STG BACKEND FOR IDRIS2

Andor Penzes

MY BACKGROUND

- Interested in Software Development
- Interested in Computer Science
- Past: Software Developer C# Java
- Past: QA Test Automation Engineer
- Present: Haskell Developer
- Present: Learning Idris

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

OUTLINE

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

STEP 1: FOUNDATIONS OF COMPILER INTERNALS

- $(\lambda x.\, x) y$
- Extensions
  - $let\ x = y\ in\ z$
  - $3 :: Int$
  - $Pair\ 'a'\ 1$
  - $printLn\ \text{”}\ 3\ \text{”}$
  - $case\ x\ of \langle Pair\ a\ b \rightarrow \ldots \rangle$

GHC HASKELL AND SPINELESS TAGLESS G-MACHINE

Making a fast curry: push/enter vs. eval/apply for higher-order languages

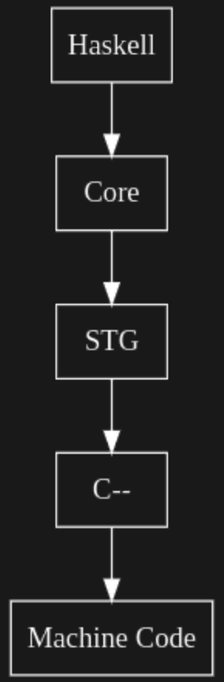
Simon Marlow and Simon Peyton Jones

GHC HASKELL WITH A TWIST

ExtSTG and STG interpreter, work of Csaba Hruska

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

ARCHITECTURE OF GHC HASKELL



WHAT IS EXT-STG

- 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 use GHC 9.0 as a code generator, processes External-STG, outputs executable.

HOW TO USE EXT-STG TO GENERATE EXECUTABLE

Convert from Ext-STG to GHC-STG

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

Compile program from GHC-STG

```haskell
-- Haskell
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
      ...
```

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: Introduce a variable for each non-trivial intermediate expression. Each argument of a function is a variable or constant.
- VMDef: Virtual Machine, needs representation of closures in registers, partial and non-partial function application.

MY IDRIS IR CHOICE

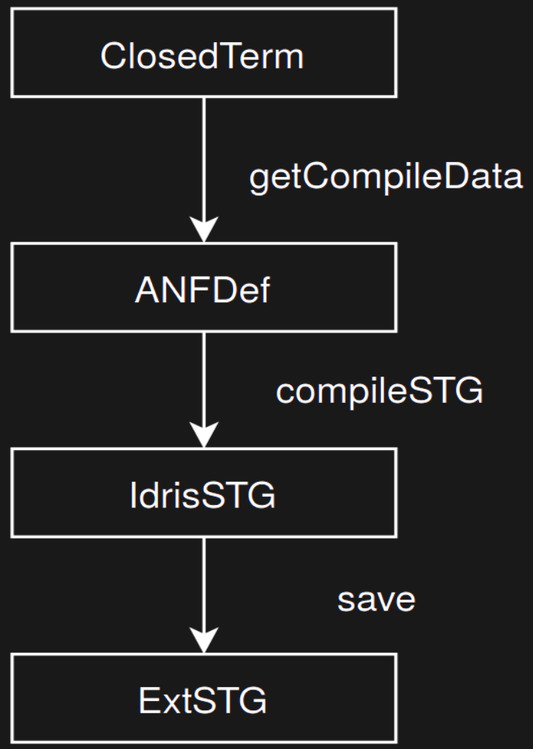
```
data ANF : Type where
  AV          : AVar -> ANF
  AAppName    : Name -> List AVar -> ANF
  AUnderApp   : Name -> (missing : Nat) -> (args : List AVar) -> ANF
  AApp        : (closure : AVar) -> (arg : AVar) -> ANF
  ALet        : (var : Int) -> ANF -> ANF -> ANF
  ACon        : Name -> (tag : Maybe 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 = ...
```



INTERLUDE

NOTATIONS

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

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

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

EXAMPLE: ADT IN STG

```haskell
-- Haskell
data List a = Nil | Cons a (List a)
```

```haskell
-- STG-Def
record STyCon where
  constructor MkSTyCon
  Name     : String
  Id       : TyConId
  DataCons : (List SDataCon)

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

```haskell
-- STG-Def
lazyList : STyCon
lazyList = MkSTyCon "List"
  [ MkDataCon "Nil" []
  , MkDataCon "Cons" [LiftedRep, LiftedRep]
  ]

strictList : STyCon
strictList = MkSTyCon "List"
  [ MkDataCon "Nil" []
  , MkDataCon "Cons" [UnliftedRep, UnliftedRep]
  ]
```

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?

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, ...

```
-- 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 data are declared in STG

```
-- STG-Def
idrInt : STyCon
idrInt = MkSTyCon "IdrInt"
  [ MkDataCon "IdrInt" [Int64Rep]
  ]
```

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

HOW TO REPRESENT ALGEBRAIC DATA TYPES?

- Idris types are represented in Def with TCon and DCon
- DCon is compiled to SDataCon in STG
- TCon is compiled to STyCon in STG
- TCon and DCon is compiled to ACon in ANF

```
-- STG-Def
record STyCon ...
record DataCon ...
```

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

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

```
-- STG-Def
record STyCon ...
record DataCon ...
```

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

Idris type system is about dependent types.

```
-- Idris
data Dec : Type -> Type where
     Yes : a -> Dec a
     No : (a -> Void) -> Dec a
```

```
-- ANF
ACon "Dec" Nothing  [ALocal 0]
ACon "Yes" (Just 0) [ALocal 0, ALocal 1]
ACon "No"  (Just 0) [ALocal 0, ALocal 1]
```

STG type system is about data representation.

```
-- STG-Def
stgDec : STyCon
stgDec = MkSTyCon "Dec"
  [ MkDataCon "Yes" [UnliftedRep, UnliftedRep]
  , MkDataCon "No" [UnliftedRep, UnliftedRep]
  ]
```

How to implement special values?

- Idris type constructors
- Erased value

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

Haskell

```haskell
-- Haskell
data Erased = Erased

data IType
  = IInt
  | IList  IType
  | IMaybe IType
  ...
```

STG

```haskell
-- STG-Def
erased : STyCon
erased = MkSTyCon "Erased"
  [ MkDataCon "Erased" []
  ]

idrisType : STyCon
idrisType = MkSTyCon "IType"
  [ MkDataCon "IInt"   []
  , MkDataCon "IList"  [UnliftedRep]
  , MkDataCon "IMaybe" [UnliftedRep]
  ...
  ]
```

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)) (AlgAlt (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    ~> StgCase  (anf a1) v (DefaultAlt (anf a2)) -- Keeps Idris' strictness
ACon        m tag vs   ~> StgCon   m vs
AOp         primop as  ~> StgOpApp (anfprim primop) as
AExtPrim    n vs       ~> StgApp   n vs
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    = StgRec       SBinder Rhs
data TopBinding = StgTopLifted Binding
```

```
-- Idris compiler          -- STG-Def
MkAFun arg body      ~> StgTopLifted (StgRec name (StgRhsClosure ReEntrant (anf body)))
MkACon tag arity     ~> -- TCon and DCon are used from Defs
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

User refers the fully qualified Haskell function.

```
-- Idris example
%foreign "stg:base:Prelude.putStr"
prim__putStr : String -> PrimIO ()
```

is compiled to

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

Strictness in STG

- Laziness in data representation: LiftedRep / UnliftedRep
- Laziness in control flow StgLet / StgCase

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

STEP 5: ENJOY YOUR IDRIS PROGRAM!

FUTURE WORK

- Separate STG module generation, compilation
- Better ADT mapping in FFI
- Expose GHC RTS features, like STM

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

THANK YOU

QUESTIONS?