The Glasgow Haskell Compiler

Pedro Vasconcelos

March 21, 2023

Plan

1. Core
2. Desugaring into Core
3. Unboxed types
4. Typeclasses
5. Core-to-Core transformations
6. Exercises

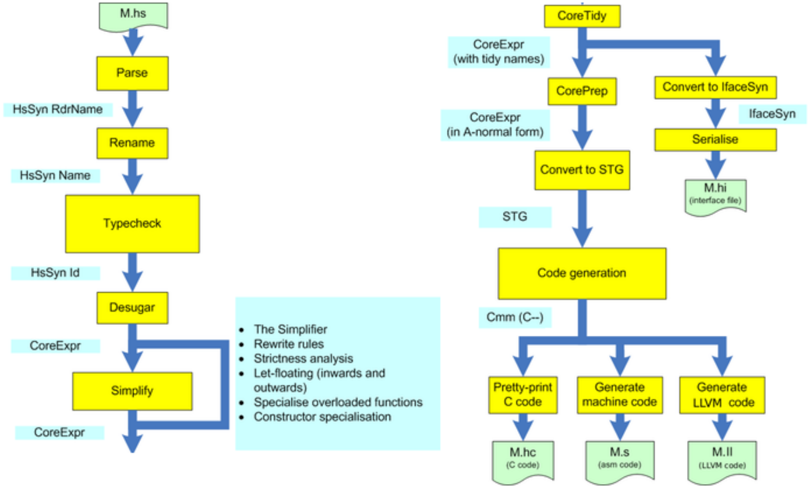
Bibliography

- The Glasgow Haskell Compiler, in *The Architecture of Open Source Applications* (vol II), Simon Marlow and Simon Peyton-Jones.
  `http://aosabook.org/en/ghc.html`
- *A transformation-based optimiser for Haskell*, Simon Peyton Jones and André Santos, 1997
- *Into the Core* @ ZuricHac 2022:
  `https://youtu.be/Gml1m-3L47s`

GHC History

- Haskell is a very large language, with many syntax constructs and extensions
- Developed over 30 years and still evolving
- The compiler is structured in stages
    - Translates source Haskell into an intermediate Core language
    - Optimizes Core using program transformations
    - Translates Core into lower-level languages (STG, C--, Asm)
- GHC source code line count has increased $5\times$ from 1992–2010
- However: the Core language has changed remarkably little

GHC Pipeline



Plan

1. Core
2. Desugaring into Core
3. Unboxed types
4. Typeclasses
5. Core-to-Core transformations
6. Exercises

What is Core?

- A small purely functional language
- Explicitly typed
- Used as intermediate form for Haskell programs
- Allows many high-level optimizations as Core-to-Core transformations

A typed intermediate language

Haskell	Core
Big	Small
Implicitly typed	Explicitly typed
Binders are typically un-annotated	Every binder is type-annotated
`\x -> x && y`	`\(x::Bool) -> x && y`
Type inference (complex, slow)	Type checking (simple, fast)
Ad-hoc restrictions to make inference feasible	Very expressive; simple, uniform

Why a typed intermediate language?

- Haskell type inference/checking works at the source level
- But then the compiler translates the code into typed Core
- Complex features are translated into a simpler language
- Many optimizations done as Core-to-Core transformations
- Sanity checking the compiler: type checking after each optimization phase

Simply typed lambda calculus

```
compose :: (Int -> Bool) -> (Char -> Int)
        -> Char -> Bool
compose = λ(f::Int -> Bool) (g::Char -> Int)
           (x::Char) ->
    let t :: Int = g x
    in f t
```

- Put a type annotation into every binder (lambda, let)
- But: what about polymorphism?

The problems with polymorphism

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose = λ(f::b -> c) (g::a -> b) (x::a)
  = let t::b = g x
      in f t
```

How to deal with applications?

```
isPos :: Int -> Bool
neg :: Int -> Int

  compose isPos neg
= { replace f = ifPos, g = neg }
  \x::a -> let t::b = neg x
              in isPos t
```

Now the type annotations are wrong!

The polymorphic lambda calculus

Also known as System F (Girard and Reynolds)

```
compose :: ∀a b c. (b -> c) -> (a -> b) -> a -> c
compose = Λa b c. λ(f::b -> c) (g::a -> b) (x::a)
   = let t::b = g x
     in f t

compose @Int @Int @Bool isPos neg
= { replace a=Int, b=Int, c=Bool, f=isPos, g=neg }
 λ(x::Int) ->
   let t::Int = neg x
   in f x
```

- Big lambdas are applied to types, just as small lambdas are applied to values
- Now the types are correct

Syntax of Core

Expressions

$$
\begin{array}{rcl}
e & ::= & k \mid C \mid x \\
& \mid & e_1\ e_2 \mid \lambda(x:\tau).\, e \\
& \mid & e\ \tau \mid \Lambda(a:\kappa).\, e \\
& \mid & \text{let } \mathit{bind} \text{ in } e \\
& \mid & \text{case } e \text{ of } \{\mathit{alt}_1 \ldots \mathit{alt}_n\} \\
\mathit{alt} & ::= & \text{DEFAULT} \mid k \\
& \mid & C\ x_1 \ldots x_n \rightarrow e \\
\mathit{bind} & ::= & x:\tau = e \\
& \mid & \text{rec } \{x_1:\tau_1 = e_1; \ldots; x_n:\tau_n = e_n\}
\end{array}
$$

Types

$$
\begin{array}{rcl}
\tau, \sigma & ::= & k \mid \alpha \\
& \mid & T\ \tau_1 \ldots t_n \\
& \mid & \tau_1\ \tau_2 \\
& \mid & \forall(a:\tau).\, \sigma \\
& \mid & \tau_1 \rightarrow \tau_2
\end{array}
$$

Plan

Desugaring into Core

- Haskell source code must be translated into Core ("desugaring")
- Each function is translated to a single equation
- Multiple equations are translated into case expressions
- Case expressions in Core must be simple: nested patterns are translated into nested case expressions
- To inspect the result of desugaring module Foo.hs:

  ```
  ghc -c -ddump-simpl Foo.hs
  ```

- Or use the Haskell playground: `https://play.haskell.org/`

Desugaring into Core (cont.)

Example:

```
-- Haskell
last :: [a] -> a
last [x] = x
last (_:xs') = last xs'

-- Core
last = Λa. λ(xs::[a]) ->
            case xs of
            [] -> error "pattern match failed"
            (x:xs') -> case xs' of
                        [] -> x
                        (y:ys) -> last @a xs'
```

Desugaring into Core (cont.)

A more complicated example:

```
-- Haskell
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

-- Core
zip = Λa b. λ(xs::[a]) (ys::[b]) ->
    case xs of
        [] -> [] @(a,b)
        x:xs' -> case ys of
            [] -> [] @(a,b)
            y:ys' -> (:) @(a,b) (x,y)
                         (zip @a @b xs' ys')
```

Plan

Unboxed types

- Core defines *unboxed* types for primitive machine types: `Int#`, `Double#`, etc.
- Haskell's types Int and Double are *boxed* using data constructors:

```
data Int = I# Int#
data Float = F# Float#
data Double = D# Double#
```

- Primitive operations work on unboxed types only: `+#`, `-#`, etc.
- Intuition: boxed values live on the heap, unboxed values live on the stack/registers

Unboxed types (cont.)

Example:

```
-- Haskell
add :: Int -> Int -> Int
add x y = x + y

-- Core
add = \(bx::Int) (by::Int) ->
    case bx of
      I# x -> case by of
                I# y -> I# (x +# y)
```

- Boxed values are taken apart by case expressions
- Primitive operations such as `+#` only over unboxed values
- Unboxed values cannot be used in polymorphic contexts (e.g. no `[Int#]`)

Why distinguish boxed vs. unboxed?

```
foo :: Int -> Int
foo n = let r = expensive n
        in if n>0
           then r `div` n
           else 0
```

- Because of lazy evaluation, we must delay the computation of `expensive n` (thunk)
- Thus: `r` must be a *boxed* int
- But: exposing unboxed types allows automatic unboxing as a Core transformation

Automatic unboxing

```
-- Haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)

-- Core, 1st version (naive)
fact :: Int -> Int
fact = \(bn::Int) ->
 case bn of
   I# n -> case n of
        0# -> I# 1#
        DEFAULT -> case (n #- 1#) of
                     n1 -> case fact (I# n1) of
                        I# r1 -> I# (r1 *# n)
```

Automatic unboxing (cont.)

```
-- Core, 2nd version (optimized)
fact :: Int -> Int
fact = \(bn::Int) ->
   case bn of
     I# n -> case wfact n of
                  r -> I# r

wfact :: Int# -> Int#
wfact = \(n::Int#) ->
   case n of
    0# -> 1#
    DEFAULT -> case (n #- 1#) of
                  n1 -> case wfact n1 of
                           r1 -> r1 *# n
```

Automatic unboxing (cont.)

- The worker function `wfact` operates on unboxed integers (no heap allocations)
- The original function `fact` is just a "wrapper" around the optimized function

When can we perform unboxing?

- We cannot always unbox to respect the non-strict semantics
- But we can unbox arguments that are strict, i.e. will definitely be evaluated
- GHC employs a safe compile-time approximation: strictness analysis
- The programmer can help:
  - by declaring data fields as strict:

    ```
    data Point = MkPoint !Int !Int
    ```

  - by using “bang patterns” in functions:

    ```
    {-# LANGUAGE BangPatterns #-}
    foo :: Int -> Int -> ...
    foo !x !y = -- strict in x and y
    ```

Plan

Typeclasses

- Typeclasses are Haskell's approach to ad-hoc polymorphism (i.e. overloading)
- Translated into dictionary passing at the level of Core

References:

1. How to make ad-hoc polymorphism less ad-hoc, Phil Wadler and Stephen Blott, 1989
2. Implementing and Understanding Type Classes, Caml mailing list, Oleg Kiseylov, 2007 `http://okmij.org/ftp/Computation/typeclass.html`

A simple class

```
class Show a where
   show :: a -> String

instance Show Bool where
  show b = if b then "True" else "False"

instance Show Int where
  show = intToString

intToString :: Int -> String
...

print :: Show a => a -> IO ()
print x = putStrLn (show x)
```

How can this be implemented?

Challenges

- A type class can define any number of methods
- A class can be implemented (instanced) at any number of types
- Instances may be defined in a separate module from the classes
- A function may be overloaded with more than one type class

Dictionary passing

- Each *class declaration* corresponds to a *data type declaration* for the record of operations (the *dictionary*)
- Each *instance declaration* corresponds to a *value* of the corresponding dictionary type
- Overloaded functions take *dictionaries as extra parameters*

Example again

```
class Show a where
   show :: a -> String

instance Show Bool where
  show b = if b then "True" else "False"

instance Show Int where
  show = intToString

print :: Show a => a -> IO ()
print x = putStrLn (show x)
```

Example again (cont.)

After translation:

```
data DShow a = DShow { -- dictionary for Show
    show :: a -> String
  }

dShowBool :: DShow Bool -- instance for Bool
dShowBool = DShow @Bool
   (\(b::Bool) -> if b then "True" else "False")

dShowInt :: DShow Int     -- instance for Int
dShowInt = DShow @Int intToString

print :: forall a. DShow a -> a -> IO ()
print = \a -> (dShow::DShow a) (x::a) ->
   putStrLn (show @a dShow x)
```

Another example: a simplified Num class

```
class Num a where
    fromInt :: Int -> a
    (+) :: a -> a -> a

foo :: Num a => a -> a
foo x = x+1
```

Another example: a simplified Num class (cont.)

After translation:

```
data DNum a = DNum {
    fromInt :: Int -> a
    (+) :: a -> a -> a
}

foo = \a (dNum::DNum a) (x::a) ->
          (+) @a dNum x (fromInt @a @dNum (I# 1#))
```

Example: more than one class

```
class Show a where
    show :: a -> String

class Num a where
    fromInt :: Int -> a
    (+) :: a -> a -> a

foo :: (Show a, Num a) => a -> String
foo x = show (x+1)
```

Example: more than one class (cont.)

After translation:

```
data DShow a = DShow {
     show :: a -> String
     }

data DNum a = DNum {
     fromInt :: Int -> a
     (+) :: a -> a -> a
}

foo :: forall a. DShow a -> DNum a -> a -> String
foo = \a (dShow::DShow a) (dNum::DNum a) (x::a)
    = show @a dShow
          ((+) @a dNum x (fromInt @a dNum (I# 1#))))
```

Observations

- Dictionaries behave similiarly to virtual method calls in an OO-language
- In particular they allow separate compilation

Unlike OO methods, dictionaries are passed separately from data objects:

- allow multiple dictionaries for a single value
- the compiler ensures that values and dictionaries "meet up" at the right place
- allows dispatching on the *result type* (e.g. `fromInt`)
- allows type safety for *binary methods* (e.g. ==)
- allows *multi-parameter classes*

Specialization

- In cases where GHC knows the concrete type it can avoid passing the dictionary
- This transformation is called specialization
- Similar to *monomorphization* in C++ and Rust

Specialization (cont.)

```
-- Haskell code
bar :: Int -> Int
bar x = x + 1

-- naive translation
bar = \(x :: Int) ->
     (+) @Int GHC.Num.dNumInt x (fromInt (I# 1#))

-- specialized translation (optimization)
bar = \(x :: Int) ->
      case x of { I# y ->
         I# (+# y 1#)
      }
```

Plan

The simplifier

- The simplifier performs many optimizations as Core-to-core program transformations
- Each transformation performs a small change, but collectively they have a large impact
- Higher level of optimizations run the simplifier many times because transformations can “cascade”

Inlining and Beta-reduction

Inlining: replace one or more occurence of a let-bound variable by a copy of the right-hand side.

```
let x = e1
in ... x ... x ...
```

$\Downarrow$

```
let x = e1
in ... e1 ... e1 ...
```

Beta-reduction: apply known function.

```
(\x -> e1) e2
```

$\Downarrow$

```
let x = e2 in e1
```

Inlining and Beta-reduction (cont.)

Advantages:

- Functional programs typically define many small functions; inlining removes the cost of function application
- But more importantly: inlining opens possibilities for other transformations

Disadvantages:

- Must careful not to duplicate work; for example, inlining WHNFs (such as functions) is OK
- May cause cause generated code size expansion

Cross-module inlining

GHC produces for each `.hs` file:

- a object code file `.o`; and
- an interface file `.hi`

The interface file contains:

- the types of every exported binding
- for "small" definitions: the complete definition of the functions

This allows performing inlining across modules.

Cross-module inlining (cont.)

Foo.hs

```haskell
module Foo where
foo xs = null (reverse xs)
```

```
ghc -O -c Foo.hs
ghc --show-iface Foo.hi
```

$\Downarrow$

```
...
foo :: [a] -> GHC.Types.Bool
...
Unfolding: ...
 ( @a (xs :: [a]) ->
    case GHC.List.reverse1 @a xs ([] @a) of
      [] -> True ; : _ _ -> False )
```

Case of known constructor

```
case (C e1 e2 ... ek) of
...
C x1 x2 .. xk -> rhs
...
```

$\Downarrow$

```
let x1=e1; x2=e2;...;xk=ek
in rhs
```

Case of known constructor (cont.)

Example:

```
case False of
  True  -> e1
  False -> e2
```

$\Downarrow$

```
e2
```

Does not typically occur in source Haskell, but may result from other transformations!

Case of case

Example:

```
if (not x) then e1 else e2
```

After inlining and beta-reducing:

```
case (case x of True -> False
                False -> True) of
  True -> e1
  False -> e2
```

Case of case (cont.)

Push the outer case into the branches of the inner case:

```
case x of
   True-> case False of { True-> e1; False-> e2 }
   False-> case True of { True-> e1; False-> e2 }
```

Now we can simplify the inner cases:

```
case x of
   True -> e2
   False -> e1
```

Case of case (cont.)

Another example; assume

```
(||) = \a b -> case a of {True-> True; False-> b}
```

Then inlining `||` into

```
if (x || y) then e1 else e2
```

gives:

```
case (case x of {True->True; False->y}) of
  True -> e1
  False -> e2
```

This a “case-of-case” as before…

Case of case (cont.)

```
case (case x of {True->True; False->y}) of
  True -> e1
  False -> e2
```

$\Downarrow$

```
case x of
  True -> case True of {True->e1; False->e2}
  False -> case y of {True->e1; False->e2}
```

Now only one of the branches simplifies:

```
case x of
  True -> e1
  False -> case y of {True->e1; False->e2}
```

We have duplicated `e1` — this can be problem if it is a big expression.

Join points

Solution: name the right hand sides to avoid duplication (join points)

```
let j1 = e1
    j2 = e2
in case x of
  True -> j1
  False -> case y of { True -> j1; False -> j2 }
```

- Now `j1` is define once
- Since `j2` is only used once it could also be inlined without duplicating work
- The backend can implement join points as a simple jump (not a general let)

Join points (cont.)

What about pattern variables?

```
case (case b of {True-> b1; False->b2}) of
    [] -> e1
    (x:xs) -> e2
```

Solution: make join points into functions.

```
let j1 = e1
     j2 = \x xs -> e2
in case b of
   True -> case b1 of
               [] -> j1
               (x:xs) -> j2 x xs
   False-> case b2 of
               [] -> j1
               (x:xs) -> j2 x xs
```

Generalising case elimination

Example:

```
if null xs then r else tail xs

null = \as -> case as of
                 {[]-> True; (b:bs)-> False}
tail = \cs -> case cs of
                 {[]->error "..."; (b:bs)->bs}
```

Generalising case elimination (cont.)

After inlining:

```
case (case xs of []->True; (b:bs)->False) of
  True -> r
  False -> case xs of
                []-> error "..."
                (d:ds) -> ds
```

Applying case-of-case:

```
case xs of
  [] -> r
  (b:bs) -> case xs of
              [] -> error ".."
              (d:ds) -> ds
```

Note that the inner `case xs` is redudant!

Generalising case elimination (cont.)

Eliminating the inner case and replacing `bs` for `ds`:

```
case xs of
  [] -> r
  (b:bs) -> bs
```

Rewrite rules

Like most conventional compilers, GHC tries to simplify expressions at compile time (constant folding).

`(1# +# 2#)` $\Longrightarrow$ `(3#)`

`(x +# 0#)` $\Longrightarrow$ `x`

Rewrite rules (cont.)

Unlike typical compilers: programmers can also define domain-specific rewrite rules.

```
map f [] = []
map f (x:xs) = f x : map f xs

{-# RULES
 "map/map" forall f g xs.
           map f (map g xs) = map (f . g) xs
  #-}
```

Rewrite rules (cont.)

GHC will replace the left hand side by the right hand side:

```
map (\x -> 3*x) (map (\y->y+1) list)
```

$\Downarrow$

```
map ((\x -> 3*x).(\y->y+1)) list
```

This is an optimization because it avoids the allocation of an intermediate list (list fusion).

Rewrite rules (cont.)

- Can only be used to replace function applications
- Can be used for domain-specific specialization:

```
genericLookup :: Ord a => Table a b -> a -> b
intLookup ::             Table Int b -> Int -> b

{-# RULES
  "genericLookup/Int" genericLookup = intLookup
  #-}
```

- GHC does **not** check that the rule is valid or terminating
- What's wrong with this rule?

```
{-# RULES
  "reverse/reverse" reverse (reverse xs) = xs
  #-}
```

Rewrite rules (cont.)

- Rewrite rules were initially designed to support list fusion
- But library writters found other uses
- E.g. in Data.Vector, Data.Text it enables the elimination of intermediate data structures
- Further reading: *Playing by the rules: rewriting as practical optimization technique in GHC*, Simon Peyton Jones, Andrew Tolmach, Tony Hoare, 2001

Plan

Exercises

Translate the following into Core by hand.

```
foo :: Int -> Int -> Int
foo x y = 2*x+y

append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Exercises (cont.)

Inspect the Core generated for the following with different levels of optimization.

```
foo :: Int -> Int -> Int
foo x y = 3*x+2*y

mysym :: [Int] -> Int
mysym [] = 0
mysum (x:xs) = x + mysum xs

avg :: [Double] -> Double
avg xs = loop xs 0 0
   where
   loop :: [Double] -> Double -> Int -> Double
   loop [] s n      = s / fromIntegral n
   loop (x:xs) s n = loop xs (x+s) (n+1)
```