Subtypes for Free!

George Stelle and Stephanie Forrest

Motivation

It would sometimes be nice if a type system could be as precise as possible to restrict what a value will be. For example, the type `Bool` ensures a value will either be `true` or `false`, but it doesn't know which. What we want is the type system to be precise when possible, so instead of always saying `Bool` (or "I don't know"), it could say `True`, `False`, or `Bool`. In this example, `True` and `False` are *subtypes* of `Bool`, i.e. every value of type `True` is also of type `Bool`.

Existing Approaches

There is a significant literature on subtyping, mostly in the context of object-oriented (OO) languages. Indeed, subtyping in languages that combine OO features with functional features implement subtypes using objects [**?**, **?**]. We are interested in subtyping without requiring all the language complexities introduced by OO constructs, and will see that doing so gives us some inference advantages.

Generalized algebraic data types (GADTs) allow for a restricted form of subtyping as well. For example, the canonical example of `Expr Int` as separated from `Expr Bool` in a simple language with booleans and integers. Our approach subsumes GADTs, while adding extra flexibility in how we can represent subtypes.

Our Approach

Our approach is to use Scott encodings of algebraic data types, along with Hindley Milner type inference, to achieve a very general form of subtype polymorphism. We show how GHC can take advantage of this approach without modification by ~~abusing~~ taking advantage of impredicative types.

Scott encodings are encodings of algebraic data types that encode their own `case` destruction. For booleans, for example, the encoding is identical to Church encoding: `true t f = t`, and `false t f = f`.

Our primary insight is that by wrapping the scott encodings in a `newtype`, and then carefully constraining the types, we can define types that represent arbitrary subsets of the constructors.

One appealing thing about our approach is that it composes well. For example, we are not aware of any other subtyping scheme that is capable of inferring that

```
null nil :: True
```

where `null` checks if a list is empty and `nil` is the empty list.

An Example

One application of subtyping is to prevent partial functions. Here we show how to use our approach to define a total version of the `fromJust` from the Haskell standard library.

To achieve this, we define a subtypeable `Maybe`. The Scott encoding for `Maybe` takes two *case* arguments, the first, `n`, corresponding to `nothing`, so it has no parameters, and the second, `j`, corresponding to `just a`, so it has one parameter of type `a`.

```
newtype Maybe' a n j m = Maybe (n -> (a -> j) -> m)
type Maybe a = forall m. Maybe' a m m m
type Just a = forall n j. Maybe' a n j j
just :: a -> Just a
just a = Maybe $ \n j -> j a
type Nothing = forall a n j. Maybe' a n j n
nothing :: Nothing
nothing = Maybe $ \n j -> n
```

For our `Maybe` type, we have a type like the standard `Data.Maybe` type in Haskell. In this case, similar to the `maybe` function from `Data.Maybe`, we don't know whether we have a value of type `Just` or a value of type `Nothing`, so we must ensure that the two cases return the same type. But if the compiler can infer that the value is of type `Just`, we don't care what the `n` cases type is, and only constrain the return type to be the same as the `Just` case `j`.

With that in mind, we can write our total function, using a `Bottom` type with no values to convince ourselves that our function will never typecheck `fromJust` applied to a type that includes `Nothing`.

```
fromJust :: Just a -> a
fromJust (Maybe j) = j bottom $ \a->a

data Bottom
bottom = error "impossible" :: Bottom
```

We can still define the partial version of `fromJust`, which like the one from the Haskell standard library, allows runtime failure.

```
partialFromJust :: Maybe a -> a
partialFromJust (Maybe m) = m (error "partialFromJust Nothing") $ \a->a
```

We have implemented more sophisticated examples, including recursive types, GADTs, and total versions of `head` and `tail`. For these, as well as an implementation of Eric Lippert's wizards and warriors example, see: http://cs.unm.edu/~stelleg/scott/.

Drawbacks

Without language support, this approach is quite verbose and unwieldy. Furthermore, due to the nature of Scott encodings, the constructor definitions grow quadratically in size as a function of the number of constructors. Because of the large number of nested `forall`s, getting variable quantification right can also be quite difficult.

Future Work

We are currently working on formalizing and verifying this approach in Coq using parametricity [**?**]. The burden of creating all of the necessary type synonyms could likely be lessened by some template Haskell. Further in the future, it would be interesting to see the ideas integrated as an extension to GHC.