SHOW CASE: STG BACKEND FOR IDRIS

MY BACKGROUND

- Interested in Software Development
- Interested in Computer Science
- Academy: No PhD
- Past: Software Developer C# Java
- Past: QA Test Automation Engineer
- Present: Haskell Developer
- Present: Learning Idris
- Misc: Handpan

OUTLINE

- Foundations
- Spineless Tagless G-machine
- External STG
- Idris Intermediate Representations (IR)
- CompileData API
- Compile IR
- Ideas, Future Work

STEP 1: FOUNDATIONS

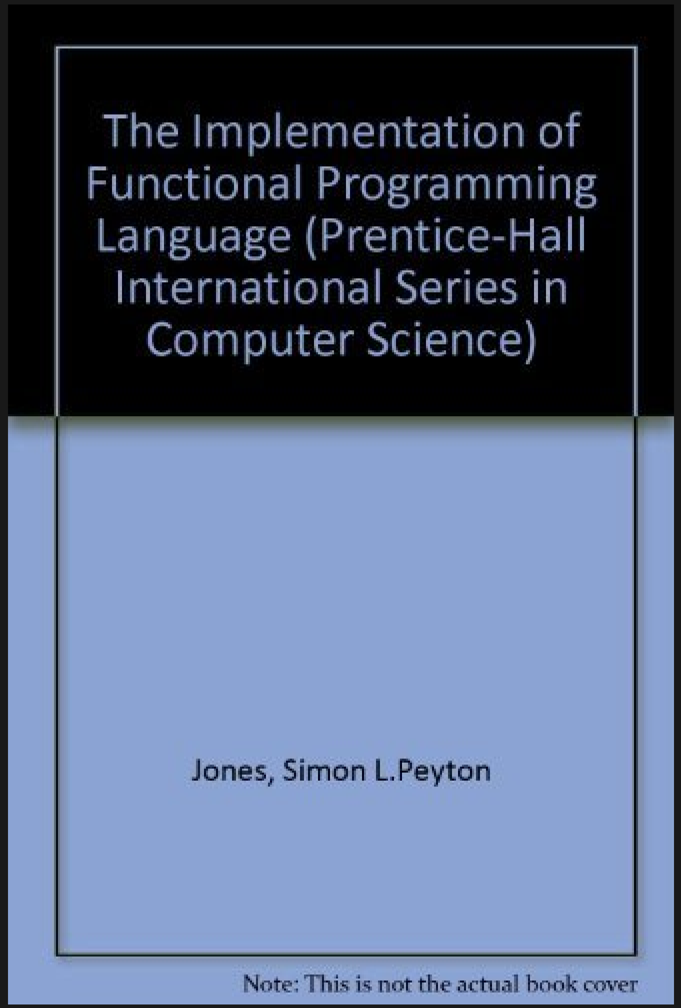
- Lambda calculus: (\x => x) y
- Extensions to LC, such as
  - Let bindings: let x = y in z
  - Primitive values/types: 3 Int
  - Constructors: Pair 'a' 1
  - Function Call: printLn "3"
  - Case expressions: case x of { Pair 'a' 1 => ... }

THE IDRIS BACK-END I WORK ON

- GHC Haskell with a twist
- Goal: Learn the internals of the Idris compiler
- Goal: Learn the internals of the GHC compiler
- Goal: Interop between Idris and Haskell libraries

GHC HASKELL: SPINELESS TAGLESS G-MACHINE

Simon Peyton Jones

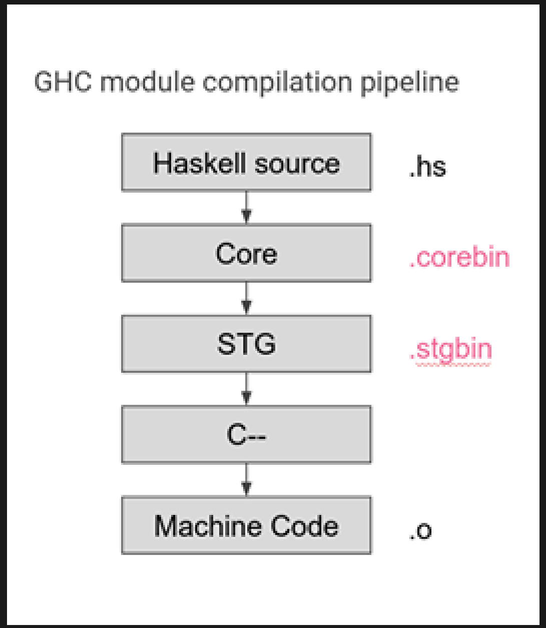


GHC HASKELL WITH A TWIST

ExtSTG and STG interpreter

- Architecture of GHC Haskell/Core/STG/Cmm/RTS
- How to use STG to generate executable via STG
- What is ExtSTG and GHC-WPC

Architecture of GHC Haskell



What is ExtSTG

- External-STG GHC independent STG representation.
- External-STG Interpreter Interpreter for the STG. Operational semantics of STG written in Haskell.
- External-STG Compiler Soon it will uses GHC 9.0 as a code generator, processes External-STG, outputs executable.

HOW TO USE STG TO GENERATE EXECUTABLE VIA STG

Convert from Ext-STG to GHC-STG

```
toStg :: Ext.Module -> StgModule
toStg Ext.Module{..} = stgModule where
  (topBindings, Env{..}) = flip runState (emptyEnv moduleUnitId moduleName) $ do
    setAlgTyCons algTyCons
    mapM cvtTopBinding moduleTopBindings
```

compileProgram from Ext-STG

```
compileProgram :: Backend -> [String] -> [String] -> [String] -> [String] -> ForeignStubs -> [TyCon] -> [StgTopBinding] -> IO ()
compileProgram backend incPaths libPaths ldOpts clikeFiles stubs tyCons topBinds_simple = runGhc (Just libdir) $ do
  dflags <- getSessionDynFlags

  liftIO $ do
```

STG EXPRESSIONS

```
-- STG-Def
data Arg
  = StgVarArg BinderId
  | StgLitArg Lit
```

```
-- STG-Def
data Expr
  = StgApp          BinderId (List Arg)
  | StgLit          Lit
  | StgConApp       DataConId (List Arg)
  | StgOpApp        StgOp (List Arg)
  | StgCase         Expr SBinder (List Alt)
  | StgLet          Binding Expr
```

```
record Alt where
  constructor MkAlt
  Con      : AltCon
  Binders  : List BinderId
  RHS      : Expr
```

STEP 2: SELECT AN IDRIS IR

- LiftedDef: Lambda lifted form, local functions transformed to global ones.
- ANFDef: Explicit variable names, intermediate expression turned into let binding, every argument is a variable.
- VMDef: Needs representation of closures, an apply function which applies an argument, also partial application.

MY IDRIS IR CHOICE

```
data ANF : Type where
  AV          : AVar -> ANF
  AAppName    : Name -> List AVar -> ANF
  AUnderApp   : Name -> (missing : Nat) -> (args : List AVar) -> ANF
  AApp        : (closure : AVar) -> (arg : AVar) -> ANF
  ALet        : (var : Int) -> ANF -> ANF -> ANF
  ACon        : Name -> (tag : Maybe Int) -> List AVar -> ANF
  AOp         : PrimFn arity -> Vect arity AVar -> ANF
  AExtPrim    : Name -> List AVar -> ANF
  AConCase    : AVar -> List AConAlt -> Maybe ANF -> ANF
  AConstCase  : AVar -> List AConstAlt -> Maybe ANF -> ANF
  APrimVal    : Constant -> ANF
  AErased     : ANF
  ACrash      : String -> ANF
```

```
-- STG-Def
data Expr
  = StgApp         BinderId (List Arg)
  | StgLit         Lit
  | StgConApp      DataConId (List Arg)
  | StgOpApp       StgOp (List Arg)
  | StgCase        Expr SBinder (List Alt)
  | StgLet         Binding Expr
```

STEP 3: HOW TO USE COMPILER BACK-END API?

```
main : IO ()
main = mainWithCodegens [("stg", stgCodegen)]

compile
  : Ref Ctxt Defs -> (tmpDir : String) -> (outputDir : String) -> ClosedTerm -> (outfile : String)
  -> Core (Maybe String)
compile defs tmpDir outputDir term outfile = do
  coreLift $ putStrLn "Compile closed program term..."
  cdata <- getCompileData ANF term
  stgs  <- compileModule $ anf cdata
  let res = show $ toJSON stgs
  let out = outputDir  outfile
  Right () <- coreLift $ writeFile out res
  pure (Just out)
```

STEP 4: HOW TO COMPILE IDRIS IR TO SOMETHING?

- How to represent primitive values?
- How to represent Algebraic Data Types?
- How to implement special values?
- How to implement primitive operations?
- How to compile IR expressions?
- How to compile Definitions?
- How to implement Foreign Function Interface?
- How to compile modules?
- How to embed code snippets?
- What should the runtime system support?

NOTATIONS

```
-- Haskell - Haskell syntax
newtype SourceLoc = SourceLoc (Int, Int)
```

```
-- STG-Def - Idris syntax for some Ext-STG
data PrimRep
  = ...
  | UnliftedRep -- Boxed, in WHNF
```

```
-- Idris compiler - Idris syntax, internals of the Idris compiler
data ANF : Type where
  ...
  ACon : Name -> (tag : Maybe Int) -> List AVar -> ANF
```

How to represent primitive values?

- Primitive values defined in `Core.TT.Constant`
- Int, Integer, Bits, Char, String, Double, World
- And their type counterpart: IntType, IntegerType, ...

```haskell
-- Idris compiler
data Constant
  = I Int                 | IntType
  | BI Integer            | IntegerType
  | B8 Int                | Bits8Type
  | B16 Int               | Bits16Type
  | B32 Int               | Bits32Type
  | B64 Integer           | Bits64Type
  | Str String            | StringType
  | Ch Char               | CharType
  | Db Double             | DoubleType
  | WorldVal              | WorldType
```

HIGH LEVEL HASKELL

```haskell
-- Haskell
data IdrInt    = IdrInt    Int8#
data IdrDouble = IdrDouble Double#
data IdrChar   = IdrChar   Char#
data IdrStr    = IdrStr    Addr#
data IdrWorld  = IdrWorld
```

STG DATACON

```
-- STG-Def
data PrimRep
  = ...
  | LiftedRep   -- Boxed, in thunk or WHNF
  | UnliftedRep -- Boxed, in WHNF
  | Int64Rep    -- Unboxed, Signed, 64 bits value (with 32 bits words only)
  | Word64Rep   -- Unboxed, Unisgned, 64 bits value (with 32 bits words only)
  | WordRep     -- Unboxed, Unisgned, word-sized value
  | DoubleRep
```

This is how BOXED values are created in STG:

```
-- STG-Def
example : DataCon
example = MkDataCon "IdrInt" (MkDataConId (MkUnique 'u' 0)) [Int64Rep]
```

```
-- STG-Def
data Unique = MkUnique Char Int
data DataConId = MkDataConId Unique

data DataConRep = AlgDataCon (List PrimRep)

record DataCon where
  constructor MkDataCon
  Name    : String
  Id      : DataConId
  Rep     : DataConRep
```

HOW TO REPRESENT ALGEBRAIC DATA TYPES?

Data types in Idris can be defined via `data` and `record`. See previous slide.

```
-- Idris compiler
data ANF : Type where
  ...
  ACon : Name -> (tag : Maybe Int) -> List AVar -> ANF
  ...
```

HASKELL

```haskell
-- Haskell
data Triplet a b c = Triplet a b c
```

STG DATACON

```
-- STG-Def
example : DataCon
example = MkDataCon "Triplet" (MkDataConId (MkUnique 'u' 0)) [UnliftedRep, UnliftedRep, UnliftedRep]
```

```
-- STG-Def
data PrimRep
  = ...
  | LiftedRep   -- Boxed, in thunk or WHNF
  | UnliftedRep -- Boxed, in WHNF
```

```
-- STG-Def
record DataCon where
  constructor MkDataCon
  Name    : String
  Id      : DataConId
  Rep     : DataConRep
```

How to implement special values?

- Idris type constructors
- Erased value

```
-- Idris compiler
data ANF : Type where
  ...
  ACon : Name -> (tag : Maybe Int) -> List AVar -> ANF
  AErased : ANF
  ...
```

HASKELL

```haskell
-- Haskell
data IdrisTypes
  = IdrIntT
  | IdrList IdrisType
  | IdrMaybe IdrisType
  ...

data Erased = Erased
```

STG DATACON

```
-- STG-Def
example1 : DataCon
example1 = MkDataCon "IdrList" (MkDataConId (MkUnique 'u' 0)) [UnliftedRep]

example2 : DataCon
example2 = MkDataCon "Erased" (MkDataConId (MkUnique 'u' 0)) []
```

```
-- STG-Def
record DataCon where
  constructor MkDataCon
  Name    : String
  Id      : DataConId
  Rep     : DataConRep
```

Association between data and type constructors.

STG code generator needs types associated with data constructors.

```
-- Idris compiler
data Def : Type where
  DCon : (tag : Int) -> (arity : Nat) -> Def
  TCon : (tag : Int) -> (arity : Nat) -> ... -> (datacons : List Name) -> Def
```

But Def is not at ANF level, it needs some magic. (Details in the implementation)

```
-- STG-Def
data TyConId = MkTyConId Unique

record STyCon where
  constructor MkSTyCon
  Name     : Name
  Id       : TyConId
  DataCons : (List SDataCon)
```

How to implement primitive operations?

- Arithmetic operations (Add, Sub, Mul, Div, Mod, Neg)
- Bit operations (ShiftL, ShiftR, BAnd, BOr, BXor)
- Comparing values (LT, LTE, EQ, GTE, GT)
- String operations (Length, Head, Tail, Index, Cons, Append, Reverse, Substr)
- Double precision floating point operations (Exp, Log, Sin, Cos, Tan, ASin, ACos, ATan, Sqrt, Floor, Ceiling)
- Casting of numeric and string values
- BelieveMe: This primitive helps the type checker. When the type checker sees the `beleive_me` function call, it will cast type `a` to type `b`. For details see below.
- Crash: The first parameter of the crash is a type, the second is a string that represents the error message.

Primitive values are boxed we need to unbox them

```haskell
-- Haskell
idrAddPrim (IdrInt a) (IdrInt b) = IdrInt (+# a b)
```

```haskell
-- Haskell
idrAddPrim a b =
  case a of
    (IdrInt au) -> case b of
      (IdrInt bu) -> IdrInt (+# au bu)
```

It seems complex, but contains most of the STG Expression constructions

```
-- STG-Def
StgCase (StgApp a) (AlgAlt (TypeConId a))
  [ MkAlt (AltDataCon (DataConId a)) (mkBinder au)
      (StrCase (StgApp b)) (AltAlt (TypeConId b))
        [ MkAlt (AltDataCon bDataConId) (mkBinder bu)
            (StgCase (StgOpApp "+#" au bu) (mkBinder x)
              [ MkAlt AltDefault (StgConApp (DataCon IdrInt) (StgVarArg x)) ]
        ]
  ]
```

```
-- STG-Def
data Expr
  = StgApp     BinderId (List Arg)
  | StgLit     Lit
  | StgConApp DataConId (List Arg)
  | StgOpApp  StgOp (List Arg)
  | StgCase   Expr SBinder (List Alt)
  | StgLet    Binding Expr
```

```
-- STG-Def
record Alt where
  constructor MkAlt
  Con      : AltCon
  Binders : List BinderId
  RHS      : Expr
```

HOW TO COMPILE IR EXPRESSIONS?

Schematics of ANF to STG mapping

```
-- Idris compiler          -- STG-Def
AV          v           ~> StgApp    v  []
AAppName    n vs        ~> StgApp    n  vs
AUnderApp   n x vs      ~> StgApp    n  vs
AApp        v1 v2       ~> StgApp    v1 v2
ALet        v a1 a2     ~> StgLet    (StgNonRec v (anf v1)) (anf v2)
ACon        m tag vs    ~> StgCon    m vs
AOp         primop as   ~> StgOpApp  (anfprim primop) as
AExtPrim    n vs        ~> StgApp    n vs -- TODO
AConCase    v alts      ~> StgCase   (StgApp v []) x (tagAlts alts)
AConstCase  v alts      ~> StgCase   (StgApp v []) x (constAlts alts)
APrimVal    c           ~> StgCon    "Idr..." (StgLitArg c)
AErased                 ~> StgCon    "Erased" []
ACrash      msg         ~> StgApp    "error" [msg]
```

Many mappings needs to be implemented, eg: (local)variables, data constructors...

HOW TO COMPILE DEFINITIONS?

```
-- Idris compiler
record CompileData where
  constructor MkCompileData
  ...
  anf : List (Name, ANFDef)

data ANFDef : Type where
  MkAFun     : (args : List Int) -> ANF -> ANFDef
  MkACon     : (tag : Maybe Int)   -> (arity : Nat) -> ANFDef
  MkAForeign : (ccs : List String) -> (fargs : List CFType) -> CFType -> ANFDef
  MkAError   : ANF -> ANFDef
```

```
-- STG-Def
data Rhs
  = StgRhsClosure UpdateFlag (List SBinder) Expr
  | StgRhsCon     DataCon    (List Arg)

data Binding    = StgNonRec    SBinder Rhs
data TopBinding = StgTopLifted Binding
```

```
-- Idris compiler       -- STG-Def
MkAFun arg body      ~> StgTopLifted (StgNonRec name (StgRhsClosure ReEntrant (anf body)))
MkACon tag arity     ~> -- Redundant information in ANF
MkAForeign css ts t  ~> StgTopLifted (StgNonRec name (StgRhsClosure ReEntrant (foreign css)))
MkAError body        ~> StgTopLifted (StgNonRec name (StgRhsClosure ReEntrant (anf body)))
```

HOW TO IMPLEMENT FOREIGN FUNCTION INTERFACE? - TYPES

```
-- Idris-compiler
data CFType : Type where
  CFUnit : CFType
  CFInt : CFType
  CFUnsigned8 : CFType
  CFUnsigned16 : CFType
  CFUnsigned32 : CFType
  CFUnsigned64 : CFType
  CFString : CFType
  CFDouble : CFType
  CFChar : CFType
```

```
CFPtr : CFType
CFGCPtr : CFType
CFBuffer : CFType -- Random access array
CFWorld : CFType  -- Token for IO computations
CFFun : CFType -> CFType -> CFType  -- Callbacks
CFIORes : CFType -> CFType          -- Effectful
CFStruct : String -> List (String, CFType) -> CFType -- C-Struct
CFUser : Name -> List CFType -> CFType -- User defined type
```

Represented as Boxed data in STG, similar to primops

HOW TO IMPLEMENT FOREIGN FUNCTION INTERFACE? - FFI STRING

```
-- Idris example
%foreign "C:idris2_putStr,libidris2_support"
         "node:lambda:x=>process.stdout.write(x)"
         "stg:base:Prelude.putStr"
prim__putStr : String -> PrimIO ()
```

User defines the STG function with its Haskell module path.

```
-- STG-Def
MkAForeign css ts t  ~> StgTopLifted (StgNonRec "prim__putStr" ... (StgApp "base:Prelude.putStr" as))
```

Differences between strict and lazy representation van be controlled via an analysis, inserting LiftedRep and UnliftedRep in the Binders.

How to compile modules?

- CompileData contains all the functions.
- STG-Backend currently compiles into one big STG module

How to embed code snippets?

- FFI
- Externals
- Elaboration: Think of them like macros, or like template Haskell.

What should the runtime system support?

- Boxed primitive values
- Data constructors
- Data and type constructor association
- Memory management
- Threading

STEP 5: ENJOY YOUR IDRIS PROGRAM!

FUTURE WORK

- Separate STG module generation, compilation
- Better ADT mapping in FFI
- Threading

CONCLUSION

- Compilation of a lambda calculus like language
- Values should be represented as boxed
- Dynamic memory management is needed, modern languages have it
- Mix and match of different library components
- FFI can get tricky for non owned libraries
- No need for full implementation if Idris is meant to be used as a strong DSL

QUESTIONS?