System F and GHC Core

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK

2018-02-26

Goals

- System F
- The GHC Core language

Another look at polymorphism

One function, many types

```
reverse :: [a] -> [a]
```

One function, many types

```haskell
reverse :: [a] -> [a]
```

```haskell
reverse :: [Int] -> [Int]
reverse :: [Char] -> [Char]
reverse :: [Bool] -> [Bool]
reverse :: [IO ()] -> [IO ()]
reverse :: [Maybe Double] -> [Maybe Double]
reverse :: [[Either Int ()]] -> [[Either Int ()]]
reverse :: [Int -> [Bool]] -> [Int -> [Bool]]
```

Polymorphic function as a dialogue

Hi, I'm the `reverse` program.

Could you please let me know lists of what type you want to process?

Polymorphic function as a dialogue

Hi, I'm the `reverse` program.

Could you please let me know lists of what type you want to process?

Oh, yes, `Bool` please.

Polymorphic function as a dialogue

Hi, I'm the `reverse` program.

Could you please let me know lists of what type you want to process?

Oh, yes, `Bool` please.

Of course, I can do that. Then please give me a list of `Bool`s.

Polymorphic function as a dialogue

Hi, I'm the `reverse` program.

Could you please let me know lists of what type you want to process?

Oh, yes, `Bool` please.

Of course, I can do that. Then please give me a list of `Bool`s.

That would be `[True, False, False]`.

Polymorphic function as a dialogue

Hi, I'm the `reverse` program.

Could you please let me know lists of what type you want to process?

Oh, yes, `Bool` please.

Of course, I can do that. Then please give me a list of `Bool`s.

That would be `[True, False, False]`.

Ok, then the result is `[False, False, True]`.

Polymorphic function as a dialogue

Hi, I'm the `reverse` program.

Could you please let me know lists of what type you want to process?

Oh, yes, `Bool` please.

Of course, I can do that. Then please give me a list of `Bool`s.

That would be `[True, False, False]`.

Ok, then the result is `[False, False, True]`.

The type is an input in this dialogue.

Types as arguments

```
reverse ::                  [a] -> [a]
```

This is the type of `reverse` as we typically write it in Haskell.

Types as arguments

```haskell
reverse :: forall a . [a] -> [a]
```

In fact, this is an abbreviation for an explicitly quantified type.

Types as arguments

```
reverse :: forall a . [a] -> [a]
```

As just seen, we can view quantification over a type as asking for a type as input.

Types as arguments

```
reverse :: forall a . [a] -> [a]
reverse = /\ a -> \ (xs :: [a]) -> ...
```

We can make the type input explicit by introducing a type lambda.

Types as arguments

```
reverse :: forall a . [a] -> [a]
reverse = /\ a -> \ (xs :: [a]) -> ...
```

We can make the type input explicit by introducing a type lambda.

```
reverse Bool [True, False, False]
```

If we call a polymorphic function, we consequently also have to supply the argument as an extra argument.

Types as arguments

```
reverse :: forall a . [a] -> [a]
reverse = /\ a -> \ (xs :: [a]) ->
  foldl
    (flip (:))
    []
    xs
```

Types as arguments

```
reverse :: forall a . [a] -> [a]
reverse = /\ a -> \ (xs :: [a]) ->
  foldl
    (flip (:))
    []
    xs
```

```
foldl :: forall a b . (b -> a -> b) -> b -> [a] -> b
flip  :: forall a b c . (a -> b -> c) -> b -> a -> c
(:)   :: forall a . a -> [a] -> [a]
[]    :: forall a . [a]
```

Similarly, if we want to provide the definition of `reverse`, we will almost inevitably make use of other polymorphic entities …

Types as arguments

```
reverse :: forall a . [a] -> [a]
reverse = /\ a -> \ (xs :: [a]) ->
  foldl a [a]
    (flip a [a] [a] ((:) a))
    ([] a)
    xs
```

```
foldl :: forall a b . (b -> a -> b) -> b -> [a] -> b
flip  :: forall a b c . (a -> b -> c) -> b -> a -> c
(:)   :: forall a . a -> [a] -> [a]
[]    :: forall a . [a]
```

… so we should supply type arguments to these as well.

Types as arguments

```
reverse = /\ a -> \ (xs :: [a]) ->
  foldl a [a]
    (flip a [a] [a] ((:) a))
    ([] a)
    xs
```

Even if we remove the type signature, the term now contains sufficient information to easily check the type correctness of it.

Types as arguments

```
reverse = Λ a -> \ (xs :: [a]) ->
  foldl a [a]
    (flip a [a] [a] ((:) a))
    ([] a)
    xs
```

Even if we remove the type signature, the term now contains sufficient information to easily check the type correctness of it.

In particular, we know exactly where to generalise (at type abstractions) and to instantiate (at type applications).

Types as arguments

```
reverse = /\ a -> \ (xs :: [a]) ->
  foldl a [a]
    (flip a [a] [a] ((:) a))
    ([] a)
    xs
```

Having the ability to apply type abstraction freely gives us a powerful type system with higher-rank polymorphism.

On the other hand, it contains too much noise to be pleasant to use directly.

Types as arguments in Haskell

There are situations in Haskell that make type annotations necessary.

For example, let's say we want to parse a `String` as an `Int` and then `show` the result:

```haskell
strange :: String -> String
strange = show . read
```

```
Ambiguous type variable  a0  arising from a use of  show
prevents the constraint  (Show a0)  from being solved .
Probable fix : use a type annotation to specify what  a0  should be .
 ...
Ambiguous type variable  a0  arising from a use of  read
prevents the constraint  (Read a0)  from being solved .
Probable fix : use a type annotation to specify what  a0  should be .
 ...
```

Types as arguments in Haskell

There are situations in Haskell that make type annotations necessary.

For example, let's say we want to parse a `String` as an `Int` and then `show` the result:

We can fix this by following GHC's suggestion:

```
strange :: String -> String
strange = show . (read :: String -> Int)
```

```
GHCi> strange "00042"
"42"
```

Types as arguments in Haskell

There are situations in Haskell that make type annotations necessary.

For example, let's say we want to parse a `String` as an `Int` and then `show` the result:

However, using

```
{-# LANGUAGE TypeApplications #-}
```

we can also write:

```
strange :: String -> String
strange = show . read @ Int
```

```
GHCi> strange "00042"
"42"
```

Types as arguments in Haskell (contd.)

- So it is actually possible to explicitly apply a type in Haskell, using the `@` operator.
- This feature has been a relatively recent addition to GHC.
- Explicit type abstraction, however (i.e., the use of `@` in patterns) is not (yet) possible.

System F

System F

- A lambda calculus with explicit type abstraction and type application and thus higher-rank polymorphic types.
- All abstractions are explicitly annotated, allowing for straight-forward type checking.

System F

- A lambda calculus with explicit type abstraction and type application and thus higher-rank polymorphic types.
- All abstractions are explicitly annotated, allowing for straight-forward type checking.
- Type inference (recovering erased annotations) is undecidable.

System F

- A lambda calculus with explicit type abstraction and type application and thus higher-rank polymorphic types.
- All abstractions are explicitly annotated, allowing for straight-forward type checking.
- Type inference (recovering erased annotations) is undecidable.
- Not useful as a programming language for end users, but useful for theory and also as the basis of internal / intermediate languages.
- GHC has an intermediate language called Core which is based on an extension of System F.

The plan

- Introduce the syntax and type rules of System F in the theoretical setting.
- Then have a look at Core and see how it compares.

System F syntax

Terms:

$$
\begin{array}{lll}
e ::= & x & \text{(variable)} \\
\mid & (e_1 \ e_2) & \text{(term application)} \\
\mid & \lambda(x :: \tau) \rightarrow e & \text{(term abstraction)} \\
\mid & (e \ \tau) & \text{(type application)} \\
\mid & \Lambda\alpha \rightarrow e & \text{(type abstraction)}
\end{array}
$$

System F syntax

Terms:

$$
\begin{array}{rll}
e ::= & x & \text{(variable)} \\
\mid & (e_1 \; e_2) & \text{(term application)} \\
\mid & \lambda(x :: \tau) \to e & \text{(term abstraction)} \\
\mid & (e \; \tau) & \text{(type application)} \\
\mid & \Lambda\alpha \to e & \text{(type abstraction)}
\end{array}
$$

Types:

$$
\begin{array}{rll}
\tau ::= & \alpha & \text{(type variable)} \\
\mid & T_n \; \tau_1 \; \ldots \; \tau_n & \text{(constructor application)} \\
\mid & \tau_1 \to \tau_2 & \text{(function type)} \\
\mid & \forall\alpha.\ \tau & \text{(quantified type)}
\end{array}
$$

Environment

$$
\begin{array}{rcll}
\Gamma & ::= & \epsilon & \text{(empty environment)} \\
& | & \Gamma, x :: \tau & \text{(introducing a variable)} \\
& | & \Gamma, \alpha & \text{(introducing a type variable)}
\end{array}
$$

Type judgements

$\Gamma \vdash \tau \text{ wf}$

Expresses that a type τ is well-formed in environment Γ.

$\Gamma \vdash e :: \tau$

Expresses that expression e has type τ in environment Γ.

Well-formedness of types

In classic System F, the only constraint on types is that they must not contain unknown type variables:

Well-formedness of types

In classic System F, the only constraint on types is that they must not contain unknown type variables:

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \ \mathsf{wf}}$$

Well-formedness of types

In classic System F, the only constraint on types is that they must not contain unknown type variables:

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \ \mathsf{wf}}$$

$$\frac{\Gamma \vdash \tau_1 \ \mathsf{wf} \qquad \dots \qquad \Gamma \vdash \tau_n \ \mathsf{wf}}{\Gamma \vdash T_n \ \tau_1 \ \dots \ \tau_n \ \mathsf{wf}}$$

Well-formedness of types

In classic System F, the only constraint on types is that they must not contain unknown type variables:

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \ \mathsf{wf}}$$

$$\frac{\Gamma \vdash \tau_1 \ \mathsf{wf} \qquad \dots \qquad \Gamma \vdash \tau_n \ \mathsf{wf}}{\Gamma \vdash T_n \ \tau_1 \ \dots \ \tau_n \ \mathsf{wf}}$$

$$\frac{\Gamma \vdash \tau_1 \ \mathsf{wf} \qquad \Gamma \vdash \tau_2 \ \mathsf{wf}}{\Gamma \vdash \tau_1 \to \tau_2 \ \mathsf{wf}}$$

Well-formedness of types

In classic System F, the only constraint on types is that they must not contain unknown type variables:

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \ \mathsf{wf}}$$

$$\frac{\Gamma \vdash \tau_1 \ \mathsf{wf} \qquad \dots \qquad \Gamma \vdash \tau_n \ \mathsf{wf}}{\Gamma \vdash T_n \ \tau_1 \dots \tau_n \ \mathsf{wf}}$$

$$\frac{\Gamma \vdash \tau_1 \ \mathsf{wf} \qquad \Gamma \vdash \tau_2 \ \mathsf{wf}}{\Gamma \vdash \tau_1 \to \tau_2 \ \mathsf{wf}}$$

$$\frac{\Gamma, \alpha \vdash \tau \ \mathsf{wf}}{\Gamma \vdash \forall \alpha.\ \tau \ \mathsf{wf}}$$

Well-typed terms

$$\frac{x :: \tau \in \Gamma}{\Gamma \vdash x :: \tau}$$

The rule for variables is unsurprising.

Well-typed terms (contd.)

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2}$$

For application, the first term must have a function type. *It cannot be quantified.*

The second term must have a compatible argument type.

Well-typed terms (contd.)

$$\frac{\Gamma \vdash \tau_1 \ \mathsf{wf} \qquad \Gamma, x :: \tau_1 \vdash e :: \tau_2}{\Gamma \vdash \lambda(x :: \tau_1) \to e :: \tau_1 \ \texttt{->} \ \tau_2}$$

We have to check the type-annotation for well-formedness, because it is part of the program.

Algorithmically, no guessing is involved, because we can extend the environment with the type from the annotation.

Well-typed terms (contd.)

$$\frac{\Gamma \vdash e :: \forall \alpha.\ \tau_1 \qquad \Gamma \vdash \tau_2 \ \mathsf{wf}}{\Gamma \vdash (e\ \tau_2) :: [\alpha \mapsto \tau_2]\ \tau_1}$$

For a type application to be well-typed the term must have a quantified type.

The result type is obtained by replacing the type variable with the argument type in the body.

Well-typed terms (contd.)

$$\frac{\Gamma, \alpha \vdash e :: \tau}{\Gamma \vdash \Lambda\alpha \to e :: \forall\alpha.\ \tau}$$

Type abstraction introduces polymorphism.

Example: Polymorphic identity

$$\dfrac{\dfrac{\dfrac{\alpha \in \alpha}{\alpha \vdash \alpha \ \mathsf{wf}} \qquad \dfrac{x :: \alpha \in \alpha, x :: \alpha}{\alpha, x :: \alpha \vdash x :: \alpha}}{\alpha \vdash \lambda(x :: \alpha) \to x :: \alpha \to \alpha}}{\epsilon \vdash \Lambda\alpha \to \lambda(x :: \alpha) \to x :: \forall\alpha.\ \alpha \to \alpha}$$

Example: Identity of identity

Assume

$$\Gamma = \epsilon, i :: \forall \alpha.\ \alpha \to \alpha,$$

Example: Identity of identity (contd.)

$$\dfrac{\dfrac{\dfrac{i :: \forall\alpha.\ \alpha \to \alpha \in \Gamma,\ \beta}{\Gamma,\ \beta \vdash i :: \forall\alpha.\ \alpha \to \alpha} \qquad \dfrac{\dfrac{\beta \in \Gamma,\ \beta}{\Gamma,\ \beta \vdash \beta\ \mathsf{wf}} \qquad \dfrac{\beta \in \Gamma,\ \beta}{\Gamma,\ \beta \vdash \beta\ \mathsf{wf}}}{\Gamma,\ \beta \vdash (\beta \to \beta)\ \mathsf{wf}}}{\Gamma,\ \beta \vdash i\ (\beta \to \beta) :: (\beta \to \beta) \to \beta \to \beta} \qquad \dfrac{\ldots}{\Gamma,\ \beta \vdash i\ \beta :: \beta \to \beta}}{\dfrac{\Gamma,\ \beta \vdash i\ (\beta \to \beta)\ (i\ \beta) :: \beta \to \beta}{\Gamma \vdash \Lambda\beta \to i\ (\beta \to \beta)\ (i\ \beta) :: \forall\beta.\ \beta \to \beta}}$$

GHC Core

GHC pipeline

GHC pipeline

- Parse to an abstract syntax tree.

GHC pipeline

- Parse to an abstract syntax tree.
- Rename (resolve all names).

GHC pipeline

- Parse to an abstract syntax tree.
- Rename (resolve all names).
- Typecheck and desugar to Core.

GHC pipeline

- Parse to an abstract syntax tree.
- Rename (resolve all names).
- Typecheck and desugar to Core.
- Optimise (on Core).

GHC pipeline

- Parse to an abstract syntax tree.
- Rename (resolve all names).
- Typecheck and desugar to Core.
- Optimise (on Core).
- Compile to STG.

GHC pipeline

- Parse to an abstract syntax tree.
- Rename (resolve all names).
- Typecheck and desugar to Core.
- Optimise (on Core).
- Compile to STG.
- Generate code for one of several back-ends (e.g. native code generators, LLVM).

Core and STG

Core:

- A very simple language based on System F.
- Explicitly typed.
- Still very close to Haskell.

Core and STG

Core:

- A very simple language based on System F.
- Explicitly typed.
- Still very close to Haskell.

STG:

- Spineless Tagless G-Machine.
- No longer typed.
- Still functional, but severely restricted (functions can only be applied to variables).
- All allocation is explicit (let-bindings correspond to allocation).

Why Core?

Simple:

- Not difficult to generate from Haskell.
- Good for optimisations, because few constructs have to be considered.
- Easy to reason about.

Why Core?

Simple:

- Not difficult to generate from Haskell.
- Good for optimisations, because few constructs have to be considered.
- Easy to reason about.

Typed:

- Useful sanity check for all optimisations.
- Type checking is very simple and straight-forward (not even close to the complexity of Haskell's type inference engine).
- Explicit typing overhead does not matter for machine-generated code.

A look at the Core AST

```
data Expr b =
    Var      Id                            -- variable
  | Lit      Literal                       -- literal
  | App      (Expr b) (Expr b)             -- application
  | Lam      b (Expr b)                    -- abstraction
  | Let      (Bind b) (Expr b)             -- let
  | Case     (Expr b) b Type [Alt b]       -- case
  | Cast     (Expr b) Coercion             -- type cast
  | Tick     (Tickish Id) (Expr b)         -- annotations
  | Type     Type                          -- type
  | Coercion Coercion                      -- coercion
```

Parameterised over the binding construct (usually just a variable).

A look at the Core AST

```haskell
data Expr b =
    Var      Id                              -- variable
  | Lit      Literal                         -- literal
  | App      (Expr b) (Expr b)               -- application
  | Lam      b (Expr b)                      -- abstraction
  | Let      (Bind b) (Expr b)               -- let
  | Case     (Expr b) b Type [Alt b]         -- case
  | Cast     (Expr b) Coercion               -- type cast
  | Tick     (Tickish Id) (Expr b)           -- annotations
  | Type     Type                            -- type
  | Coercion Coercion                        -- coercion
```

Parameterised over the binding construct (usually just a variable).

Type application and type abstraction via normal `App` and `Lam` using `Type`.

Additions w.r.t. System F

- User-defined datatypes
- Structure on the type level (kinds)
- Case statements to eliminate user-defined datatypes
- Let bindings
- Casts and coercions (used for type system extensions)

What’s not in Core

- Any form of syntactic sugar (e.g. for ranged lists, list comprehensions, do notation, …).
- Type classes
- Full pattern matching
- …

Looking at Core

We can look at Core produced by GHC by passing `-ddump-simpl` to the compiler.

Looking at Core

We can look at Core produced by GHC by passing `-ddump-simpl` to the compiler.

Depending on the optimisation level, Core is going to be either relatively close to the original program (`-O0`) or possibly quite far away (`-O` or `-O2`).

Identity in Core

```
id x = x
```

Identity in Core

```
id x = x
```

```
Core1.id :: forall t_ap4. t_ap4 -> t_ap4
[GblId, Arity=1, Caf=NoCafRefs, Str=DmdType]
Core1.id = \ (@ t_ap4) (x_aoX :: t_ap4) -> x_aoX
```

Type abstraction signalled via use of `@`.

All variable names are made unique.

All other names are fully qualified.

Tweaking Core output

With `-dsuppress-all`:

```
id
id = \ @ t_ap4 x_aoX -> x_aoX
```

Often, a bit too much (no types anymore).

Tweaking Core output

With `-dsuppress-all`:

```
id
id = \ @ t_ap4 x_aoX -> x_aoX
```

Often, a bit too much (no types anymore).

With
`-dsuppress-idinfo -dsuppress-module-prefixes`:

```
id :: forall t_ap4. t_ap4 -> t_ap4
id = \ (@ t_ap4) (x_aoX :: t_ap4) -> x_aoX
```

We'll use this for now.

Identity of identity in Core

```
idOfId = id id
```

Identity of identity in Core

```
idOfId = id id
```

With `-O0`:

```
idOfId :: forall a_ap4. a_ap4 -> a_ap4
idOfId = \ (@ a_ap4) ->
  id @ (a_ap4 -> a_ap4) (id @ a_ap4)
```

Identity of identity in Core

```
idOfId = id id
```

With `-O0`:

```
idOfId :: forall a_ap4. a_ap4 -> a_ap4
idOfId = \ (@ a_ap4) ->
  id @ (a_ap4 -> a_ap4) (id @ a_ap4)
```

With `-O`:

```
idOfId :: forall a_aqn. a_aqn -> a_aqn
idOfId = id
```

Building a list in Core

```
list = [False]
```

Building a list in Core

```haskell
list = [False]
```

```
list :: [Bool]
list = : @ Bool False ([] @ Bool)
```

Everything is prefix in Core.

List constructors are polymorphic, so type application is needed.

Pattern matching in Core

```haskell
(||) :: Bool -> Bool -> Bool
False || x = x
True  || x = True
```

```
|| :: Bool -> Bool -> Bool
|| =
  \ (ds_dwt :: Bool) (x_aw4 :: Bool) ->
    case ds_dwt of _ {
      False -> x_aw4;
      True -> True
    }
```

Nested pattern matching

```
and :: [Bool] -> Bool
and [] = True
and (True  : xs) = and xs
and (False : xs) = False
```

Nested pattern matching

```
and :: [Bool] -> Bool
and [] = True
and (True  : xs) = and xs
and (False : xs) = False
```

```
and :: [Bool] -> Bool
and =
  \ (ds_dwt :: [Bool]) ->
    case ds_dwt of _ {
      [] -> True;
      : ds1_dwB xs_aw6 ->
        case ds1_dwB of _ {
          False -> False;
          True -> and xs_aw6
        }
    }
```

Case statement in Core

- Pattern matching in Core is always flat (only top-level constructors).
- Pattern matching in Core always evaluates the analyzed term to WHNF. In Haskell, pattern match can trigger arbitrary amounts of evaluation depending on the patterns, including none.

Overloading

```haskell
two :: Int
two = 1 + 1
```

Overloading

```haskell
two :: Int
two = 1 + 1
```

With `-O0`:

```
two :: Int
two = + @ Int $fNumInt (I# 1#) (I# 1#)
```

With `-O`:

```
two :: Int
two = I# 2#
```

Dictionary translation of type classes

Overview of dictionary translation

type class	record type
instance	term of record type
instance with constraints	function that transforms record
class method	record selector function
method invocation	application to record argument
overloaded function	function with record argument

Example

```haskell
two :: Int
two = 1 + 1
```

```haskell
(+) :: Num a => a -> a -> a
```

Example

```haskell
two :: Int
two = 1 + 1
```

```haskell
(+) :: Num a => a -> a -> a
```

```
two :: Int
two = + @ Int $fNumInt (I# 1#) (I# 1#)
```

Example

```haskell
two :: Int
two = 1 + 1
```

```haskell
(+) :: Num a => a -> a -> a
```

```
two :: Int
two = + @ Int $fNumInt (I# 1#) (I# 1#)
```

- First argument to `+`: the instantiation of `a`.
- Second argument to `+`: a dictionary called `$fNumInt` of type `Num a`.
- Class constraints become explicit arguments.
- We can see the constructor of type `Int` being applied to unboxed integers in the other arguments.

Another example

```
inc :: [Int] -> [Int]
inc = map (+ 1)
```

Another example

```haskell
inc :: [Int] -> [Int]
inc = map (+ 1)
```

With `-O0`:

```
inc :: [Int] -> [Int]
inc =
  map
    @ Int
    @ Int
    (let {
       ds_dHn :: Int
       ds_dHn = I# 1# } in
     \ (ds1_dHm :: Int) ->
       + @ Int $fNumInt ds1_dHm ds_dHn)
```

The operator section is desugared.

Another example

```haskell
inc :: [Int] -> [Int]
inc = map (+ 1)
```

With `-O`:

```
inc1 :: Int -> Int
inc1 =
  \ (ds_dID :: Int) ->
    case ds_dID of _
      { I# x_aIV -> I# (+# x_aIV 1#) }

inc :: [Int] -> [Int]
inc = map @ Int @ Int inc1
```

Dictionary is optimised away.

Addition happens on unboxed integers.

Yet another example

```
double x = 2 * x
```

Yet another example

```
double x = 2 * x
```

```
double :: forall a_aqr. Num a_aqr => a_aqr -> a_aqr
double =
  \ (@ a_aqr) ($dNum_aH5 :: Num a_aqr)
    (x_aoX :: a_aqr) ->
    * @ a_aqr $dNum_aH5
      (fromInteger @ a_aqr $dNum_aH5 2) x_aoX
```

Yet another example

```haskell
double x = 2 * x
```

```
double :: forall a_aqr. Num a_aqr => a_aqr -> a_aqr
double =
  \ (@ a_aqr) ($dNum_aH5 :: Num a_aqr)
    (x_aoX :: a_aqr) ->
    * @ a_aqr $dNum_aH5
      (fromInteger @ a_aqr $dNum_aH5 2) x_aoX
```

- We receive and pass on a `Num` dictionary.
- Literals are not overloaded in Core. This is desugared via `fromInteger`.

Range example

```
ten x = [x .. x + 9]
```

Range example

```
ten x = [x . . x + 9]
```

```
ten :: forall a_aqw.
  (Enum a_aqw, Num a_aqw) => a_aqw -> [a_aqw]
ten =
  \ (@ a_aqw)
    ($dEnum_aOY :: Enum a_aqw)
    ($dNum_aOZ :: Num a_aqw)
    (x_ap0 :: a_aqw) ->
    enumFromTo
      @ a_aqw
      $dEnum_aOY
      x_ap0
      (+ @ a_aqw $dNum_aOZ x_ap0
        (fromInteger @ a_aqw $dNum_aOZ 9))
```

Range example

```
ten x = [x .. x + 9]
```

```
ten :: forall a_aqw.
  (Enum a_aqw, Num a_aqw) => a_aqw -> [a_aqw]
ten =
  \ (@ a_aqw)
    ($dEnum_aOY :: Enum a_aqw)
    ($dNum_aOZ :: Num a_aqw)
    (x_ap0 :: a_aqw) ->
    enumFromTo
      @ a_aqw
      $dEnum_aOY
      x_ap0
      (+ @ a_aqw $dNum_aOZ x_ap0
        (fromInteger @ a_aqw $dNum_aOZ 9))
```

Multiple dictionaries. Range is desugared via `enumFromTo`.

Class example

```haskell
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

Class example

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

```
mempty :: forall m_aoZ[sk]. Monoid m_aoZ[sk] => m_aoZ[sk]
mempty =
  \ (@ m_aoZ[sk]) (tpl_B1 :: Monoid m_aoZ[sk]) ->
    case tpl_B1 of tpl_B1
      { C:Monoid tpl_B2 tpl_B3 -> tpl_B2 }
```

Class example

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

```
mempty :: forall m_aoZ[sk]. Monoid m_aoZ[sk] => m_aoZ[sk]
mempty =
  \ (@ m_aoZ[sk]) (tpl_B1 :: Monoid m_aoZ[sk]) ->
    case tpl_B1 of tpl_B1
      { C:Monoid tpl_B2 tpl_B3 -> tpl_B2 }
```

- Dictionary type definition not explicitly shown.
- We match on constructor `C:Monoid` of type `Monoid`.
- Two arguments of `C:Monoid`, two class methods.
- We extract the first (corresponding to `mempty`).
- An extractor for `mappend` is also generated.

Instance example

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Instance example

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

```
$fMonoid[] :: forall a_awF. Monoid [a_awF]
$fMonoid[] =
  \ (@ a_awH) ->
    C:Monoid @ [a_awH] ([] @ a_awH) (++ @ a_awH)
```

Instance example

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

```
$fMonoid[] :: forall a_awF. Monoid [a_awF]
$fMonoid[] =
  \ (@ a_awH) ->
    C:Monoid @ [a_awH] ([] @ a_awH) (++ @ a_awH)
```

- The instance translates into the dictionary definition.
- We apply the `C:Monoid` constructor to the implementations of the two class methods.

Instance transformer example

```haskell
instance Monoid b => Monoid (a -> b) where
  mempty _       = mempty
  mappend f g x = mappend (f x) (g x)
```

Instance transformer example

```
instance Monoid b => Monoid (a -> b) where
  mempty _       = mempty
  mappend f g x = mappend (f x) (g x)
```

```
$fMonoid(->) :: forall b_apJ a_apK.
  Monoid b_apJ => Monoid (a_apK -> b_apJ)
$fMonoid(->) =
  \ (@ b_awV) (@ a_awW)
    ($dMonoid_awX :: Monoid b_awV) ->
    C:Monoid @ (a_awW -> b_awV)
      (\ _ -> mempty @ b_awV $dMonoid_awX)
      ($cmappend_rxA @ b_awV @ a_awW $dMonoid_awX)
```

Instance transformer example

```haskell
instance Monoid b => Monoid (a -> b) where
  mempty _       = mempty
  mappend f g x = mappend (f x) (g x)
```

```
$fMonoid(->) :: forall b_apJ a_apK.
  Monoid b_apJ => Monoid (a_apK -> b_apJ)
$fMonoid(->) =
  \ (@ b_awV) (@ a_awW)
    ($dMonoid_awX :: Monoid b_awV) ->
    C:Monoid @ (a_awW -> b_awV)
      (\ _ -> mempty @ b_awV $dMonoid_awX)
      ($cmappend_rxA @ b_awV @ a_awW $dMonoid_awX)
```

- Turns into a function on dictionaries.
- Part of the definition is in a helper function `$cmappend_rxA` not shown here.

Map fusion example

```haskell
mapTwice :: (a -> a) -> [a] -> [a]
mapTwice f = map f . map f
```

Map fusion example

```
mapTwice :: (a -> a) -> [a] -> [a]
mapTwice f = map f . map f
```

With `-O0`:

```
mapTwice :: forall a_aoX.
  (a_aoX -> a_aoX) -> [a_aoX] -> [a_aoX]
mapTwice =
  \ (@ a_awj) (f_aoY :: a_awj -> a_awj) ->
    . @ [a_awj]
      @ [a_awj]
      @ [a_awj]
      (map @ a_awj @ a_awj f_aoY)
      (map @ a_awj @ a_awj f_aoY)
```

Note that `.` is the composition operator in prefix use.

Map fusion example

```haskell
mapTwice :: (a -> a) -> [a] -> [a]
mapTwice f = map f . map f
```

With `-O`:

```
mapTwice :: forall a_aqg.
  (a_aqg -> a_aqg) -> [a_aqg] -> [a_aqg]
mapTwice =
  \ (@ a_axC) (f_aqh :: a_axC -> a_axC)
    (eta_B1 :: [a_axC]) ->
    map
      @ a_axC @ a_axC
      (\ (x_ayc :: a_axC) -> f_aqh (f_aqh x_ayc)) eta_B1
```

The two `map` invocations have been successfully fused (via a GHC rewrite rule about which we might learn more later).

Summary

- By looking at GHC Core, we can see how GHC simplifies various language constructs.
- The main use of looking at Core in practice is to observe and check how GHC optimises code.