Dependently Typed Programming with Singletons

Richard A. Eisenberg
University of Pennsylvania
Philadelphia, PA, USA
eir@cis.upenn.edu

Stephanie Weirich
University of Pennsylvania
Philadelphia, PA, USA
sweirich@cis.upenn.edu

Abstract

Haskell programmers have been experimenting with dependent types for at least a decade, using clever encodings that push the limits of the Haskell type system. However, the cleverness of these encodings is also their main drawback. Although the ideas are inspired by dependently typed programs, the code looks significantly different. As a result, GHC implementors have responded with extensions to Haskell's type system, such as GADTs, type families, and datatype promotion. However, there remains a significant difference between programming in Haskell and in full-spectrum dependently typed languages. Haskell enforces a phase separation between runtime values and compile-time types. Therefore, singleton types are necessary to express the dependency between values and types. These singleton types introduce overhead and redundancy for the programmer.

This paper presents the singletons library, which generates the boilerplate code necessary for dependently typed programming using GHC. To compare with full-spectrum languages, we present an extended example based on an Agda interface for safe database access. The paper concludes with a detailed discussion on the current capabilities of GHC for dependently typed programming and suggestions for future extensions to better support this style of programming.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Data types and structures; D.3.2 [Programming Languages]: Language Classifications—Haskell; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

Keywords Haskell; dependently typed programming; singletons; GADTs

1. Introduction

Haskell programmers have been experimenting with rich interfaces inspired by dependently typed programming for more than a decade. The goal of these rich interfaces is both to extend the reach of Hindley-Milner type inference (e.g., for generic programming, type-safe *printf*, n-ary zips, heterogeneous lists) and to enhance the lightweight verification capabilities of the Haskell type checker (e.g., for length-indexed lists, well-typed abstract syntax). The techniques used in these examples are astoundingly diverse: from phantom types [Leijen and Meijer 1999], to nested datatypes [Bird and Paterson 1999; Okasaki 1999], to a higher-order polymorphism encoding of Leibniz equality [Baars and Swierstra 2002; Cheney and Hinze 2002; Weirich 2004], to overlapping type classes [Kiselyov et al. 2004], to a tagless algebra [Carette et al. 2009], to functional dependencies [Guillemette and Monnier 2008a; McBride 2002]. The flexibility of the Haskell type system and the ingenuity of Haskell programmers have been demonstrated beyond doubt.

However, the cleverness of these encodings is also their drawback. Although the ideas behind the encodings are inspired by dependently typed programs, the code does not *look* like code in any full-spectrum dependently typed language, such as Cayenne [Augustsson 1998], Coq [Coq development team 2004], Epigram [McBride 2004], or Agda [Norell 2007]. As a result, several authors [Guillemette and Monnier 2008b; McBride 2002; Neubauer and Thiemann 2002] have pushed for more direct mechanisms, and GHC implementors have responded with Generalized Algebraic Datatypes (GADTs) [Cheney and Hinze 2002; Peyton Jones et al. 2006; Schrijvers et al. 2009; Xi et al. 2003], type-level functions [Chakravarty et al. 2005], and type-level datatypes with kind polymorphism [Yorgey et al. 2012]. These additions provide native support for constrained data (replacing the use of phantom types, nested datatypes, and type equality encodings) and type-level computation (replacing the use of logic programming with type classes and functional dependencies).

1.1 An example of singletons

A natural number datatype

```
data Nat where
  Zero :: Nat
  Succ :: Nat → Nat
```

can now automatically be used at the type level to indicate the length of a vector represented by a GADT.

```
data Vec :: ∗ → Nat → ∗ where
  VNil :: Vec a 'Zero
  VCons :: a → Vec a n → Vec a ('Succ n)
```

(The single quotes in front of the data constructor names indicate that they were *promoted* from the expression language.)

Furthermore, *type families* can express functions on this type-level data, such as one that computes whether one natural number is less than another.

```
type family   (m :: Nat) :< (n :: Nat) :: Bool
type instance m          :< 'Zero      = 'False
type instance 'Zero      :< ('Succ n)  = 'True
type instance ('Succ m)  :< ('Succ n)  = m :< n
```

However, there is still at least one significant difference between programming in Haskell and in full-spectrum dependently typed languages. Haskell enforces a phase separation between runtime

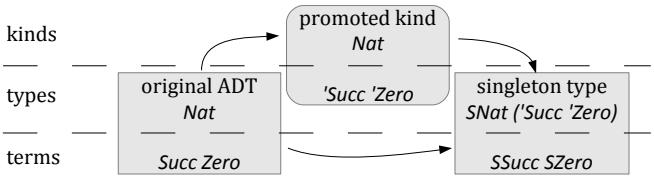

Figure 1. Singleton generation of *Succ Zero*

values and compile-time types. Consequently, to express the dependency between the value of one runtime argument and the compile-time type of another requires the definition and use of *singleton types*—types with only one non-$\bot$ value.

For example, consider the safe indexing operation for vectors below, called *nth*. This operation ensures that the index m is less than the length of the vector n with the constraint $(m :< n) \sim True$. However, because *nth* requires the index at runtime, this index cannot *only* be a type. We must also include a runtime witness for this index, called a singleton, that can be used for computation. The type of singleton values for natural numbers is *SNat*, a GADT indexed by a *type* of *kind* *Nat*.

```
data SNat :: Nat → ∗ where
  SZero :: SNat 'Zero
  SSucc :: ∀ (n :: Nat). SNat n → SNat ('Succ n)
```

A graphical schematic of the relationship between the original datatype, the promoted kind, and the derived singleton type can be seen in Figure 1. Because the constructors of *SNat* mirror those of the kind *Nat*, only one non-$\bot$ term exists in each fully-applied type in the *SNat* family. Hence, these types are called singleton types. In such types, the type variable indexing the type and the one non-$\bot$ term of that type are always isomorphic. Thus, singleton types can be used to force term-level computation and type-level computation to proceed in lock-step.

This singleton is the first runtime argument of the *nth* function and determines the element of the vector that should be returned.

```
nth :: (m :< n) ~ 'True ⇒ SNat m → Vec a n → a
nth SZero       (VCons a _)  = a
nth (SSucc sm') (VCons _ as) = nth sm' as
```

The *nth* code type checks in the second case because pattern matching refines the type variables m and n to be headed by *Succ*. Therefore, the constraint $m :< n$ reduces to the simpler constraint required by the recursive call to *nth*. Furthermore, GHC observes that indexing must succeed. An attempt to add the following case to the pattern match results in a compile-time error.

```
nth m VNil = error "index out of bounds"
```

1.2 The singletons library

Programming with singletons incurs programming overhead in code when compared with programming in a full-spectrum language. As in Figure 1, there are now three sorts of *Nat*s, not just one. Datatype promotion [Yorgey et al. 2012] automatically provides the kind level version, but not the corresponding singleton *SNat*. This datatype must be declared separately, even though its definition is straightforward. Furthermore, this overhead exists not just for datatype definitions, but also for function definitions. For example, the $:<$ type family only applies to types. Two more definitions of $<$ are required to compare term-level *Nat*s and singleton *SNat*s.

Fortunately, this overhead is boilerplate. As previous work has demonstrated, there is a mechanical derivation of singleton types and functions (most notably by Monnier and Haguenauer [2010], but also foreshadowed in earlier work [Bernardy et al. 2010; Crary and Weirich 2000]).

This paper presents the singletons library[1] that supports dependently typed Haskell programming using the singletons design pattern. All of the boilerplate code needed to marry values with types is produced by the Template Haskell primitives provided by the library. These primitives produce singleton types from datatypes, along with type families and singleton functions from function definitions. The singletons library removes the tedium and possibility of errors from this duplication process. Any suitable (see Section 3.2) function over original datatypes can be promoted to the type level and translated to work with singletons.

While the singletons library automates the promotion and refinement of datatypes and functions, it does not hide the nature of the encoding from the programmer. The plumbing remains visible, so to speak. In our experience programming with using the singletons library, we have had to remain aware of the definitions made in the library and how they interact with GHC's type inference engine in order for our code to compile. Nevertheless, the automation provided by the library allowed us to focus our efforts on the harder parts of a project instead of on the boilerplate necessary to declare singleton types.

To focus the discussion of this paper, we present an extended example of the use of the singletons library. In *The Power of Pi*, Oury and Swierstra [2008] describe a dependently typed safe database interface. They claim that the interface, written in Agda, would not be possible in Haskell. However, using a translation involving singletons, along with extensions to Haskell added after Oury and Swierstra's paper, this interface can indeed be written in Haskell.

1.3 Contributions

The contributions of this paper include:

- A discussion and analysis of the current state of dependently typed programming in Haskell, demonstrating how well recent (and not so recent) extensions to GHC can be used for this purpose.
- A library, singletons, to better support dependently typed programming in Haskell. This library uses Template Haskell to automatically generate singleton types, automatically lift functions to the type level, and automatically refine functions with rich types. The generated code is tied together using a uniform, kind-directed interface. In the context of the sophisticated Haskell type system, there are a number of design variations in this generation: we present the library in Section 2, explain its implementation in Section 3, and discuss trade-offs in its the design in Section 4.
- An extended example of a type-safe database interface, written using the singletons library, based on an Agda implementation by Oury and Swierstra [2008] (Section 5).
- Proposals for future additions to GHC to better support dependently typed programming (Section 7).

Colors Three colors will be used throughout this paper to distinguish the origin of Haskell source code: code written as part of the singletons library is blue; code generated through the use of the library is red; and other code is black.

2. Programming with singletons

We begin by describing the important language features and elements of the singletons library that support dependently typed pro-

[1] `cabal install singletons`

```
data family Sing (a :: κ)
class SingI (a :: κ) where
  sing :: Sing a
class SingE (a :: κ) where
  type Demote a :: *
  fromSing :: Sing a → Demote (Any :: κ)
class (SingI a, SingE a) ⇒ SingRep a
instance (SingI a, SingE a) ⇒ SingRep a
data SingInstance (a :: κ) where
  SingInstance :: SingRep a ⇒ SingInstance a
class (t ~ Any) ⇒ SingKind (t :: κ) where
  singInstance :: ∀ (a :: κ). Sing a → SingInstance a
```

Figure 2. The definitions of the singletons library

gramming in GHC. As in the introduction, to maintain overall familiarity, we base our examples in this section on length-indexed vectors. We have already seen that working with length-indexed vectors requires *indexed datatypes*, *type-level computation*, and *singleton types*. Below, we briefly review how GHC (and singletons) supports these features in more detail, before discussing additional ingredients for dependently typed programming. A summary of the singletons library definitions is in Figure 2.

2.1 Indexed datatypes

The definition of the datatype *Vec* requires GADTs because its two data constructors *VNil* and *VCons* do not treat the second type argument (of kind *Nat*) uniformly. In the first case, we know that empty vectors have zero length, so this constructor produces a vector of type *Vec a 'Zero*. Likewise, the *VCons* constructor increments the statically-tracked length.

This non-uniform treatment means that pattern matching increases the knowledge of the type system. For example, a safe *head* function can require that it only be called on non-empty lists.

```
head :: Vec a ('Succ n) → a
head (VCons h _) = h
```

Because the type index to *Vec* is the successor of some number, it is impossible for the *head* function to be called with *VNil*. Furthermore, GHC can detect that the *VNil* case could never occur. When checking such a pattern, the compiler would derive an equality *Vec a ('Succ n)* ~ *Vec a 'Zero*, because the type of the pattern must match the type of the argument. Such an equality can never hold, so the case must be impossible. GHC issues a compile-time error if this case were to be added.

2.2 Type-level computation

The :< type operator is an example of a type-level function. In the *nth* example, we use this function in a constraint—indexing a vector is valid only when the index is in range. We can represent such propositions equally as well using a multiparamter type class. However, there are situations where type-level computation is essential. For example, to append two length-indexed vectors, we need to compute the length of the result, and to do that we need to be able to add.

The singletons library supports the automatic reuse of runtime functions at the type-level, through function promotion. The following Template Haskell [Sheard and Peyton Jones 2002] splice not only defines *plus* as a normal function,

```
$(promote [d|
  plus :: Nat → Nat → Nat
  plus Zero     m = m
  plus (Succ n) m = Succ (plus n m) |])
```

but also *generates* the following new definition (note that GHC requires the type family name to be capitalized):

```
type family     Plus (n :: Nat) (m :: Nat) :: Nat
type instance Plus 'Zero m = m
type instance Plus ('Succ n) m = 'Succ (Plus n m)
```

Above, the $(...) syntax is a Template Haskell splice—the contents of the splice are evaluated at compile time. The result is a list of Template Haskell declarations that is inserted into Haskell's abstract syntax tree at the splice point. This code also demonstrates Template Haskell's quoting syntax for declarations, [*d*| ... |]. Any top-level declarations can appear in such a quote.

We can use *Plus* to describe the type of append:

```
append :: Vec a n → Vec a m → Vec a (Plus n m)
append VNil       v2 = v2
append (VCons h t) v2 = VCons h (append t v2)
```

The length of the combined vector is simply the sum of the lengths of the component vectors.

2.3 Singleton datatypes

The singletons library also automates the definition of singleton types. The *genSingletons* function generates the singleton definitions from datatypes already defined. For example, the following line generates singleton types for *Bool* and *Nat*:

```
$(genSingletons [''Bool, ''Nat])
```

The double-quote syntax tells Template Haskell that what follows is the name of a type. (A single quote followed by a capitalized identifier would indicate a data constructor.)

The singletons library uses a kind-indexed data family, named *Sing*, to provide a common name for all singleton types.

```
data family Sing (a :: κ)
```

A *data family* is a family of datatype definitions. Each instance in the family has its own set of data constructors, but the family shares one type constructor. The applicable data constructors for a particular datatype are determined by the parameters to the data family. *Kind-indexed* type and data families are a new addition to GHC, introduced with datatype promotion [Yorgey et al. 2012]. A kind-indexed type family can branch on the kind of its argument, not just the type, and the constructors of a kind-indexed data family are determined by the kind arguments as well as the type arguments to the data constructor.

The call to *genSingletons* above generates the following declarations (see below for a description of the class *SingRep*):

```
data instance Sing (a :: Bool) where
  STrue :: Sing 'True
  SFalse :: Sing 'False
data instance Sing (a :: Nat) where
  SZero :: Sing 'Zero
  SSucc :: SingRep n ⇒ Sing n → Sing ('Succ n)
```

Each constructor in an unrefined datatype produces a constructor in the singleton type. The new constructor's name is the original constructor's name, prepended with *S*.[2]

The singletons library also produces synonyms to *Sing* to enforce the kind of a type argument. These synonyms are just the original datatype names prepended with an *S*:

[2] Symbolic names (operators) are prepended with :%. It is possible that the new names introduced here and elsewhere will clash with other names. At present, it is the library user's responsibility to avoid such name clashes.

```
type SNat  (a :: Nat)  = Sing a
type SBool (a :: Bool) = Sing a
```

These synonyms can be used in type signatures when the kind of the type parameter is known. Using the synonym instead of *Sing* adds documentation and kind-checking at little cost.

2.4 Singleton functions

Functions can also have singleton types. In order to generate singleton versions of functions, they must be defined within a splice.

```
$(singletons [d|
  isEven :: Nat → Bool
  isEven Zero          = True
  isEven (Succ Zero)   = False
  isEven (Succ (Succ n)) = isEven n

  nextEven :: Nat → Nat
  nextEven n = if isEven n then n else Succ n |])
```

This code generates not just the promoted version of the function (as a type family) but also a runtime version of the function that works with singleton types. The name of the new function is the original function's name, prepended with an *s* and with the next letter capitalized.[3] Note the use of *sTrue* instead of *STrue* in the code below. These are smart constructors for the singleton types, described in Section 3.1. The *sIf* construct, provided by the singletons library, branches on *SBool*s.

```
sIsEven :: Sing n → Sing (IsEven n)
sIsEven SZero          = sTrue
sIsEven (SSucc SZero)  = sFalse
sIsEven (SSucc (SSucc n)) = sIsEven n

sNextEven :: Sing n → Sing (NextEven n)
sNextEven n = sIf (sIsEven n) n (sSucc n)
```

With these definitions, we can write a function to extend a vector until it has an even length, duplicating the first element if necessary:

```
makeEven :: SNat n → Vec a n → Vec a (NextEven n)
makeEven n vec = case sIsEven n of
  STrue  → vec
  SFalse → case vec of
    VCons h t → VCons h (VCons h t)
```

To make this code type check, we must use the function *sIsEven*. Pattern matching on the result of *sIsEven* brings information about *n* into the context so that the cases in the pattern match have the expected type.

Along with *genSingletons* and *singletons*, the singletons library provides *genPromotions* and *promote*, which convert term-level declarations into type-level declarations only. Generating singletons requires promoting first, so most users will use only the *genSingletons* and *singletons* functions. See Section 3.3 for more details on singleton conversion for functions.

2.5 Forgetting static information

The overloaded operation *fromSing* eliminates a singleton term and gives back a term of the unrefined datatype. It witnesses one direction of the isomorphism between the members of a singleton type family and the unrefined version of the type. For example, suppose we have a function with the following type signature that takes some number of elements from a vector and forms a list with those elements:

```
vtake :: Nat → Vec a n → [a]
```

To call *vtake* with a value of type *SNat n*, we need to convert to a plain old *Nat*. The function *fromSing* does that.

```
vtake' :: (m :<=: n) ~ 'True ⇒ SNat m → Vec a n → [a]
vtake' m vec = vtake (fromSing m) vec
```

The *fromSing* function is defined in the class *SingE*, repeated here for convenience:

```
class SingE (a :: κ) where
  type Demote a :: *
  fromSing :: Sing a → Demote (Any :: κ)
```

The *Demote* associated kind-indexed type family returns the type from which a kind was promoted. The most interesting aspect of the instances is the definition of *Demote*, which is a little subtle in the case of a parameterized type. As examples, here are the instances for *Nat* and *Maybe*:

```
instance SingE (a :: Nat) where
  type Demote a = Nat
  fromSing SZero     = Zero
  fromSing (SSucc n) = Succ (fromSing n)
instance SingE (a :: Maybe κ) where
  type Demote a = Maybe (Demote (Any :: κ))
  fromSing SNothing = Nothing
  fromSing (SJust a)  = Just (fromSing a)
```

Ideally, we would write *Demote* with only an explicit kind parameter. However, this feature is not yet supported in GHC. Instead, *Demote* takes a type parameter *a* and its kind κ, and it branches only on its kind parameter κ. To write the instance for *Maybe*, we need to supply the recursive call to *Demote* with some type of kind κ. We use the *Any* type, which is a primitive provided by GHC that is an inhabitant of every kind.[4] In the case of *Demote*, it provides an exact solution to our problem: we simply use *Any* with an explicit kind signature to get the recursive *Demote* call to work as desired. Because recursive calls to *Demote* must use *Any*, it is also necessary to use *Any* in the type signature for *fromSing*; otherwise the type checker tries to unify *Demote* (*a* :: κ) with *Demote* (*Any* :: κ). Using the knowledge that the type parameter is irrelevant, we can see that these two types clearly unify, but the compiler does not have that specialized knowledge and issues an error.

2.6 Implicit arguments

Sometimes, runtime singleton arguments can be determined by compile-time type inference. For example, here is a function that creates a vector containing some repeated value:

```
replicate₁ :: SNat n → a → Vec a n
replicate₁ SZero     _ = VNil
replicate₁ (SSucc n) a = VCons a (replicate₁ n a)
```

However, the compiler can often use type inference to calculate the value of *SNat n* that is required in a call to this function. For example, when we know (from unification) that a vector of length two is required, then the only possible argument to this function is *SSucc* (*SSucc SZero*). Therefore, the compiler should be able to infer this argument and supply it automatically.

The singletons library supports such implicit arguments using the *SingI* type class.

```
class SingI (a :: κ) where
  sing :: Sing a
```

This class merely contains the singleton value in its dictionary, which is available at runtime. (Because of the *Sing* data family,

[3] Symbolic function names are prepended with %:

[4] *Any* is an analogue of ⊥ at the type level.

note that we need only have one class that contains many different types of singleton values.)

To enable GHC to implicitly provide the singleton argument to *replicate*, we rewrite it as follows:

```
replicate2 :: ∀ n a. SingI n ⇒ a → Vec a n
replicate2 a = case (sing :: Sing n) of
  SZero    → VNil
  SSucc _ → VCons a (replicate2 a)
```

In the first version, the *SNat* parameter is passed explicitly and is used in a straightforward pattern match. In the second version, the *SNat* parameter is passed implicitly via a dictionary for the *SingI* type class. Because a pattern match is still necessary, we have to produce the singleton term using the method *sing*. In the recursive call to $replicate_2$, we need an implicit *SNat*, not an explicit one. This implicit parameter is satisfied by the class constraint on the *SSucc* constructor.

Instances of the *SingI* class are automatically generated along with the singleton type definitions. For example,

```
$(genSingletons [''Nat])
```

generates the following instance declarations:

```
instance SingI 'Zero where
  sing = SZero
instance SingRep n ⇒ SingI ('Succ n) where
  sing = SSucc sing
```

The* SingRep *class The *SingI* and *SingE* classes are kept separate because while it is possible to define instances for *SingE* on a datatype-by-datatype basis, the instances for *SingI* must be defined per constructor. However, it is often convenient to combine these two classes. The *SingRep* class is a essentially a synonym for the combination of *SingI* and *SingE*.[5] As such, it is unnecessary for the singletons library to generate instances for it. All parameters to singleton type constructors have a *SingRep* constraint, allowing a programmer to use *sing* and *fromSing* after pattern matching with these constructors.

```
class (SingI a, SingE a) ⇒ SingRep a
instance (SingI a, SingE a) ⇒ SingRep a
```

2.7 Explicitly providing implicit arguments

What if we are in a context where we have a *value* of type *SNat n* but no *dictionary* for *SingI n*? Nevertheless, we would still like to call the $replicate_2$ function. What can we do?

On the surface, it might seem that we could simply call $replicate_2$ without fuss; after all, the compiler can ascertain that *n* is of kind *Nat* and any type of kind *Nat* has an associated instance of the class *SingI*. There are two fallacies in this line of reasoning. First, the dictionary for *SingI n* must be available at runtime, and the value of *n*—a type—is erased at compile time. Second, the compiler does not perform the induction necessary to be sure that every type of kind *Nat* has an instance of *SingI*. If Haskell permitted programmers to supply dictionaries explicitly, that construct could solve this problem. This idea is explored further in Section 7.

The solution in the current version of Haskell is the *SingKind* class, which is defined over a kind and can provide the necessary instance of *SingI*. The intuition is that *SingKind* is the class of kinds that have singletons associated with them. Ideally, the definition of *SingKind* would start with

```
class SingKind (κ :: □) where ...
```

where □ is the sort of kinds and informs the compiler that κ is a kind variable, not a type variable. At present, such a definition is not valid Haskell. Instead, we use this definition as a workaround:

```
class (t ∼ Any) ⇒ SingKind (t :: κ) where
  singInstance :: ∀ (a :: κ). Sing a → SingInstance a
```

In this definition, *Any* is once again used to pin down the value of the *t* type variable, indicating that only κ matters. The *singInstance* method returns a term of type *SingInstance*, which stores the dictionaries for *SingI* and *SingE*.

```
data SingInstance (a :: κ) where
  SingInstance :: SingRep a ⇒ SingInstance a
```

Here is the generated instance of *SingKind* for *Nat*:

```
instance SingKind (Any :: Nat) where
  singInstance SZero     = SingInstance
  singInstance (SSucc _) = SingInstance
```

For example, using *SingKind*, the programmer can satisfy the *SingI n* constraint for $replicate_2$ as follows:

```
mkTrueList :: SNat n → Vec Bool n
mkTrueList n = case singInstance n of
  SingInstance → replicate2 True
```

3. Implementing the singletons library

In this section we go into more detail about the automatic generation of singleton types and functions, as well as the promotion of term-level functions to the type level.

3.1 Generating singleton types

Given a promotable[6] datatype definition of the form

```
data T a1 ... an = K t1 ... tm
```

the Template Haskell splice $(genSingletons [''T]) produces the the following instance of *Sing*:

```
data instance Sing (x :: T a1 ... an) where
  SK :: ∀ (b1 :: t1) ... (bm :: tm).
    (SingKind (Any :: a_{i1}), ..., SingKind (Any :: a_{ip}),
      SingRep b1, ..., SingRep bm) ⇒
    Sing b1 → ... → Sing bm → Sing ('K b1 ... bm)
```

where $i_1, ..., i_p$ are the indices of the kind variables *a* that appear outside of any kind constructor in any of the $t_1, ..., t_m$.

The type of the singleton data constructor *SK* is created by translating each parameter of the original data constructor *K*. Any such parameter t_i that is not an arrow type is converted to an application of the *Sing* data family to a fresh type variable b_i of kind t_i, having been promoted. (We discuss the special treatment for arrow types below, in Section 3.3.) The result type of the data constructor is the *Sing* data family applied to the promoted data constructor applied to all of the generated type variables.

To allow for the use of functions with implicit parameters, the the type also includes a *SingRep* constraint generated for each parameter b_i. (Section 4.3 discusses this design decision.) If the original datatype *T* is a parameterized datatype, then it is also necessary to add a *SingKind* constraint for any parameters used in the arguments of the data constructor. Those that do not occur do not require such a constraint, as explained shortly.

For example, the declaration generated for the *Maybe* type is:

[5] With the new ConstraintKinds extension, it is possible to make a true synonym, using **type**, for a pair of class constraints. However, pairs of constraints are not yet compatible with Template Haskell, so we are unable to use this simplification.

[6] See Yorgey et al. [2012] for an explanation of what types are promotable.

```
data instance Sing (b :: Maybe κ) where
  SNothing :: Sing 'Nothing
  SJust    :: ∀ (a :: κ). (SingKind (Any :: κ), SingRep a) ⇒
              Sing a → Sing ('Just a)
```

Note that this definition includes two class constraints for *SJust*. The first constraint *SingKind* (*Any* :: κ) is necessary for *Maybe*'s instance for *SingKind*. In the *SJust* case below, we need to know that the kind κ has an associated singleton.

```
instance SingKind (Any :: Maybe κ) where
  singInstance SNothing = SingInstance
  singInstance (SJust _) = SingInstance
```

Kind parameters mentioned only within the context of a kind constructor need not have the explicit *SingKind* constraint. Consider a kind parameter κ that appears only in the kind *Maybe* κ. There will be a type parameter (*b* :: *Maybe* κ) and a term parameter *Sing b*. Any code that eventually extracts a value of type *Sing* (*t* :: κ) for some type *t* of kind κ would have to do so by pattern-matching on a constructor of *Sing* (*b* :: *Maybe* κ) — in other words, the *SJust* constructor. This pattern match would include the *SingKind* constraint written as part of the *SJust* constructor, so we can be sure that the kind κ has an associated singleton type, as desired. Thus, including the *SingKind* constraint on κ in the translation above would be redundant.

The second constraint, *SingRep a*, ensures that the singleton type for *a* is available implicitly and that this singleton can be demoted to a raw type using *fromSing*.

Smart constructors The definition of the *SJust* constructor above requires a caller to satisfy two constraints: one for *SingKind* and one for *SingRep*. However, a dictionary for *SingKind* can always produce one for *SingRep*, so this is redundant. Listing both constraints improves usability when pattern-matching, as both instances are brought into the context—a programmer does not have to use the *singInstance* method to get to the *SingRep* instance. To ease the constraint burden when using a singleton constructor, smart constructors are generated without the *SingRep* constraints:

```
sSucc :: Sing n → Sing ('Succ n)
sSucc n = case singInstance n of SingInstance → SSucc n
```

3.2 Promoting functions to the type level

The current version of GHC automatically promotes suitable datatypes to become data kinds. However, there is no promotion for functions. Instead, the *promote* and *singletons* functions of the singletons library generate type families from function definitions with explicit type signatures. The explicit type signatures are necessary because the GHC does not currently infer the parameter kinds or result kind of a type family (they default to $*$) and because Template Haskell splices are processed before type inference.

Although the syntax of term-level functions and type-level functions is quite different, these differences are not substantial. Term-level constructs, such as conditionals and case statements, can be accurately simulated by defining extra type-level functions to perform pattern matching. Figure 3 summarizes the current state of the implementation, showing what Haskell constructs are supported and what are not. The constructs in the second column are those that we believe are relatively straightforward to add.[7]

The constructs in the last column are those that would require a change in Haskell to implement fully. First, Iavor Diatchki is currently working to promote certain literals (currently, strings and natural numbers) to the type level and produce singleton types for them.[8] The interface to this extension will agree with the singletons interface. Next, all uses of a type family must be fully saturated. While it is conceivable to simulate a closure at the type level with the current features of Haskell, the closure would have to be immediately applied to arguments to satisfy the saturation constraint. Such a restriction makes programming with λ-expressions impractical, so they are not supported. See Section 7 for more discussion of this topic. Finally, **do** expressions and arithmetic sequences both rely on type classes, which are currently not promotable.

Implemented	Not yet	Problematic
variables tuples constructors infix expressions ! / ~ / _ patterns aliased patterns lists $(\cdot)$ sections $(x\,\cdot)$ sections *undefined* deriving *Eq*	unboxed tuples records scoped type variables overlapping patterns pattern guards $(\cdot\,x)$ sections **case** **let** list comprehensions	literals λ-expressions **do** arithmetic sequences

Figure 3. The current state of function promotion

3.3 Converting functions to work with singletons

As discussed in Section 2.4, any term function defined in the context of a call to the *singletons* function is not only promoted to the type level, but also redefined to work with singleton-typed arguments. For example, the *isEven* function generates both the type family *IsEven* and the term function *sIsEven* :: *Sing b* → *Sing* (*IsEven b*). As discussed above, the function definition must be explicitly annotated with its type for this translation to succeed.

The translation algorithm uses the explicit type of the function to derive the desired kind of the generated type family declaration and the type of the singleton version of the function. For example, consider the function *fmap*, specialized to *Maybe*. The splice (ignoring the body of the *fmap* function)

```
$(singletons [d|
  fmap :: (a → b) → Maybe a → Maybe b |])
```

produces the declarations:

```
type family Fmap (f :: κ1 → κ2) (m :: Maybe κ1)
  :: Maybe κ2
sFmap :: ∀ (f :: κ1 → κ2) (m :: Maybe κ1).
  SingKind (Any :: κ2) ⇒
  (∀ (b :: κ1). Sing b → Sing (f b)) →
  Sing m → Sing (Fmap f m)
```

The kind of the type family is the promoted type of the function. Because the *fmap* is polymorphic in two type variables, *Fmap* is polymorphic in two kind variables κ_1 and κ_2.

However, the type of *sFmap* is both kind- and type-polymorphic. The original function takes two arguments, of type $(a \to b)$ and *Maybe a*, so the refined version also takes two arguments. All non-arrow types, such as *Maybe a*, are converted to an application of *Sing* to a fresh type variable, such as *Sing m*. Arrows, such as $a \to b$, on the other hand, are converted to polymorphic function types over singletons.

This type translation is actually a well-known extension of singleton types to arrow kinds [Crary and Weirich 2000; Guillemette

[7] Promoting **let** and **case** would require generating a new type family to perform the pattern match. A user would then need to specify the argument type and return type of these statements for them to be promoted.

[8] See `http://hackage.haskell.org/trac/ghc/wiki/TypeNats`

and Monnier 2008a]. In general, for f of kind $\kappa_1 \to \kappa_2$, the singleton type can be expressed by the following kind-indexed function (writing the kind indices explicitly).

$$\begin{aligned} &\mathit{Singleton}[\![\kappa_1 \to \kappa_2]\!] f = \\ &\quad \forall(\alpha : \kappa_1).\, \mathit{Singleton}[\![\kappa_1]\!]\alpha \to \mathit{Singleton}[\![\kappa_2]\!](f\,\alpha) \\ &\mathit{Singleton}[\![\kappa]\!]\tau = \mathit{Sing}\ \tau \qquad \text{when } \kappa \text{ is a base kind, like } \mathit{Nat} \end{aligned}$$

The intuition is that a higher-order function on singletons can take only functions that operate on singletons as parameters. Any such function must necessarily be polymorphic in its singleton type parameters. Thus, arrow types get translated to higher-rank polymorphic functions as above. However, because type-level functions must be fully saturated in Haskell (see Section 7), there is currently a limited use for such parameters. The type f can be instantiated with only type and promoted data constructors.

In the type of *sFmap*, the *SingKind* constraint is necessary because this function must process a singleton value whose type index is of kind κ_2. In other words, the singleton type associated with κ_2 appears in a negative context in the type of *sFmap*. Every singleton type that appears in a negative context must have a kind that has an associated singleton because terms of these types are passed to the smart constructors defined for other singleton types. This fact is trivially true whenever the outer-most kind constructor is fixed (such as in *Nat* or *Maybe* κ), but it must be explicitly declared when the kind is fully polymorphic, as here. The singletons library detects polymorphic kinds whose singleton types are in a negative context and adds the necessary *SingKind* constraints.

The translation of the bodies of the functions is straightforward. Locally bound variables are translated to the variables generated during translating variable patterns, other variables are translated into the singleton versions of the variables of the same name (*fmap* to *sFmap*, for example), conditional statements are translated to use an *sIf* function defined to work with singleton *Bool* values, and constructors are translated to the associated smart constructor. For example, here is the body of the translated *fmap* function:

```
sFmap _ SNothing = SNothing
sFmap f (SJust a) = sJust (f a)
```

4. Design decisions

In designing the conversion algorithms, support classes and definitions, there were a number of design choices. This section presents some of those alternatives.

4.1 Singletons for parameterized types

The singletons library uses a novel translation for parameterized datatypes, diverging from prior work. A more standard translation would include an extra parameter for each parameter of the original datatype. For example, under this scheme we would generate the following singleton type for *Maybe*:

```
data SMaybe (s :: κ → *) (m :: Maybe κ) where
  SNothing :: SMaybe s 'Nothing
  SJust    :: ∀ (a :: κ). (SingKind (Any :: κ), SingRep a) ⇒
              s a → SMaybe s ('Just a)
```

This type definition is polymorphic over κ, the kind parameter to *Maybe*. Therefore, to construct the singleton type for a, of kind κ, this *SMaybe* type is also parameterized by s. With this definition, we might create a singleton value for the type (*'Just* (*'Succ 'Zero*)) of kind *Maybe Nat*, as below:

```
sJustOne :: SMaybe SNat ('Just ('Succ 'Zero))
sJustOne = SJust (SSucc SZero)
```

This parameter s is awkward. The definition *SMaybe* is not as flexible as it appears. For every kind, there is only one singleton type associated with that kind, so once κ has been instantiated, there is only one possibility for s.

Fortunately, the presence of type and data families in GHC improves the translation. Compare this definition to that in Section 3.1. The key difference is the use of *Sing* to select the singleton type for the parameter a, using its kind κ, even though the kind is abstract. That both simplifies and unifies the definition of singletons for parameterized datatypes.

4.2 The Sing kind-indexed data family

The implementation presented in this paper uses a kind-indexed data family for *Sing*. One may ask why it is better to make *Sing* a data family instead of a type family. The problem with a *Sing* type family involves type inference. Consider the following type signature, using the *SingTF* type family instead of the *Sing* data family:

```
sPlus :: SingTF n → SingTF m → SingTF (Plus n m)
```

If a programmer tries to write *sPlus* (*SSucc SZero*) *SZero*, the compiler has no way to infer the values of n and m, as those type variables are used only in the context of type family applications. On the other hand, because all data families are necessarily injective, the *Sing* data family does not hinder type inference.

The drawback of using a data family is that we cannot define an instance for arrow kinds. As described previously, the singleton type for a type of kind $\kappa_1 \to \kappa_2$ should be the polymorphic function type $\forall\,(b :: \kappa_1).\ \mathit{Sing}\ b \to \mathit{Sing}\ (a\ b)$, not a new datatype. Therefore, there is no data instance of *Sing* for types of these kinds. As a result, we cannot create a singleton for the type *Maybe* (*Nat* → *Nat*), for example. However, given the saturation requirement for type functions, such types would be of limited utility. Alternatively, using a type family for *Sing* almost allows this definition, but currently fails because the result of a type family currently cannot be a quantified type. Type families also suffer the limitations of type inference discussed above.

4.3 Implicit vs. explicit parameters

When defining a function in Haskell, a programmer has a choice of making a singleton parameter implicit (using a class constraint *SingI*) or explicit (using *Sing* directly).[9] This choice makes a difference. An explicit parameter aids in type inference. A function f of type $(\mathit{SingI}\ a) \Rightarrow \mathit{If}\ a\ \mathit{Nat}\ \mathit{Bool} \to \mathit{Bool}$ cannot be called because there is no way to determine a through unification and thus derive the implicit argument. In general, if a appears only as the arguments to type functions, then an explicit *Sing* a parameter should be used. Alternatively, when arguments can be inferred, it is simpler to make them implicit. Furthermore, it is easy to convert an implicit parameter to an explicit one (simply by using *sing*), whereas the converse is not possible, in general.[10] The *singInstance* method provides access to a *SingI* constraint, but itself requires a *SingKind* constraint.

In the singletons library, we have made sure that implicit parameters are always available. Each singleton data constructor includes a *SingRep* constraint for each of its arguments. This creates some redundancy, as a data constructor such as *SSucc* includes both implicit and explicit arguments. Alternatively, we could have defined *SNat* so that *SSucc* includes *only* the implicit arguments:

[9] Lewis et al. have described an extension ImplicitParams to Haskell that enables implicit parameters [Lewis et al. 2000]. The discussion here is about class constraints considered as implicit parameters, and is not directly related.

[10] One way this symmetry could be remedied is by introducing the notion of local instances into Haskell. This concept is discussed further in Section 7.

```
data instance Sing (a :: Nat) where
  SZero :: Sing 'Zero
  SSucc :: SingRep n ⇒ Sing ('Succ n)
```

Because it is easy to convert implicit parameters to explicit ones, storing only the implicit parameters would work for all applications. However, extracting an explicit form of an implicit argument would require an explicit type signature (as seen above in $replicate_2$) and would then require heavy use of ScopedTypeVariables. Though this system is free from implicit/explicit redundancy, the ease of use is impacted.

5. Example: A safe database interface

Oury and Swierstra [2008] present a dependently typed database interface supporting an expressive and strongly typed relational algebra, written in Agda. In their discussion of the system, the authors note various constructs that Agda supports that are not present in Haskell. Here, we present a translation of Oury and Swierstra's example using singletons. Notwithstanding the caveat that Haskell admits $\bot$, this translation preserves the typing guarantees of the original and retains its simplicity. The complete code for the example is online.[11]

The goal is to write a strongly typed interface to a database, similar to HaskellDB [Bringert et al. 2004; Leijen and Meijer 1999]. That work uses phantom types provided in the client code to control the types of values returned from database queries. The interface is responsible for making sure that the data in the database is compatible with the types provided.

The use of singletons in our version of the library removes a significant amount of boilerplate code present in the HaskellDB version. However, note that this section is not a proposal for a new, production-ready database interface. Instead, this example is included to contrast with dependently typed code written in Agda.

5.1 Universe of types

The interface must define a set of types used in the database. Following Oury and Swierstra, we define the universe type *U* and type-level function *El* as follows:

```
$(singletons [d|
  data U = BOOL | STRING | NAT | VEC U Nat
    deriving (Eq, Read, Show) |])
type family    El (u :: U) :: *
type instance El BOOL = Bool
  -- other instances
type instance El (VEC u n) = Vec (El u) n
```

The *El* type-level function connects a constructor for the *U* datatype to the Haskell type of the database element.

Magalhães [2012] also treats the topic of universes in the context of GHC's recent capabilities; we refer the reader to his paper for a more complete consideration of this encoding using promoted datatypes.

5.2 Attributes, schemas, and tables

A database is composed of a set of tables, where each table is a set of rows of a particular format. The format of a row (the columns of a table) is called that row's schema. In this interface, a schema is an ordered list of attributes, where each attribute has an associated name and type. The type of an attribute is denoted with an element of *U*. Ideally, the name of an attribute would be a *String*; however, type-level *Strings* are not yet available.[12] Nevertheless, the code in this paper uses strings at the type level; please refer to the version online to see the desugared version. Here are the relevant definitions:

```
$(singletons [d|
  data Attribute = Attr String U
  data Schema    = Sch [Attribute] |])
```

Note that we define the schema using the *singletons* function to generate its associated singleton type.

We next define rows in our database:

```
data Row :: Schema → * where
  EmptyRow :: [Int] → Row (Sch '[])
  ConsRow  :: El u → Row (Sch s) →
              Row (Sch ((Attr name u) ': s))
```

The *Row* datatype has two constructors. *EmptyRow* takes a list of *Int*s to be used as the unique identifier for the row. (A list is necessary because of the possibility of Cartesian products among rows.) *ConsRow* takes two parameters: the first element and the rest of the row. The types ensure that the schema indexing the row has the right attribute for the added element.

A table is just a list of rows, all sharing a schema:

```
type Table s = [Row s]
```

For a very simple database containing student information, we could define the following schema

```
$(singletons [d|
  gradingSchema =
    Sch [Attr "last"  STRING,
         Attr "first" STRING,
         Attr "year"  NAT,
         Attr "grade" NAT,
         Attr "major" BOOL] |])
```

and table (using (▷) for *ConsRow*)

```
["Weirich"   ▷ "S" ▷ 12 ▷ 3 ▷ False ▷ EmptyRow [0],
 "Eisenberg" ▷ "R" ▷ 10 ▷ 4 ▷ True  ▷ EmptyRow [1]]
  :: Table GradingSchema
```

The explicit type signature is necessary because it is impossible to infer the full details of the schema from this definition.

5.3 Interacting with the database

Client code interacts with the database through an element of the type *Handle*:

```
data Handle :: Schema → * where ...
```

We have elided the constructors for *Handle* here, as we are not interested in the concrete implementation of the database access.

Client code connects to the database through the *connect* function, with the following type signature:

```
connect :: String → SSchema s → IO (Handle s)
```

This is the first use in this example of a singleton type. Recall that the name *SSchema* is a synonym for the *Sing* kind-indexed data family, where the kind of its argument is *Schema*. The first argument to *connect* is the database name, the second is the expected schema, and the return value (in the *IO* monad) is the handle to the database. The *connect* function accesses the database, checks that the provided schema matches the expected schema, and, on success, returns the handle to the database. Internally, the check against

[11] `http://www.cis.upenn.edu/~eir/papers/2012/singletons/code.tar.gz`, requires GHC version ⩾ 7.5.20120529.

[12] Type-level *Symbol*s (essentially, atomic *String*s) are forthcoming in Diatchki's TypeNats library.

the database's schema is performed using the *fromSing* function—the schema loaded from the database will not be a singleton, so it is necessary to convert the singleton *SSchema s* to a *Schema* to perform the comparison.

In our strongly typed interface, this function is the one place where a runtime type error can error, which happens when the expected schema and actual schema do not match. In this case, this function throws an error in the *IO* monad. Outside of this one function, a client is guaranteed not to encounter type errors as it reads from the database.

5.4 Relational algebra

Following Oury and Swierstra, we define the type *RA* (short for relational algebra) as follows:

```
data RA :: Schema → ∗ where
  Read    :: Handle s → RA s
  Union   :: RA s → RA s → RA s
  Diff    :: RA s → RA s → RA s
  Product :: (Disjoint s s' ∼ 'True, SingI s, SingI s') ⇒
             RA s → RA s' → RA (Append s s')
  Project :: (SubsetC s' s, SingI s) ⇒
             SSchema s' → RA s → RA s'
  Select  :: Expr s BOOL → RA s → RA s
```

The *RA* type is itself indexed by the schema over which it is valid. The constructors represent different algebraic operations a client might wish to perform. The first three allow a client to consider all rows associated with a table, take the union of two sets of rows, and take the difference between two sets of rows. The *Product* constructor represents a Cartesian product, requiring that the two schemas being combined are disjoint. The *Project* constructor projects a set of columns (possibly reordered) from a larger set; it requires the resulting schema to be a subset of the original schema. The *Select* constructor uses a Boolean expression to select certain rows. The *Expr* GADT (elided) uses its type parameter to constrain of the result value of an expression.

The *Product* and *Project* constructors deserve attention, as they each exhibit special features of our use of singleton types.

The* Product *constructor The constraint on the *Product* constructor uses the *Disjoint* type family to ensure that the two schemas do not share attributes. The code below is written at the term level and defined over the simple datatypes described above. It uses common features, such as wildcard patterns and infix operators. These functions are promoted to the type families *Append*, *AttrNotIn* and *Disjoint* by *singletons*.

```
$(singletons [d|
    -- append two schemas
  append :: Schema → Schema → Schema
  append (Sch s1) (Sch s2) = Sch (s1 ++ s2)
    -- check that a schema is free of a certain attribute
  attrNotIn :: Attribute → Schema → Bool
  attrNotIn _ (Sch []) = True
  attrNotIn (Attr name u) (Sch ((Attr name' _) : t)) =
    (name ≢ name') ∧ (attrNotIn (Attr name u) (Sch t))
    -- check that two schemas are disjoint
  disjoint :: Schema → Schema → Bool
  disjoint (Sch []) _ = True
  disjoint (Sch (h : t)) s =
    (attrNotIn h s) ∧ (disjoint (Sch t) s) |])
```

The* Project *constructor The *Project* constructor encodes the type constraints using type classes and GADTs instead of a type family. (We compare the two approaches in Section 6.) It uses the GADT *SubsetProof*, shown below, to encode the *relation* that one schema is contained within another. This relation relies on *InProof*, which holds when an attribute occurs in a schema. The classes *InC* and *SubsetC* make these proofs inferrable by the Haskell type-checker, so they need not be provided with each use of *Project*. Note that there is one instance of each class per constructor of the associated datatype.

```
data InProof :: Attribute → Schema → ∗ where
  InElt  :: InProof attr (Sch (attr ': schTail))
  InTail :: InC name u (Sch attrs) ⇒
    InProof (Attr name u) (Sch (a ': attrs))
class InC (name :: String) (u :: U) (sch :: Schema) where
  inProof :: InProof (Attr name u) sch
instance InC name u (Sch ((Attr name u) ':
                          schTail)) where
  inProof = InElt
instance InC name u (Sch attrs) ⇒
  InC name u (Sch (a ': attrs)) where
  inProof = InTail

data SubsetProof :: Schema → Schema → ∗ where
  SubsetEmpty :: SubsetProof (Sch '[]) s'
  SubsetCons  ::
    (InC name u s', SubsetC (Sch attrs) s') ⇒
      SubsetProof (Sch ((Attr name u) ': attrs)) s'
class SubsetC (s :: Schema) (s' :: Schema) where
  subset :: SubsetProof s s'
instance SubsetC (Sch '[]) s' where
  subset = SubsetEmpty
instance (InC name u s', SubsetC (Sch attrs) s') ⇒
         SubsetC (Sch ((Attr name u) ': attrs)) s' where
  subset = SubsetCons
```

Automatic inference of these classes requires the OverlappingInstances extension. In general, both instances of *InC* are applicable for matching attributes, keeping in mind that GHC's search for an instance examines only the instance head, not the constraints. However, the first instance above is always more specific than the second, meaning it would take precedence. This preference for the first instance gives the expected behavior—matching at the first occurrence—in the event that two attributes in a schema share a name. The alternative to OverlappingInstances is to require client code to build the proof terms of type *InProof* and *SubsetProof* explicitly.

5.5 Queries

Client code can retrieve rows from the database using the *query* function with the following signature:

```
query :: ∀ s. SingI s ⇒ RA s → IO [Row s]
```

For example, the following code lists the names of all students earning a grade less than 90, by first selecting the students matching the criteria (using constructors of the *Expr* GADT) and projecting their first and last names.

```
$(singletons [d|
  names = Sch [Attr "first" STRING,
               Attr "last"  STRING] |])

main :: IO ()
main = do
  h ← connect "data/grades" sGradingSchema
  notAStudents ←
    query $ Project sNames $
```

```
    Select (LessThan (Element (sing :: Sing "grade"))
                     (LiteralNat 90)) (Read h)
  putStrLn (show notAStudents)
```

Note that this code uses *sGradingSchema*, the singleton value that corresponds to *gradingSchema*, defined above. To construct the singleton string value we assume that *String* is in the *SingI* class.[13]

The most illustrative part of the implementation of *query* is the handling of the *Project* constructor. We highlight a function that extracts an element from a row:

```
extractElt :: ∀ nm u sch. InC nm u sch ⇒
  Sing (Attr nm u) → Row sch → El u
extractElt attr r =
  case inProof :: InProof (Attr nm u) sch of
    InElt → case r of
      ConsRow h t → h
      -- EmptyRow _ → IMPOSSIBLE
      _ → bugInGHC
    InTail → case r of
      ConsRow h t → extractElt attr t
      -- EmptyRow _ → IMPOSSIBLE
      _ → bugInGHC
```

To extract the value of a certain element from a row, we must know where that element appears. We could do a straightforward search for the element, but in general, that search may fail. Instead, we pattern-match on the proof, produced from *InC*, that the desired element is in the schema that indexes the row. (To extract this proof, using *inProof*, we need an explicit type signature.)

If the proof that an attribute with name *name* is in the schema *s* is witnessed by *InElt*, we return the first element in the row. (The *InElt* constructor indicates that the first element in the row is the one that matches the desired name.) Thus, we pattern-match on *row* to extract its head element. This head element must be present—the case that *row* is *EmptyRow* can be statically shown impossible by *InElt*.[14] If the *InProof* is witnessed by *InTail*, we recur on the tail of *row*, which also must exist.

5.6 Comparison with Agda

The translation of this example from Agda into Haskell shows the expressiveness of dependently typed programming in Haskell with singletons. In particular, compare the following Agda definitions, taken verbatim from Oury and Swierstra's work (the curly braces { } in the code below indicate implicit arguments and the _ ! _ notation declares an infix operator):

```
data RA : Schema → Set where
  Read    : ∀ {s} → Handle s → RA s
  Union   : ∀ {s} → RA s → RA s → RA s
  Diff    : ∀ {s} → RA s → RA s → RA s
  Product : ∀ {s s'} → {So (disjoint s s')}
    → RA s → RA s' → RA (append s s')
  Project : ∀ {s} → (s' : Schema)
    → {So (sub s' s)} → RA s → RA s'
  Select  : ∀ {s} → Expr s BOOL
    → RA s → RA s
```

Oury and Swierstra's *So* proposition matches up with the comparison against *'True* used in the Haskell code. Note that the Haskell definitions are very similar to these definitions, gaining no extra annotations or syntactic infelicities other than the use of *singletons*. The Haskell version even preserves whether individual parameters are implicit or explicit. This direct comparison shows that writing Haskell code using singleton types can be a good approximation of Agda.

5.6.1 Constraining the schema

There is a weakness lurking beneath the surface in this example compared to the Agda version. Oury and Swierstra also propose an improvement to their definition of the *Schema* type, introducing a constraint that each attribute in a schema is distinct from all those that come after it. This improvement cannot be made to the Haskell version because, no matter whether the constraint is given implicitly (through a class constraint) or explicitly (through a GADT parameter), the constrained *Schema* type would no longer be promotable to the type level [Yorgey et al. 2012]. A more complete discussion of this restriction appears in Section 7.

5.6.2 Functional vs. relational reasoning

A careful comparison will pick up one key change between the two versions in the types of the *Project* constructors. In Agda, the validity of the projection is guaranteed by *So* (*sub s' s*); in Haskell, it is guaranteed by *SubsetC s' s*. This subtle change makes a large difference in the implementation of the *query* function, as we describe below.

Using the* sub *function The Agda *Project* constructor uses the function *sub* to guarantee that the schema *s'* is a subset of the schema *s*. (The *So* primitive asserts that the result of this function is *'True*.) To understand how this works, we consider a Haskell definition of *sub*, promoted to the type function *Sub* (which requires a standard *lookup* function, elided):

```
$(singletons [d|
  sub :: Schema → Schema → Bool
  sub (Sch []) _ = True
  sub (Sch ((Attr name u) : attrs)) s' =
    lookup name s' ≡ u ∧ sub (Sch attrs) s' |])
```

Recall that the *extractElt* function, which retrieves an element from a row, is an important component of projection. In this version, this function uses *Lookup* to compute the correct return type.

```
extractElt :: Lookup name s ~ u ⇒
  Sing name → Row s → El u
```

The context where we call this function has the constraint *Sub s' s* ~ *'True*. Pattern matching gives us the case where *s'* is not empty. It is equal to some *'Sch (('Attr name u) ': attrs)*. In that case, corresponding to the second clause of *sub*, above, the constraint reduces to *(Lookup name s :≡: u :∧ Sub ('Sch attrs) s)* ~ *'True*, from the definition of *Sub*, where *:≡:* and *:∧* are Boolean equality and conjunction at the type level.[15] To call *extractElt*, we must satisfy the constraint *Lookup name s* ~ *u*, which GHC cannot immediately derive from the above.

We can use clever pattern matching on results that we already know to manipulate this constraint so that *(Lookup name s' :≡: u)* ~ *'True* is in the context. However, this constraint is distinct from the

[13] This is another instance of our simplified treatment of strings. The desugared version can be with our posted code found online.

[14] If we include the case for *EmptyRow* in the case statement, GHC rightly issues an error that we have written inaccessible code. However, if we omit this case and compile with warnings for incomplete patterns, GHC wrongly warns us that the pattern match is incomplete. This infelicity is in the GHC Trac database, ticket #3927. Until that bug is fixed, programmers should try all alternatives and manually remove those ones that the compiler labels as inaccessible code, as we have done in the code presented here. To suppress the warning, we include the wildcard case seen above. The *bugInGHC* function simply calls *error* with an appropriate message.

[15] As part of promoting and refining datatypes that derive the type class *Eq*, the singletons library generates instances for *:≡:* and an instance for the type class *SEq*, the singleton counterpart of *Eq*.

desired one, $Lookup\ name\ s' \sim u$. Getting to our goal requires the following function, which reflects an equality function call $:\equiv:$ to an equality constraint.

$$\begin{aligned}&\textbf{data}\ Eql :: \kappa \to \kappa \to * \ \textbf{where}\\ &\quad Refl :: Eql\ x\ x\\ &boolToProp :: \forall\ (u_1 :: U)\ (u_2 :: U).\ (u_1 :\equiv: u_2) \sim\ 'True \Rightarrow\\ &\quad Sing\ u_1 \to Sing\ u_2 \to Eql\ u_1\ u_2\end{aligned}$$

The $boolToProp$ function can be defined by an exhaustive pattern-match on all possibilities for u_1 and u_2, using recursion in the VEC case. Of course, those matches that give u_1 and u_2 different values are rejected as impossible by the $(u_1 :\equiv: u_2) \sim\ 'True$ constraint.

Using the* SubsetC *relation In this case, we must derive the $InC\ name\ u\ s'$ constraint from the constraint $SubsetC\ s\ s'$, again when s equals $'Sch\ ('Attr\ name\ u\ ':\ attrs)$. In that case, there is only one way to derive the associated $SubsetProof$ extracted from this constraint. Therefore we can pattern match this term against the $SubsetCons$ constructor, of type:

$$\begin{aligned}&SubsetCons :: (InC\ name\ u\ s', SubsetC\ (Sch\ attrs)\ s') \Rightarrow\\ &\quad SubsetProof\ (Sch\ ((Attr\ name\ u)\ ':\ attrs))\ s'\end{aligned}$$

This match directly brings the necessary constraint into scope. In this example, working with relations instead of functions simplifies static reasoning. We discuss this trade-off in more detail below.

6. Expressing type-level constraints

When we wish to express a compile-time constraint, we have at least three options: we can use a Boolean-valued type-level function, a GADT, or a class constraint. All three of these techniques are used above.

Using type families When using the singletons library, it is straightforward to write a function at the term level returning a $Bool$, promote that function to the type level (using the promote or singletons function) and then use the promoted version to satisfy some constraint. This ability, demonstrated with $Disjoint$ above, is the chief advantage of the type families—it corresponds with the way Haskellers already know how to solve problems.

The chief disadvantage of this technique is that it can be difficult to use a constraint defined via a function, as we saw in Section 5.6.2. The type checker may know that the result of the function call is $'True$, but sometimes it is surprising that the compiler cannot use this fact to infer additional information. For example, even if the function has a closed domain, the compiler cannot reason by case analysis (without additional runtime cost). Furthermore, there is no formal connection between Boolean-valued functions (such as $:\wedge$ and $:\equiv:$) and their corresponding constraints.

Using a relation encoded by a GADT Using indexed types, like GADTs, to encode *relations* is standard practice in dependently-typed programming. A GADT provides explicit evidence that the constraint holds—pattern matching the data constructors of a GADT allows programmers to explicitly invert this relation to bring new constraints into the context.

There are two main drawbacks to this approach. All GADT terms must be explicitly built—the compiler does none of the work for us here. Furthermore, GADT-encoded constraints can be trivially satisfied by $\bot$, meaning that the programmer cannot absolutely rely on the constraint.

Using type classes As we saw in the example, multiparameter type classes can also express constraints on the relations between the arguments to a function. In fact, they can work in conjunction with GADTs to allow the compiler to implicitly provide proofs.

A big advantage to using this technique is that the instance cannot be instantiated by $\bot$. Because the compiler produces the instance, a programmer can be sure that an instance exists and is valid. There is no way to spoof the type checker into producing a bogus instance.[16]

Sometimes, when using class constraints, it is necessary to enable either the OverlappingInstances extension or the more ominous-sounding IncoherentInstances extension. The use of these extensions do not reduce the veracity of the proof. Both of these extensions give the compiler the latitude to choose a satisfying instance among a set of choices, perhaps using heuristics such as specificity; without the extensions enabled, the compiler insists on the uniqueness of the available option. The use of IncoherentInstances is similar to the nondeterminism of the use of *auto* in a proof in Coq. In both cases, the programmer lets the compiler choose among a set of options, but she can always be sure that the compiler will choose a valid option if one is chosen at all.

A noted disadvantage to class constraints is that all inference *must* be done by the compiler. It is impossible to provide an explicit instance even when desired.

Conclusion All three techniques have advantages and disadvantages. Many have exploited the power of Haskell's type class mechanism to express rich constraints on types. As GADTs and type families have been introduced, there has been movement away from the logic programming capabilities of type classes and toward a more functional style of programming at the type level. However, as the example above shows, the logic programming facility of type classes still has its place and may offer more direct reasoning.

7. GHC extensions

In this section we discuss a number of extensions to GHC that would better support dependently typed programming.

Unsaturated and injective type families Current versions of Haskell do not allow for unsaturated type families. For example, take the $Plus$ type-level function over Nats discussed earlier, and consider a type-level implementation of Map.

$$Map\ (Plus\ ('Succ\ 'Zero))\ '['Zero, 'Zero]$$

Unfortunately, this is not valid Haskell code. It is not possible to use $Plus$ without supplying both of its parameters. The reason for this restriction is that allowing unsaturated type families interferes with type inference. Consider the case when the type inference engine knows that the types $a\ b$ and $c\ d$ unify. Currently, it is sound to then unify a with c and b with d. If Haskell allowed unsaturated type-level functions, however, this decomposition would not be sound. In the current implementation, all type-level functions must be fully applied, so we know that a and c could never be instantiated by type functions. In the case where the types $F\ b$ and $F\ d$ are unified, for some one-parameter type-level function F, the compiler is aware that F is a type-level function and will not unify b and d. (In this case, the ability to mark certain type families as injective, would allow b and d to unify when it is safe to do so.)

Kind-level equality When rich kinds were introduced into GHC [Yorgey et al. 2012], kind-level equality constraints were explicitly disallowed. That restriction simplified the extension, minimizing the number of changes to GHC necessary to support datatype promotion and kind polymorphism. However, this lack of kind-level equality imposes two restrictions for dependently typed programming: it is impossible to write kind-level functions and to promote GADTs.

[16] It is always possible to write bogus instances and then have the compiler find them. Whenever we refer to the consistency of the constraint language, we mean consistency with respect to the axiom system inferred by the class and family instances declared.

A kind-level function, or kind family, is one that takes a type or kind argument and returns a kind. One example of a kind function is *Promote*. This function reifies the action of datatype promotion, translating a type constant, such as *Nat*, to the corresponding promoted kind (also written *Nat*). This function is the inverse of *Demote*, shown in Section 2.5. Kind-level functions are also necessary for promoted datatypes that use type-level functions.

Among other uses, kind-level functions are necessary to promote datatypes that use type-level functions. In order to implement kind-level functions in the style of type-level functions, it is necessary to have kind-level equality constraints. There is no computation at the type or kind level. Instead, to implement type-level functions, GHC compiles type family instances into a family of equality axioms. For example, the declaration **type instance** $F\ Bool = Int$ compiles to the axiom $F\ Bool \sim Int$. When the type checker encounters a type headed by a type family, it uses these axioms to try to solve its constraints.

Furthermore, GADTs are not promotable. Our experience has shown that this is perhaps the most restrictive shortcoming of Haskell for dependently typed programming. We have presented a re-implementation of one of the dependently typed examples from Oury and Swierstra's *The Power of Pi* above. However, the other two examples in that paper cannot be translated.

For example, the first case study in *The Power of Pi* includes a datatype with the following kind signature (written in Agda):

$$\textbf{data}\ SplitView\ \{A : Set\}\ \{n : Nat\} \to (m : Nat) \to Vec\ A\ (m \times n) \to Set\ \textbf{where}\ \ldots$$

Here, we see an application of *Vec* used as a kind. However, because Haskell cannot promote GADTs, $Vec\ A\ (m \times n)$ is an invalid kind and any translation of the above code is rejected. It is possible to modify the definition of a *SplitView* type to retain its runtime behavior and allow it to be valid Haskell, but such a change would reduce the amount of information encoded in the types.

Weirich et al. [2012] have already started the work necessary to move Haskell in this direction.

Explicit dictionaries The Haskell compiler is responsible for inferring the values of any dictionaries passed to functions declared with class constraints. Though this inference works well in many cases, we have discussed examples above where it would be helpful to be able to specify a dictionary explicitly. The singletons produced by the singletons library work around this restriction by maintaining both explicit terms and implicit dictionaries for all singleton constructors. Named instances [Kahl and Scheffczyk 2001] and Modular Type Classes [Dreyer et al. 2007] may provide a starting point for such an extension.

Totality analysis Under the Curry-Howard isomorphism, Haskell is inconsistent as a logic. That means that Haskell cannot provide the same guarantees that Coq and Agda can. Although rich types mean that Haskell's standard type soundness theorem is very informative (if a program actually produces a value, we have rich information about that value), there is no guarantee that any Haskell program will produce a value. Although not having to show that everything terminates could be considered an advantage, being able to check for totality would improve the confidence in Haskell programs.

There are solutions. A whole program analysis could verify the absence of $\bot$ and show termination using heuristics [Giesl et al. 2011]. GHC's warning for incomplete pattern matching could be improved for GADTs. However, these approaches do not decompose. Either the whole program terminates, or nothing is known: in a lazy language, an otherwise-sound function can diverge when given a pathological argument. A better approach is to identify a total sub-language and use the type system to track it, as in Trellys [Casinghino et al. 2012] and F-Star [Swamy et al. 2011]. Already Haskell supports some of this distinction—all constraint evidence (including coercion proofs) is guaranteed to be total [Vytiniotis et al. 2012].

Adding Π-types to Haskell In a full-spectrum dependently-typed language, a programmer would not use singleton types in the way presented in this paper. Singletons have rightly been called a "poor man's substitute" for dependent types [Monnier and Haguenauer 2010].

It seems possible to enhance Haskell with proper Π-types, whose values are available both at compile time and at runtime, while preserving Haskell's phase distinction. One way to incorporate Π-types would be to have a declaration for a Π-type be syntactic sugar for a singleton type. However, it also seems possible to incorporate these types directly into FC—GHC's internal language—and avoid the singleton encoding altogether.

8. Additional related work

The most closely related work to this one is McBride's Strathclyde Haskell Enhancement (SHE) preprocessor.[17] Among other uses, SHE generates singleton types from datatype definitions. However, because SHE was written before GHC supported datatype promotion, most type-level data, such as numbers and Boolean values, inhabits kind $*$.

The notion of a singleton type was first put forward by Hayashi [1991]. Xi and Pfenning [1998] used singletons to simulate dependent types in their work on eliminating array bound-checking. Chen and Xi [2005] later extended this work with ATS, all the while preserving the phase-distinction via singleton types. Kiselyov and Shan [2007] used a variant of singleton types to provide extra static guarantees. Crary and Weirich [2000] used a kind-indexed definition to create singletons for arbitrary program values. Sheard et al. [2005] showed how combining rich kinds with GADTs can yield dependently typed. Xi and Pfenning [1999], working with ML, and Condit et al. [2007], working with C, have worked on integrating dependently typed features into existing languages.

9. Conclusion

Although Haskell is not a full-spectrum dependently typed language, such as Agda or Coq, recent extensions and the singletons library mean that GHC can be used for dependently-typed programming. As the line between these languages continues to blur, and they adopt the best features of each other, we look forward to more and more programming with rich, statically-checked interfaces.

Acknowledgments Thanks to Simon Peyton Jones, Iavor Diatchki, José Pedro Magalhães, Dimitrios Vytiniotis, Conor McBride, and Brent Yorgey for their collaboration and feedback. Thanks also to the anonymous reviewers for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant No. 1116620.

References

L. Augustsson. Cayenne—a language with dependent types. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250. ACM, 1998.

A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 157–166. ACM, 2002.

[17] `https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`

J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 345–356. ACM, 2010.

R. S. Bird and R. Paterson. de Bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, Jan. 1999.

B. Bringert, A. Höckersten, C. Andersson, M. Andersson, M. Bergman, V. Blomqvist, and T. Martin. Student paper: HaskellDB improved. In *Proc. 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 108–115. ACM, 2004.

J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, Sept. 2009.

C. Casinghino, V. Sjöberg, and S. Weirich. Step-indexed normalization for a language with general recursion. In Proc. 4th Workshop on *Mathematically Structured Functional Programming,* Tallinn, Estonia, pages 25–39, 2012.

M. M. T. Chakravarty, G. Keller, and S. Peyon Jones. Associated type synonyms. In *Proc. 10th ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253. ACM, 2005.

C. Chen and H. Xi. Combining programming with theorem proving. In *Proc. 10th ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 66–77. ACM, 2005.

J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 90–104. ACM, 2002.

J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proc. 16th European conference on Programming*, ESOP'07, pages 520–535. Berlin, Heidelberg, 2007.

Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 184–198. ACM, 2000.

D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In *Proc. 34th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '07, pages 63–70. ACM, 2007.

J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7:1–7:39, Feb. 2011.

L.-J. Guillemette and S. Monnier. A type-preserving compiler in Haskell. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 75–86. ACM, 2008a.

L.-J. Guillemette and S. Monnier. One vote for type families in Haskell! In *Proc. 9th Symposium on Trends in Functional Programming*, 2008b.

S. Hayashi. Singleton, union and intersection types for program extraction. In *Proc. International Conference on Theoretical Aspects of Computer Software*, TACS '91, pages 701–730. Springer-Verlag, London, UK, 1991.

W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In *Proc. 2001 ACM SIGPLAN Workshop on Haskell*, Haskell '01, pages 71–99. ACM, 2001. See also: http://ist.unibw-muenchen.de/Haskell/NamedInstances/.

O. Kiselyov and C.-c. Shan. Lightweight static capabilities. *Electron. Notes Theor. Comput. Sci.*, 174(7):79–104, June 2007.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proc. 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107. ACM, 2004.

D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proc. 2nd conference on Domain-Specific Languages*, DSL '99, pages 109–122. ACM, 1999.

J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *Proc. 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 108–118. ACM, 2000.

J. P. Magalhães. The right kind of generic programming. To appear at WGP, 2012.

C. McBride. Faking it simulating dependent types in Haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

C. McBride. Epigram, 2004. `http://www.dur.ac.uk/CARG/epigram`.

S. Monnier and D. Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Proc. 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 1–8. ACM, 2010.

M. Neubauer and P. Thiemann. Type classes with more higher-order polymorphism. In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 179–190. ACM, 2002.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

C. Okasaki. From fast exponentiation to square matrices: an adventure in types. In *Proc. 4th ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 28–35. ACM, 1999.

N. Oury and W. Swierstra. The power of Pi. In *Proc. 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 39–50. ACM, 2008.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61. ACM, 2006.

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proc. 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 341–352. ACM, 2009.

T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proc. 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002.

T. Sheard, J. Hook, and N. Linger. GADTs + extensible kind system = dependent programming. Technical report, Portland State University, 2005. `http://www.cs.pdx.edu/~sheard`.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proc. 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 266–278. ACM, 2011.

D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: A compiler pearl. To appear at ICFP, 2012.

S. Weirich. Type-safe cast: Functional pearl. *J. Funct. Program.*, 14(6): 681–695, 2004.

S. Weirich, J. Hsu, and R. A. Eisenberg. Down with kinds: adding dependent heterogeneous equality to FC (extended version). Technical report, University of Pennsylvania, 2012. URL `http://www.cis.upenn.edu/~sweirich/papers/nokinds-extended.pdf`.

H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proc. ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 249–257. ACM, 1998.

H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 214–227. ACM, 1999.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '03, pages 224–235. ACM, 2003.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proc. 8th ACM SIGPLAN workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66. ACM, 2012.