …
```

STG Expr

```
data GenStgExpr bndr occ = …
```

StgApp

```
StgApp
    occ                  -- function
    [GenStgArg occ]  -- arguments; may be empty
```

StgLit

StgLit   Literal

- Char, String, Int, …

StgConApp

StgConApp   DataCon [GenStgArg occ]

- The constructor application should already be saturated after transforming to STG

StgOpApp

StgOpApp

StgOp -- Primitive op or foreign call

[GenStgArg occ] -- Saturated

Type -- Result type (used to assign register)

StgLam

StgLam

[bndr]

StgExpr -- Body of lambda

StgCase

```
type GenStgAlt bndr occ
        = (AltCon, [bndr], -- constructor's parameters,
           [Bool], -- True if the parameter is used in RHS
           GenStgExpr bndr occ)
type GenStgLiveVars occ = UniqSet occ
```

```
StgCase
    (GenStgExpr bndr occ) -- the thing to examine
    (GenStgLiveVars occ) -- whole case live vars
    (GenStgLiveVars occ) - - rhs live vars
    bndr
    AltType -- result type
    [GenStgAlt bndr occ] -- branches of case
```

StgLet

StgLet

(GenStgBinding bndr occ) - - bindings

(GenStgExpr bndr occ) -- body

STG Abstract Machine

- The operational semantics of STG
- Defines how the STG machine is evaluated

STG Abstract Machine

- Code
- Argument stack -- values
- Return stack -- continuations
- Update stack -- update frames
- Heap -- closures
- Global Environment -- gives addresses of top-level closures

Closures

(vs \pi xs -> e) ws

free variables update arguments

(vs \pi xs ->

closure body free_arguments

e) ws

Closure

- When evaluating a closure, STG pushes the corresponding arguments into the stack, and jumps directly to the entry code of the closure — closures has a uniform representation

- The update of a thunk is done inside the closure — “self-updating”

Code

- Eval e rho
    - Evaluate expression e in environment rho
- Enter a
    - Apply a to the arguments on the argument stack
- ReturnCon c ws
    - Return the constructor c applied to values ws to the continuation on the return stack
- ReturnInt k
    - Return the primitive integer k to the continuation on the return stack

Values

- Addr a -- heap address (points to closures)
- Int n -- primitive integer value

Function val

- rho is used as the global environment
- sigma is the local environment
- If k is a literal..

```
val rho sigma k = Int k -- returns the primitive value k
```

- If v is a variable..

```
val rho sigma v = rho v (if v belongs to rho)
                = sigma v (otherwise)
```

Initial State

Code	Arg Stack	Return Stack	Update Stack	Heap	Global
Eval (main {}) {}	{}	{}	{}	h_init	sigma

Since Global does not change at all,
we will be omitting it from now on

Initial State — Global Bindings

g1 = vs1 \pi1 xs1 -> e1

…

gn = vsn \pi2 xs2 -> e2

- Every binding (g1 … gn) binds to an address
- One of the bindings is main

Initial State

```
sigma = [g1 -> (Addr a1),
           …
         gn -> (Addr an),
]
h_init = [a1 -> (vs1 \pi1 xs1 -> e1) (rho vs1),
             …
             an -> (vsn \pin xsn -> en) (rho vsn)
]
```

Application

Code (val rho sigma f = Addr a)	Arg Stack	Return Stack	Update Stack	Heap
Eval (f xs) rho	as	rs	us	h
=>				
Enter a	val rho sigma xs ++ as	rs	us	h

Application (Saturated)

Code	Arg Stack	Return Stack	Update Stack	Heap
(length as >= length xs) Enter a	as	rs	us	h[a -> (vs \n xs -> e) ws_f]
=>				
Eval e rho	as’	rs	us	h

where

ws_a ++ as’ = as

length(ws_a) = length(xs)

rho = [vs ->ws_f, xs -> ws_a]

let

Code	Arg Stack	Return Stack	Update Stack	Heap

Eval (
let
    x1 = vs1 \pi1 xs1 -> e1
    xn = vsn \pin xsn -> en …
in e) rho
	as	rs	us	h

=>				
Eval e rho’	as	rs	us	h’

where
rho’ = rho[x1 -> Addr a1 …, xn -> Addr an]
h’ = h[a1 -> (vs1 \pi1 xs1 -> e1) (rho_rhs vs1),
        …
        an -> (vsn \pin xsn -> en) (rho_rhs vsn)]
rho_rhs = rho

letrec

Code	Arg Stack	Return Stack	Update Stack	Heap

Eval (
let
   x1 = vs1 \pi1 xs1 -> e1
   xn = vsn \pin xsn -> en …
in e) rho
	as	rs	us	h

=>				
Eval e rho’	as	rs	us	h’

where

rho’ = rho[x1 -> Addr a1 …, xn -> Addr an]

h’ = h[a1 -> (vs1 \pi1 xs1 -> e1) (rho_rhs vs1),

        …

        an -> (vsn \pin xsn -> en) (rho_rhs vsn)]

rho_rhs = rho’

case e of alts

Code	Arg Stack	Return Stack	Update Stack	Heap
Eval (case e of alts) rho	as	rs	us	h
=>				
Eval e rho	as	(alts, rho):rs	us	h

Evaluate e, and push (alts, rho) into the Return Stack

Constructor Application

Code	Arg Stack	Return Stack	Update Stack	Heap
Eval (c xs) rho	as	rs	us	h
=>				
ReturnCon c (val rho sigma xs)	as	rs	us	h

Constructor Matching

Code	Arg Stack	Return Stack	Update Stack	Heap
ReturnCon c ws	as	(…;c vs -> e;…, rho):rs	us	h
=>				
Eval e rho[vs -> ws]	as	rs	us	h

Constructor Matching

Code	Arg Stack	Return Stack	Update Stack	Heap
ReturnCon c ws	as	(…;v -> e_d, rho):rs	us	h
=>				
Eval e rho[v -> a]	as	rs	us	h’

h’ = h[a -> (vs \n {} -> c vs) ws]

Primitives

Code	Arg Stack	Return Stack	Update Stack	Heap
Eval k rho or Eval (f {}) rho[f -> Int k]	as	rs	us	h
=>				
ReturnInt k	as	rs	us	h

Pattern Matching for Primitives

Code	Arg Stack	Return Stack	Update Stack	Heap
ReturnInt k	as	(…;k->e;…, rho):rs	us	h
=>				
Eval e rho	as	rs	us	h

Pattern Matching for Primitives

Code	Arg Stack	Return Stack	Update Stack	Heap
ReturnInt k -- (k /= ki, for i = 1..n)	as	(k1->e1; …; kn->en; x->e, rho):rs	us	h
=>				
Eval e rho[x -> Int k]	as	rs	us	h

Pattern Matching for Primitives

Code	Arg Stack	Return Stack	Update Stack	Heap
ReturnInt k (k /= ki, for i = 1..n)	as	(k1->e1; …; kn->en; DEFAULT->e, rho):rs	us	h
=>				
Eval e rho	as	rs	us	h

Primitive Operator

Code	Arg Stack	Return Stack	Update Stack	Heap
Eval (op {x1, x2}) rho[x1-> Int i1, x2 -> Int i2]	as	rs	us	h
=>				
ReturnInt (i1 `op` i2)	as	rs	us	h

Updates

- When the update flag in closure is marked as true, then we will update the closure when its value is needed
- If the thunk is only evaluated once, we don't need to update the pointer pointing to the thunk
- The heap is updated with the actual value

Entering Updatable Closure

Code	Arg Stack	Return Stack	Update Stack	Heap
Enter a	as	rs	us	h[a -> (vs \u {} -> e) ws_f]
=>				
Eval e rho	{}	{}	(as, rs, a):us	h

where
rho = [vs -> ws_f]

Performing The Actual Update — ReturnCon

Code	Arg Stack	Return Stack	Update Stack	Heap
ReturnCon c ws	{}	{}	(as_u, rs_u, a_u):us	h
=>				
ReturnCon c ws	as_u	rs_u	us	h_u

where
vs is a sequence of arbitrary distinct variables
length vs = length ws
h_u = h[a_u -> (vs \n {} -> c vs) ws]

Performing The Actual Update
Partial Application

h a = (vs \n xs -> e) ws_f

length as < length xs

Code	Arg Stack	Return Stack	Update Stack	Heap
Enter a	as	rs	(as_u, rs_u, a_u):us	h
=>				
Enter a	as ++ as_u	rs_u	us	h_u

where

xs_1 ++ xs_2 = xs

length xs_1 = length as

h_u = h[a_u -> (vs ++ xs_1 \n xs_2 -> e) (ws_f ++ as)]

Performing The Actual Update Partial Application (PAP Closure)

Code	Arg Stack	Return Stack	Update Stack	Heap
h a = (vs \n xs -> e) ws_f length as < length xs Enter a	as	rs	(as_u, rs_u, a_u):us	h
=>				
Enter a	as ++ as_u	rs_u	us	h_u

where

xs_1 ++ xs_2 = xs

length xs_1 = length as

h_u = h[a_u -> (f:xs_1 \n {} -> f xs_1) (a:as)]

Partial Application

- A closure is created for every “arguments number” in partial applications (common cases)

- If the partial application uses a lot of arguments, a combination of multiple closure that consists of smaller number of arguments is used

Q&A