# Agda: Typing Types with Universes

Andreas Abel

2 July 2012

## 1 Types of Types

In Haskell, programs and types are separate, with a different syntax for each of the two concepts. There is a special declaration form

```
type State s a = s → (a, s)
```

for defining type synonyms. Further, type synonyms must always be fully applied, i.e., we cannot use `State s` by itself.

In Agda, the situation is different: There is only one syntax for programs and types, which means that types and programs can be arbitrarily mixed and refer to each other. Consequently, we can use the syntax for defining functions also to define type synonyms:

```
State S A = S → A × S
```

Each function declaration must come with a type signature in Agda, so what should the type of `State` be? `State S A` is a type itself, so we need *types of types*. Agda's `Set` is such a type of types.

The two arguments `S` and `A` of `State` are also types, so we can view `State` as a function, taking two types as input and returning another type as output.

```
open import Data.Product
State : Set → Set → Set
State S A = S → A × S
```

Since types can be arbitrary expressions, we can also define `State` as a $\lambda$-abstraction, which is not possible in Haskell.

```
State : Set → Set → Set
State = λ S A → S → A × S
```

If we tell Agda the types of the $\lambda$-abstracted variables, we can even omit the type signature for `State`.

```
State = λ (S A : Set) → S → A × S
```

## 2 Haskell's Types of Types: Kinds

Haskell's equivalent to Agda's Set is *, called the *kind* of types. GHC's extensions allow to give the kind of type variables in type definitions.

```
{-# LANGUAGE KindSignatures #-}
type State (s :: *) (a :: *) = s → (a, s)
```

This says that s and a are *types*. State itself is not a type but a *type constructor*: It takes two type arguments and only then returns a type.

Usually, Haskell infers the kind of a type variable automatically, but there are situations where there is not information. For instance, in the definition

```
data Const m a = C a
```

the variable m does not appear on the right hand side of =, so its type is unspecified. If we ask GHC for the Kind of Const

```
*Main> :k Const
Const :: * -> * -> *
```

we see that the kind of m has been defaulted to *. If we want anything fancier, we have to specify the kind explicitly.

```
data Const (m :: * → *) a = C a
```

```
*Main> :k Const
Const :: (* -> *) -> * -> *
```

Kind signatures can be given wherever a type variable is bound, e.g., also in class definitions:

```
class Mond (m :: * → *) where
  neverReturn :: a → m a
  unbounded :: m a → (a → m b) → m b
```

## 3 Existential Types in Haskell

Type classes are Haskell's tool of choice to implement data abstraction. For instance, an *interface* for stacks can be given by the following 2-parameter type class.

```
{-# LANGUAGE MultiParamTypeClasses #-}
class Stack s a where
  empty :: s
  push  :: a → s → s
```

```
pop    :: s → s
top    :: s → Maybe a
```

However, sometimes we would rather need Stack to be a type, but do not want to reveal its implementation. This would be called an *abstract type*.

*Existential types* are an alternative to type classes when you want to work with abstract types. Basically, want we want to say is that there *exists* a type s which implements the stack operations, but we are not telling you which type it is. GHC allows existential data types, but paradoxically they are written with the **forall**-syntax.

```
{-# LANGUAGE ExistentialQuantification #-}
data AbstractStack a = forall s. AbstractStack
  { empty :: s
  , push  :: a → s → s
  , pop   :: s → s
  , top   :: s → Maybe a
  }
```

This defines a record AbstractStack a with constructor AbstractStack and four fields empty, push, pop and top, which are the four stack operations exposed by this interface.

The use of **forall** is a language design flaw, but it can be justified by the type of the constructor.

```
AbstractStack ::
      s
  → (a → s → s)
  → (s → s)
  → (s → Maybe a)
  → AbstractStack a
```

which is actually equivalent to

```
{-# LANGUAGE RankNTypes #-}
AbstractStack :: forall s.
      s
  → (a → s → s)
  → (s → s)
  → (s → Maybe a)
  → AbstractStack a
```

Yet, in uncurried form, the constructor would receive type (not valid Haskell syntax!)

```
AbstractStack ::
  exists s. ( s
            ,(a → s → s)
```

```
        , (s → s)
        , (s → Maybe a)
        )
    → AbstractStack a
```

which justifies the name *existential type*.

Any function working with abstract stacks must receive the implementation as one of its arguments.

```
{-# LANGUAGE NamedFieldPuns #-}
testStack :: AbstractStack Int → Int
testStack (AbstractStack {empty, push, pop, top}) =
  fromJust $ top $ pop $ push 2 $ push 4 $ empty
```

To run a function using the abstract stack interface, we need a concrete implementation, of course! How about just lists:

```
topList [] = Nothing
topList (x : xs) = Just x
listStack :: AbstractStack a
listStack = AbstractStack {empty = [], push = (:), pop = tail, top = topList}
test = testStack listStack
```

# 4 Existential Types in Agda

In Agda, existential quantification is just an instance of the Σ-type. It is defined in Data.Product, but we first look at a simpler implementation:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

Σ A B is just the type of **record**s with two fields, the first, $\mathsf{proj}_1$ being of type A, and the second, $\mathsf{proj}_2$ being of type B indexed by the first field. Thus, it is a *dependently typed record* where the type of a field (here: $\mathsf{proj}_2$) can depend on the value of another field (here: $\mathsf{proj}_1$) which comes earlier in the record. As an example, we can define a type of sequences

```
open import Data.Nat
open import Data.Product
open import Data.Vec
```

```
Seq : Set → Set
Seq A = Σ ℕ (Vec A)
```

which contains pairs $(\mathsf{n}, \mathsf{v})$ of a natural number n and a vector v of length n.

An interface IsStack S A which expresses that S is a type that supports stack operations for elements of type A, is expressed as a record in Agda.

```
open import Data.Maybe
record IsStack (S A : Set) : Set where
  field
    empty : S
    push  : A → S → S
    pop   : S → S
    top   : S → Maybe A
```

We can implement this interface via lists, just as in Haskell:

```
open import Data.List
popList : {A : Set} → List A → List A
popList []       = []
popList (x :: xs) = xs
topList : {A : Set} → List A → Maybe A
topList []       = nothing
topList (x :: xs) = just x
listStack : {A : Set} → IsStack (List A) A
listStack = record
  {empty = []
  ; push  = _::_
  ; pop   = popList
  ; top   = topList
  }
```

## 5 A Problem and its Solution via Universes

But when we use our own rolling of Σ to form the existential type of abstract stacks, we are in for a surprise.

```
AbstractStack : Set → Set
AbstractStack A = Σ Set (λ S → IsStack S A)
```

Agda complains that

```
Set₁ != Set
when checking that the expression Set has type Set
```

We have attempted to use Set, the type of types, as type itself. In other words, we have expected Set : Set, i.e., Set, the type of types, is itself a type. However, it is known from set theory that if you form a "set of all sets" you get into trouble, because then you can talk about sets that contain themselves. Ponder a bit about Russell's paradox: Let $R$ be the set of all sets that do not contain themselves. Does $R$ contain itself?

There are two ways out. First, we can ignore the problem and work in a paradoxical type theory. In Agda this can be done via

```
{-# OPTIONS –type-in-type #-}
```

and then Agda happily accepts our construction. Note however, that Agda is inconsistent now; every proposition, even a false one, is now provable, provided you know how to exploit the paradox!

The other solution is *universes*. A universe is a set of types, and our first universe is Set = Set0. It contains all types that do not mention Set itself. The next universe is Set1 which contains Set, thus Set : Set1, and all types that do or do not mention Set, but no type that mentions Set1. We go on to construct Set2 which contains types that may mention Set and Set1, in particular, Set1 : Set2. And so on.

Now we can form our existential type with version of $\Sigma$ that accepts Set as first argument.

```
record Σ₁,₀ (A : Set₁) (B : A → Set₀) : Set₁ where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
AbstractStack : Set → Set
AbstractStack A = Σ₁,₀ Set (λ S → IsStack S A)
```

An alternative is, of course, to spell out the record. A record which contains a Set as one of its fields must live in Set1 itself!

```
record AbstractStack (A : Set) : Set₁ where
  field
    Stack : Set
    isStack : IsStack Stack A
  open IsStack isStack public
```

Here, the **open** IsStack isStack **public** makes the fields empty, push of IsStack etc. available as fields of AbstractStack, which saves us from the overhead of nested records. Check the fields of a record with `C-c C-o`.

# 6 Universe Polymorphism