Generic and Flexible Defaults for Verified, Law-Abiding Type-Class Instances (Appendix)

Ryan G. Scott
Indiana University
United States
rgscott@indiana.edu

Ryan R. Newton
Indiana University
United States
rrnewton@indiana.edu

CCS Concepts • **Software and its engineering → Functional languages**; *Data types and structures.*

Keywords Type classes, generic programming, dependent types, reuse

ACM Reference Format:
Ryan G. Scott and Ryan R. Newton. 2019. Generic and Flexible Defaults for Verified, Law-Abiding Type-Class Instances (Appendix). In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3331545.3342591

A Appendix

A.1 The Singletons Encoding

The code in this paper strongly resembles Haskell, but with the addition of several dependently typed features. How is this possible, given that Haskell—or, more specifically, its flagship implementation, the Glasgow Haskell Compiler (GHC)— isn't a dependently typed language? The trick is to use *singleton types*, as popularized in Eisenberg and Weirich [4].

While GHC does not support the full spectrum of features one might find in a typical dependently typed language, it does offer many language extensions that enrich its type system. As one example, the *data kinds* extension allows one to use data constructors at the type level instead of merely the value level [7]. Combining these extensions yields an encoding that can very convincingly simulate the experience of writing dependently typed code.

This section serves as an outline as a field guide to translating the dependently typed Haskell code in the paper to singletons code that one could actually run on a modern version of GHC. Many of the definitions in this section are taken from the `singletons` Haskell library, which serves to make using this encoding slightly easier.

Haskell '19, August 22–23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342591

A.1.1 Functions at the Type Level

Consider a simple function that negates a boolean value:

```
not :: Bool → Bool
not True  = False
not False = True
```

In today's GHC, this defines a function that exists only in the value namespace, and any attempt to use `not` at the type level simply results in an error. How, then, can we justify writing code in the paper that uses `not` at the type level, such as in Section 2.2.1 [6]? The trick is to use GHC's *type families* extension [1, 3], which allows defining functions for use at the type level. For example, here is `not` formulated as a type family:

```
type family Not (x :: Bool) :: Bool where
  Not True  = False
  Not False = True
```

Type families—along with data kinds, which let us use `Bool` constructors at the type level—let us define a type-level `Not` in much the same way that we would write the value-level `not`. One slightly annoying detail is that type families must begin with a capital letter, so we must have a slightly different name for the type family, but this is only a minor hiccup.

To make life somewhat easier, the `singletons` library offers metaprogramming support for taking a value-level definition and automatically generating its type family equivalent, a process that the library calls *promotion* [2]. Therefore, when we use value-level functions at the type level in the Haskell code of this paper, one can imagine that we had promoted the function beforehand and are actually using the promoted name.

Type class methods can be promoted as well, taking advantage of the fact that type families can be associated with classes. For example, in this class with a single method:

```
class Collect c where
  emptyCollect :: c
```

We can promote it to the following class with a single associated type family, adopting the convention that promoted class names start with a `P`:

```
class PCollect c where
  type family EmptyCollect :: c
```

Note that the `c` in the kind of `EmptyCollect` is a *kind* variable, which makes use of GHC's faculties for kind polymorphism. [7]

A.1.2 Dependent Pattern Matching

In order to use functions at the type level profitably, it is not enough to just promote the functions. Consider the following function in dependent Haskell:

```
foo :: Π (b :: Bool) → if b then Int else ()
foo True  = 42
foo False = ()
```

In order to write this function in GHC, we first need to define an `If` type family, which we can do using the techniques in Section A.1.1. But there is still another obstacle: how do we encode the Π quantifier, which permits dependent pattern matching?

This is where singleton types become useful. Given a datatype that we wish to dependently match on, we can define an isomorphic singleton type. For instance, this is the singleton type for `Bool`s:

```
data SBool :: Bool → Type where
  SFalse :: SBool False
  STrue  :: SBool True
```

The `SBool` datatype is indexed by a `Bool`, which makes it valuable for carrying around type-level information about that `Bool`. For instance, matching on `STrue` brings into scope evidence that the `Bool` is equal to `True` at the type level, which can be used to help guide along evaluation of functions at the type level. Here is an example of `SBool`s in action:

```
foo :: SBool b → If b Int ()
foo STrue  = 42
foo SFalse = ()
```

When `foo` matches on `STrue`, it reveals that `b` is equal to `True`, which causes `If b Int ()` to reduce to `Int`, which is what permits the right-hand side of the equation to typecheck. (And similarly for `foo`'s second equation). Because there are so many singleton types in existence, the `singletons` library defines a `Sing` type family that provides a common name for all singleton types. Therefore, an equivalent way of writing the type of `foo` is:

```
foo :: Sing (b :: Bool) → If b Int ()
```

When written this way, the similarities between singleton types and the Π quantifier are even more apparent[1]. Whenever Haskell code in this paper features uses of Π, we are really using `Sing` under the hood.

The `singletons` library also automates some of the tedium of using singleton types. The library offers metaprogramming support for taking a datatype declaration and generating its singleton type as an output, a process the library refers to as *singling*. The library also supports singling function definitions. For instance, this is the singled version of the `not` function:

```
sNot :: Sing (b :: Bool) → Sing (Not b)
sNot STrue  = SFalse
sNot SFalse = STrue
```

The singled versions of functions can be useful in contexts that require lots of type-level computation (e.g., proofs), so the code in this paper implicitly uses many singled definitions.

In addition to top-level functions, type class methods can also be singled. We adopt the convention that the names of singled classes start with an `S`. For example, here is the singled version of `Collect`:

```
class SCollect c where
  sEmptyCollect :: Sing (EmptyCollect :: c)
```

A.1.3 Partially Applied Functions

The tricks in Sections A.1.1 and A.1.2 encode 90% of the dependently typed Haskell code in this paper. The remaining 10% concerns the use of partially applied functions, something which Haskell excels at the value level but is less adept at doing at the type level. To illustrate what the issue is, consider the following functions:

```
notList :: [Bool] → [Bool]
notList bs = map not bs

map :: (a → b) → [a] → [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

If we wish to promote `notList` to the type level, then we first need to promote `map`. Here is naïve first attempt:

```
type family Map (f :: a → b) (x :: [a]) :: b where
  Map _ '[]    = '[]
  Map f (x:xs) = f x : Map f xs
```

Although this type family typechecks, one cannot define `NotList bs = Map Not bs`. The reason this does not work is because GHC requires that all type families appear fully saturated, with all of its arguments supplied. This restriction is in place because the ability to abstract over unapplied type families would wreak havoc with GHC's type inference engine. [2] The bottom line is that `Map Not bs` is disallowed, since `Not` is partially applied.

Fortunately, there exists a way to work around this restriction. The `singletons` library offers a way to defunctionalize [5] type families using opaque symbols, which can be explicitly applied to arguments using the application operator `Apply`. For instance, here is the defunctionalization symbol for the `Not` type family:

[1] Some even go as far as defining `type Π = Sing`!

[2] See Section 7.1 of Eisenberg and Stolarek [2] for an extended discussion on this point.

```
data NotSym0 :: a ⇝ b
```

The use of "Sym0" indicates that this is a defunctionalization symbol which is applied to zero arguments. Moreover, its kind `a ⇝ b` indicates that it can be futher applied to one more argument of type `a`, returning something of type `b`. The ⇝ arrow is the kind of defunctionalization symbols:

```
data TyFun :: Type → Type → Type
type a ⇝ b = TyFun a b → Type
```

Finally, in order to be able to use `NotSym0` meaningfully, one must be able to `Apply` it. This can be done by writing an instance of the `Apply` type family, the definition of which (and an example instance) are reproduced below:

```
type family Apply (f :: a ⇝ b) (x :: a) :: b
type instance Apply NotSym0 b = Not b
```

Using defunctionalization, we can write a version of `Map` that accepts a defunctionalization symbol as its first argument:

```
type family Map (f :: a ⇝ b) (x :: [a]) :: b where
  Map _ '[]    = '[]
  Map f (x:xs) = Apply f x : Map f xs
```

This lets us complete our original goal, which was writing a promoted version of `notList`:

```
type family NotList (bs :: [Bool]) :: [Bool] where
  NotList bs = Map NotSym0 bs
```

This encoding is somewhat involved, but is thankfully only a mechanical change that can be automated. In fact, the `singletons` library generates code using ⇝ instead of → whenever function types are used in higher-order positions, and this allows it to generate the definitions of `NotList` and `Map` that we wrote above. Moreover, the library also generates all of the defunctionalization symbols for each function, which is especially useful for functions with many arguments, as they require equally many symbols.

References

[1] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 241–253. https://doi.org/10.1145/1086365.1086397

[2] Richard A. Eisenberg and Jan Stolarek. 2014. Promoting Functions to Type Families in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 95–106. https://doi.org/10.1145/2633357.2633361

[3] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 671–683. https://doi.org/10.1145/2535838.2535856

[4] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2364506.2364522

[5] John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2 (ACM '72)*. ACM, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[6] Ryan G. Scott and Ryan R. Newton. 2019. Generic and Flexible Defaults for Verified, Law-Abiding Type-Class Instances.

[7] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795