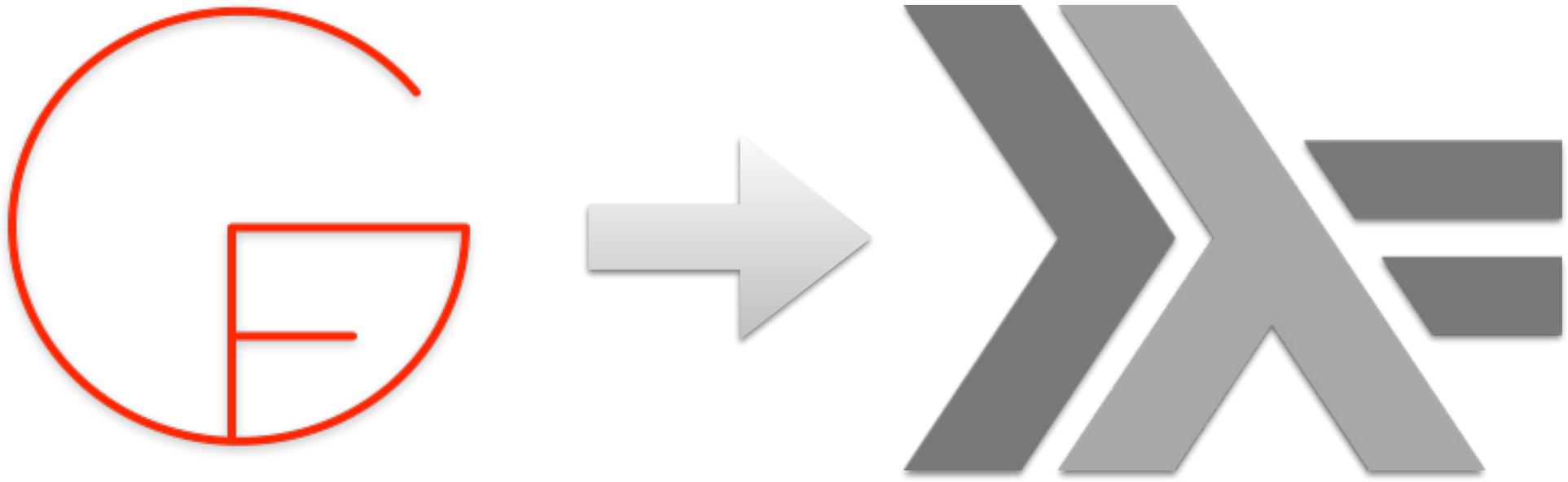

# gf -output-format=haskell -haskell=concrete

Translating GF linearization functions to Haskell

# Why?

- To use GF grammars for localisation of Haskell programs
  - Create a self-contained executable
  - Don't need to read a separate PGF file and pass around the PGF structure everywhere.
- Apply Haskell tools & libraries to GF grammars...

# Example

```
import Foods
import qualified FoodsBul as Bul
import qualified FoodsEng as Eng
import qualified FoodsFre as Fre
import qualified FoodsGer as Ger
import qualified FoodsSwe as Swe
import PGF.Haskell

main =
    do putStrLn (fromStr (Bul.linComment tree1))
       printLin (Eng.linComment tree1)
       printLin (Fre.linComment tree1)
       printLin (Ger.linComment tree1)
       printLin (Swe.linComment tree1)

printLin lin = putStrLn (fromStr (proj_s lin))

tree1 = GPred (GThis (GMod GItalian GWine)) (GVery GDelicious)
```

# GF vs Haskell

- GF and Haskell are functional languages with many similarities
  - Lambda calculus
  - Higher order functions
  - Parametric polymorphism
  - Data types & pattern matching
- But several GF language features have no direct correspondence in Haskell (and vice versa)

# GF Language Features

- Parameterized modules (functors)
- Parameter types and tables
- Anonymous record types
- Record subtyping
- Record extension
- Pattern matching with regular expressions
- Variants

# Normal grammar compilation

- GF source code is compiled into PGF (Portable Grammar Format)
- PGF represent concrete syntax using PMCFG (Parallel Multiple Context-Free Grammars)
  - Nonterminals produce tuples of strings...
- Partial evaluation is used to eliminate most high-level language features and abstractions.

# What remains after partial evaluation

- ~~Parameterized modules (functors)~~
- Parameter types and tables
- Anonymous record types
- Record subtyping
- ~~Record extension~~
- ~~Pattern matching with regular expressions~~
- Variants

# What remains after partial evaluation

- Parameter type definitions
- A flat list of linearisation functions:
  - Self-contained, monomorphic, non-recursive first-order functions
    - Anonymous records, tables, record subtyping, strings, variants
    - To be precise, strings are sequences of tokens and there are some special tokens

# Tokens in GF

- Regular tokens, e.g. `"this"`, `"fish"`
- `pre {"a"|"e"|"o"|"u" => "an"; _ => "a"}`
- Token gluing: `BIND`, `SOFT_BIND`
- `CAPIT`

# Tokens in Haskell

- `type Str = [Tok]`

- ```
  data Tok = TK String
           | TP [([String],Str)] Str
           | BIND
           | SOFT_BIND
           | CAPIT
  ```

- `fromStr :: Str -> String`
  ```

# Parameter and tables in GF

- `param Gender = Utr | Neutr`
- `param GenNum = SG Gender | GPl`
- `param AFormPos = Strong GenNum | Weak Number`

- ```
  table { GSg Utr   => “röd”;
          GSg Neutr => “rött”;
          GPl       => “röda” } : AFormPos => Str
  ```
- Normally compiled by GF to:

  ```
  table AFormPos [“röd”;”rött”;”röda”]
  ```

- Tables can become large and contain a lot of repetition
  ```

# Parameters and tables in Haskell

- Parameter types translated to the corresponding data types
- `class EnumAll a where enumAll :: [a]`
- Generate an `instance EnumAll P` for every parameter type `P`
- Use finite maps for tables: `A=>B` becomes `Map A B`
- ```
  fromList (zip enumAll [“röd”,”rött”,”röda”])
      :: Map AFormPos String
  ```
- To reduce code size, duplicate entries are detected and lifted out into a `let ... in ...`
  ```

# Anonymous record types

- Example  in GF:
  - `Utterance = { s : Str }`
    `Noun = { s : Str; g:Gender }`
  - `car_N = { s=“bil”; g=Utr } : Noun`
- The same label can appear in many different record types
- Translation to Haskell:
  - Traverse partially evaluated code and extract all record types.
  - Generate a data type for each combination of labels found, e.g.
    - `data R_g_s a b = R_g_s a b`
    - `type Noun = R_g_s Gender Str`
    - `car_N = R_g_s Utr “bil” :: Noun`

# Anonymous record types

- Projection in GF uses dot notation
  - `car_N.s`
  - Translation to Haskell uses overloading to create projection functions that work for many different record types:
    - Generate a class for each record label found, e.g.
      - ```
        class Has_s r a | r->a where proj_s :: r -> a
        class Has_g r a | r->a where proj_g :: r -> a
        ```
    - and instances for each record type, e.g.
      - ```
        instance Has_s (R_g_s a b) b where ...
        instance Has_g (R_g_s a b) a where ...
        ```
- `proj_s car_N`

# Record subtyping

- GF has record subtyping, e.g.
  - `{s:Str; g:Gender}` ≤ `{s:Str}`
- If A≤B then something of type A can be used wherever something of type B is expected
- Translation to Haskell replaces subtyping with overloading:
  - Insert an explicit coercion when the type of a subexpression doesn't match the expected type
    - Keep track of the expected type when descending terms
  - Generate coercion function for each record type, e.g.
    - ```
      to_R_g_s :: (Has_g r a, Has_s r b) => r -> R_g_s a b
      to_R_g_s r = R_g_s (proj_g r) (proj_s r)
      ```
      ```

# Example: Foods.gf

```
abstract Foods = {

flags startcat = Comment;

cat Comment; Item; Kind; Quality;

fun Is : Item -> Quality -> Comment;
    This : Kind -> Item;
    Cheese : Kind;
    Fish : Kind;
    Wine : Kind;
    Fresh : Quality;
}
```

# Example: Foods.hs

```
module Foods where

data GComment = GIs GItem GQuality
  deriving Show

data GItem = GThis GKind
  deriving Show

data GKind = GCheese| GFish| GWine
  deriving Show

data GQuality = GFresh
  deriving Show
...
```

# Example: FoodsEng.gf

```
concrete FoodsEng of Foods = {

lincat
     Comment = Str;
     Item = Str;
     Kind = Str;
     Quality = Str;

lin  Is item quality = item ++ "is" ++ quality ;
     This kind = "this" ++ kind;
     Cheese  = "cheese";
     Fish  = "fish";
     Wine  = "wine";
     Fresh  = "fresh";
}
```

# Example: FoodsEng.hs

```
module FoodsEng where
import qualified Foods as A
...
type LinComment = Str
type LinItem = Str
type LinKind = Str
type LinQuality = Str

linComment :: A.GComment -> LinComment
linItem :: A.GItem -> LinItem
linKind :: A.GKind -> LinKind
linQuality :: A.GQuality -> LinQuality

linComment (A.GIs abs_gItem_0 abs_gQuality_1) =
  let gItem_0 = linItem abs_gItem_0
  in let gQuality_1 = linQuality abs_gQuality_1
     in gItem_0 ++ (TK "is" : gQuality_1)
linItem (A.GThis abs_gKind_0) =
  let gKind_0 = linKind abs_gKind_0 in TK "this" : gKind_0
linKind A.GCheese = [TK "cheese"]
linKind A.GFish = [TK "fish"]
linKind A.GWine = [TK "wine"]
linQuality A.GFresh = [TK "fresh"]
```

# Example: FoodsSwe.gf

```
concrete FoodsSwe of Foods = {

lincat
    Comment = Str;
    Item = {s:Str;g:Gender};
    Kind = {s:Str;g:Gender};
    Quality = Gender => Str;

lin Is item quality = item.s ++ "är" ++ quality!item.g;
    This kind = {s=denna!kind.g ++ kind.s; g=kind.g};
    Cheese    = {s="osten"; g=En};
    Fish      = {s="fisken"; g=En};
    Wine      = {s="vinet"; g=Ett};
    Fresh     = table {En=>"färsk"; Ett=>"färskt"};

param Gender = En | Ett;

oper denna : Gender => Str =
       table {En=>"den här"; Ett=>"det här"};
}
```

# Example: FoodsSwe.hs

```
module FoodsSwe where
import qualified Foods as A
...
data GFoodsSwe_Gender = GFoodsSwe_En | GFoodsSwe_Ett deriving (Eq,Ord,Show)
instance EnumAll GFoodsSwe_Gender where enumAll = [GFoodsSwe_En, GFoodsSwe_Ett]

type LinComment = Str
type LinItem = R_g_s GFoodsSwe_Gender Str
type LinKind = R_g_s GFoodsSwe_Gender Str
type LinQuality = GFoodsSwe_Gender -> Str

linComment :: A.GComment -> LinComment
linItem :: A.GItem -> LinItem
linKind :: A.GKind -> LinKind
linQuality :: A.GQuality -> LinQuality

linComment (A.GIs abs_gItem_0 abs_gQuality_1) =
  let gItem_0 = linItem abs_gItem_0
  in let gQuality_1 = linQuality abs_gQuality_1
     in proj_s gItem_0 ++ (TK "är" : gQuality_1 (proj_g gItem_0))
linItem (A.GThis abs_gKind_0) =
  let gKind_0 = linKind abs_gKind_0
  in R_g_s
       (proj_g gKind_0)
       (table
          [[TK "den", TK "här"], [TK "det", TK "här"]] (proj_g gKind_0) ++
          proj_s gKind_0)
linKind A.GCheese = R_g_s GFoodsSwe_En [TK "osten"]
linKind A.GFish = R_g_s GFoodsSwe_En [TK "fisken"]
linKind A.GWine = R_g_s GFoodsSwe_Ett [TK "vinet"]
linQuality A.GFresh = table [[TK "färsk"], [TK "färskt"]]
```

# Example: FoodsSwe.hs

```
--- Type classes for projection functions ---
class Has_g r a | r -> a where proj_g :: r -> a

--- Record types ---
data R_g_s t1 t2 = R_g_s t1 t2 deriving (Eq,Ord,Show)
instance (EnumAll t1, EnumAll t2) => EnumAll (R_g_s t1 t2) where
    enumAll = (R_g_s <$> enumAll) <*> enumAll

instance Has_g (R_g_s t1 t2) t1 where proj_g (R_g_s t1 t2) = t1
instance Has_s (R_g_s t1 t2) t2 where proj_s (R_g_s t1 t2) = t2

to_R_g_s r = R_g_s (proj_g r) (proj_s r)
```

# Variants

- Translation to Haskell?
- Terms may denote many values
    - Impure extension of the lambda calculus
        - `let s = "a"|"b" in s++s`
    - Semantics?
- Translation to Haskell?
    - Lift everything into the list monad

# FoodsEng.hs with variants

```
module FoodsEng where
import qualified Foods as A
import PGF.Haskell
...
type LinComment = Str
type LinItem = Str
type LinKind = Str
type LinQuality = Str

linComment :: A.GComment -> [LinComment]
linItem :: A.GItem -> [LinItem]
linKind :: A.GKind -> [LinKind]
linQuality :: A.GQuality -> [LinQuality]

linComment (A.GIs abs_gItem_0 abs_gQuality_1) =
  let gItem_0 = linItem abs_gItem_0
  in let gQuality_1 = linQuality abs_gQuality_1
     in gItem_0 +++ ([[TK "is"]] +++ gQuality_1)
linItem (A.GThis abs_gKind_0) =
  let gKind_0 = linKind abs_gKind_0 in [[TK "this"]] +++ gKind_0
linKind A.GCheese = [[TK "cheese"]]
linKind A.GFish = [[TK "fish"]]
linKind A.GWine = [[TK "wine"]]
linQuality A.GFresh = [[TK "fresh"]]
```

# A larger example: Phrasebook

- 23 languages, 9,800 lines of GF code
  - Also imports the RGL
    - Phrasebook.pgf is 22MB
- Translates into 2,400,000 lines (151MB) of Haskell code
  - Largest languages: Snd, Urd, Hin, Fre, Pes, Fin
  - Smallest languages: Tha, Chi, Swe, Dan, Nor, Eng

# Implementation

- GF.Compile.ConcreteToHaskell: 435 lines

- GF.Haskell: 146 lines

# Possible future work

- Preserve/rediscover more abstractions to reduce code size
- Represent tables as arrays (generate `Ix` instances)
- Parsing?