Bunny コードリーディング diary （0.9.0 版）

2021-01-19 (Tue)

Kind Inference

Haskell 2010 の Section 4.6[*1] をちゃんと読んでみよう。

本節では Kind の推定、つまり、与えられたプログラムに出現する型構築子やクラスのそれぞれについて適切な Kind を計算するために用いられる規則について述べる。

Kind 推定の最初のステップは、データ型、シノニム、および、クラス定義の集合を、依存グループに分類することである。これは、4.5 節で述べられている値定義の依存解析とほぼ同様の方法で実現可能である。たとえば、以下のプログラム片には、データ型構築子 D、シノニム S、および、クラス C が含まれるが、これらはすべて同一の依存グループに属する：

```
data C a => D a = Foo (S a)
type S a = [D a]
class C a where
    bar :: a -> D a -> Bool
```

各グループにおける変数、構築子、および、クラスの Kind は、型推論と Kind を保った同一化に関する標準的な技法[*2] によって決定できる。たとえば、上の例におけるパラメータ `a` は、`bar` の型において関数構築子 `->` の引数となっており、したがってその Kind は `*` でなければならない。さらに、D と S の種は `* -> *` でなくてはならず、クラス C のインスタンスはいずれも Kind `*` をもつ。

It is possible that some parts of an inferred kind may not be fully determined by the corresponding definitions; in such cases, a default of is assumed. For example, we could assume an arbitrary kind κ for the a parameter in each of the following examples:

```
data App f a = A (f a)
data Tree a  = Leaf | Fork (Tree a) (Tree a)
```

This would give kinds $(\kappa \to) \to \kappa \to$ and $\kappa \to$ for App and Tree, respectively, for any kind κ, and would require an extension to allow polymorphic kinds. Instead, using the default binding $\kappa =$, the actual kinds for these two constructors are $(\to) \to\to$ and $\to$, respectively.

Defaults are applied to each dependency group without consideration of the ways in which particular type constructor constants or classes are used in later dependency groups or elsewhere in the program.

[*1] https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-970004.6

[*2] MP Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. Journal of Functional Programming, 5(1):136, January 1995.

For example, adding the following definition to those above does not influence the kind inferred for Tree (by changing it to (→) →, for instance), and instead generates a static error because the kind of `[]`, →, does not match the kind that is expected for an argument of Tree:

```
type FunnyTree = Tree []      -- invalid
```

This is important because it ensures that each constructor and class are used consistently with the same kind whenever they are in scope.

kiExpr