Ghostbuster: A Tool for Simplifying and Converting GADTs

Trevor L. McDonell, Timothy A. K. Zakian*, Matteo Cimini, Ryan R. Newton

Indiana University, Oxford University*

{ mcdonelt, mcimini, rrnewton }@indiana.edu, timothy.zakian@cs.ox.ac.uk

Abstract

Generalized Algebraic Dataypes, or simply GADTs, can encode non-trivial properties in the types of the constructors. Once such properties are encoded in a datatype, however, *all* code manipulating that datatype must provide proof that it maintains these properties in order to typecheck. In this paper, we take a step towards *gradualizing* these obligations. We introduce a tool, Ghostbuster, that produces simplified versions of GADTs which elide selected type parameters, thereby weakening the guarantees of the simplified datatype in exchange for reducing the obligations necessary to manipulate it. Like *Ornaments*, these simplified datatypes preserve the recursive structure of the original, but unlike Ornaments we focus on information-preserving bidirectional transformations. Ghostbuster generates type-safe conversion functions between the original and simplified datatypes, which we prove are the identity function when composed. We evaluate a prototype tool for Haskell against thousands of GADTs found on the Hackage package database, generating simpler Haskell'98 datatypes and round-trip conversion functions between the two.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

Keywords Haskell, Datatypes, GADTs

1. Introduction

Languages in the Haskell, OCaml, Agda, and Idris traditions can encode complicated invariants in datatype definitions. This introduces safety at the cost of complexity. For example, consider the standard GADT (generalized algebraic datatype) formulation of length-indexed lists:

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

Although this datatype provides additional static guarantees—for example, that we can not take the head of an empty list—writing functions against this type necessarily involves additional work to manage the indexed length type `n`. In some situations however, such as when prototyping a new algorithm, the user may prefer to *delay* the effort required to fulfill these type obligations until they can verify that the new algorithm is beneficial. In that case, we can convert the length-indexed list into a regular list by *erasing* the length index `n` and operating over a simplified representation:

```
data Vec' a = VNil' | VCons' a (Vec' a)
```

before converting back to the original datatype in order to test the changes within the larger code base.

However, this final step requires re-establishing the type-level invariants that were encoded in the original datatype, which may not be straightforward. Perhaps the user should stick to regular ADTs for this project? Unfortunately, that too may not be an option. In the 16,183,864 lines of public Haskell code we surveyed, we found 11,213 existing GADTs. A person tasked with working in an *existing* project is unlikely to be able to reimplement all of a project's datatypes and operations on them from scratch.

Inspired by the theory of *Ornaments* [10, 15], we can think about moving between families of related datatypes that have the same recursive structure: rather than always working with a GADT, a user could choose to (initially) write code against a simpler datatype, while still having it seamlessly interoperate with code using the fancier one. A practical tool to do this could enable a *gradual* approach to discharging obligations of indexed datatypes. In this paper, we present such a tool. We require that it (1) define canonical simplified datatypes; and (2) create conversion functions between the original and simplified representations.

Is it possible to define such a simplification strategy by merely choosing which type indices to remove from a datatype? While such a method would be convenient for the user, it is far from obvious that there exists a class of datatypes for which such an erasure selection yields a canonical simplified datatype *and* guarantees that conversion functions can successfully round-trip all values of the GADT through the simplified representation and back.

In this work, we show how to do exactly that. Using our tool—named *Ghostbuster*—the user simply places the following pragma above the definition of `Vec`:

```
{-# Ghostbuster: synthesize n #-}
```

and Ghostbuster will generate the definition `Vec'` as well as conversion functions between the two representations:

```
downVec :: Typeable n ⇒ Vec a n → Vec' a
upVec   :: Typeable n ⇒ Vec' a → Maybe (Vec a n)
```

Since `upVec` may fail at runtime if the actual size of the vector does not match the expected size as specified (in the type) by the caller, we return the result wrapped in `Maybe`, but equivalently we could throw an `error` on failure or return a diagnostic message using `Either`. If we do not know the specific type index that must be *synthesized* during the conversion process, we can keep it *sealed* under an existential binding and still make use of it in contexts that operate over any instance of the sealed type:

```
data SealedVec a where
   SealedVec :: Typeable n ⇒ Vec a n → SealedVec a

upVecS   :: Vec' a → SealedVec a
withVecS :: SealedVec a → (∀ n. Vec a n → b) → b
```

Assuming we had such functionality, would that truly make our lives any easier, or have we just moved our type-checking responsibilities elsewhere? We will show that manipulating these simplified—or ghostbusted—datatypes is not at all burdensome, and can indeed make life simpler. As an example, consider implementing deserialization for our indexed list. With Haskell'98 datatypes such as `Vec'`, a `Read` instance can be derived automatically, but an attempt to do so with the `Vec` GADT results in a cryptic

error message mentioning symbols and type variables only present in the compiler-generated code. Disaster! On the other hand, by leveraging Ghostbuster we can achieve this almost trivially:[1]

```
instance (Read a, Typeable n) ⇒ Read (Vec a n) where
  readsPrec i s =
    [ (v,s) | (v',s) ← readsPrec i s
            , let Just v = upVec v' ]
```

In this paper we scale up the above type-index erasure approach to handle a large number of datatypes automatically. We make the following contributions.

- We introduce the first practical solution to incrementalize the engineering costs associated with GADTs.
- We give an algorithm for deleting *any* type variable that meets a set of non-ambiguity criteria. Our ambiguity criteria establish a *gradual erasure guarantee*: if a multi-variable erasure is valid, then any subset of these variables also forms a valid erasure (Section 5).
- We formalize the algorithm in the context of a core language. We show that down-conversion functions are total and down-then-up is exactly the identity function on all values in the original GADT (Section 6).
- We show how the encoding of dynamically typed values that emerges from the algorithm can be asymptotically more efficient than a traditional type `Dynamic` (Section 2.2).
- Viewed in the context of the literature on `deriving` type class instances for datatypes, Ghostbuster increases the reach of deriving capabilities beyond previous functional language implementations, by lifting derivations on simpler types to fancier ones, as with Read above (Section 3).
- We describe the Ghostbuster tool, currently implemented as a source-to-source translator for Haskell, but directly generalizable to other languages. We evaluate the runtime performance of Ghostbuster conversions compared to the ad-hoc approach to constructing GADTs using a runtime `eval`, and apply it to existing datatypes in all 9026 packages on the Hackage Haskell package server (Section 8).

Although our approach does not handle all datatypes or Haskell features, it clearly delineates the class of valid erasures and lays the groundwork for future research.

2. Design Constraints

The central facility provided by Ghostbuster is a method to allow users to select a subset of type variables of a given GADT, from which we derive a new datatype that does not contain those type variables—they have been *erased* from the datatype. Furthermore, we generate a *down-conversion* function from the original datatype to the newly generated one, as well as an *up-conversion* function from the simplified type back to the original, re-establishing type-level invariants as necessary.[2]

Before jumping into the details of how this is implemented, we first highlight some of the different problems that can occur when attempting to erase a type variable from a GADT, which will give us some intuition on the design constraints and behavior of Ghostbuster. Section 3 explores a larger example in more detail.

2.1 Prerequisite: Testing Types at Runtime

Ghostbuster blurs the line between having a statically-typed and dynamically-checked program. With Ghostbuster, we can explicitly remove type-level information in one part of the program (down-conversion), which we then re-establish at some later point (up-conversion). To accomplish this, a central requirement for Ghostbuster is the ability to examine types at runtime and to take action based on those tests. Haskell has supported (open-world) type representations for years via the `Typeable` class:

```
class Typeable a where          -- GHC-7.10 (current)
  typeRep :: proxy a → TypeRep
```

However, this is insufficient for our purposes because examining a `TypeRep` value gives us no type-level information about the type that value represents. Instead, we require a *type-indexed* type representation, which makes the connection between the two visible to the type system:

```
class Typeable a where          -- GHC-8.2
  typeRep :: TypeRep a
```

This type-indexed representation is currently in development for GHC and is scheduled for release as part of GHC-8.2.[3] We assume the presence of this new design throughout the paper, although until this new `Typeable` design becomes available in GHC proper we generate these type-indexed `TypeRep` values ourselves.[4,5]

We can then use the following functions to compare two types and gain type-level information when those types are equal:

```
eqT  :: (Typeable a, Typeable b) ⇒ Maybe (a :∼: b)
eqTT :: TypeRep a → TypeRep b → Maybe (a :∼: b)

data a :∼: b where
  Refl :: a :∼: a
```

2.2 Erasure Method: Checked versus Synthesized

The basic operation that we provide to users is the ability to erase type variables from a GADT. However, there are restrictions on which type variables are valid erasure candidates. Consider the standard list:

```
{-# Ghostbuster: synthesize a #-}   -- invalid!
data List a = Nil | Cons a (List a)
```

If we remove the type parameter `a` and attempt to *synthesize* it when converting back to the original datatype, we will find that it is not possible to write this up-conversion function. In contrast to our initial `Vec` example (Section 1), if we remove the information about the type of the list elements, we cannot later infer that information based solely on the recursive structure of the list.

For this reason, we allow a second, weaker form of type index erasure. Given the declaration:

```
{-# Ghostbuster: check a #-}
```

Ghostbuster will generate the following simplified representation of `List` together with its conversion functions:

[1] Admittedly, this instance would be improved if the constructors of our simplified datatype used the exact same names as the original, but we append an apostrophe to constructor and type names as a convention to clearly distinguish the generated, simplified datatypes.

[2] Note that this directionality metaphor is reversed with respect to the one associated with subtyping in object-oriented languages, where "upcast" is a total function and "downcast" is partial. Our (core) language has no subtyping relation, and the directionality metaphor that we chose is that fancier types are higher.

[3] `https://ghc.haskell.org/trac/ghc/wiki/Typeable`

[4] For simplicity our local `TypeRep` and `Typeable` definitions do make a closed-world assumption, but since we require only the interface we have shown here, and as will appear in GHC-8.2 for an *open* world of types, there is no loss of generality.

[5] Additionally, the use of embedded `TypeRep` *values* rather than embedded `Typeable` class constraints simplifies our core language (Section 4).

```
data List' = Nil'
           | ∀ a. Cons' (TypeRep a) a List'

downList :: Typeable a ⇒ List a → List'
upList   :: Typeable a ⇒ List' → Maybe (List a)
```

In contrast to `Vec'`, where the erased type was synthesized during up-conversion, when erasing type variables in checked mode we must *embed* a representation of the type directly into the constructor `Cons'`, otherwise this information will be lost. We refer to the type parameter `a` as *newly existential*, as it was not existentially quantified in the original datatype fed to Ghostbuster. It is *only* newly existential type variables that require an explicit type representation to be embedded within the simplified datatype. This is important, as we surely do not want the user to have to create and manipulate `TypeRep` values for all erased parameters.

During up-conversion, we check that each element of the list does indeed have the same type the user expects:

```
upList :: ∀ a. Typeable a ⇒ List' → Maybe (List a)
upList Nil'              = Just Nil
upList (Cons' a' x xs') = do
  Refl ← eqTT a' (typeRep :: TypeRep a)
  xs   ← upList  xs'
  return (Cons x xs)
```

Compare this to the definition of up-conversion for our original `Vec` datatype, which erased its type-indexed length parameter in synthesized mode:

```
upVecS :: Vec' a → SealedVec a
upVecS VNil'          = SealedVec VNil
upVecS (VCons' x xs') =
  case upVecS xs' of
    SealedVec xs → SealedVec (VCons x xs)

upVec :: Typeable n ⇒ Vec' a → Maybe (Vec a n)
upVec (upVecS → SealedVec v) = gcast v
```

This highlights the key difference between erasures in checked versus synthesized mode. In order to perform up-conversion on `List'` we must examine the type of each element and compare it to the type that we expect; thus, we can not create a `SealedList` which hides the type of the elements, since we would not know what type to compare against in order to perform the conversion. In contrast, up-conversion for `Vec'` does *not* need to know a-priori what the type `n` should be; only if we wish to open the `SealedVec` do we need to check (via `Data.Typeable.gcast`) that the type that was synthesized is indeed the type we anticipate.

Connection to dynamic typing We note that this embedded type representation essentially makes each list element a value of type `Dynamic`. Why then do we use explicit, *unbundled* type representations when `Dynamic` has existed in Haskell for years? For the `List` type above, we would perform the same $O(n)$ number of runtime type checks with either approach, but consider the following list-of-lists datatype:

```
data LL a where
  NilL  :: LL a
  ConsL :: [a] → LL a → LL a
```

These two competing approaches would each yield the following simplified types for `ConsL`:

```
ConsL'_dyn :: [Dynamic] → LL' → LL'
ConsL'_rep :: TypeRep a → [a] → LL' → LL'
```

Thus, during up-conversion the former would require a runtime type check on every element of the *inner* list, whereas our unbundled representation requires only a single check for each element of the *outer* list—an improvement in asymptotic efficiency. This is one reason that we design Ghostbuster to inject explicit type representations using `TypeRep`.

Finally, this observation suggests an appealing connection to gradual typing—when Ghostbusted, data structures that were refined by type indexing become regular, parametrically polymorphic data structures, which in turn become dynamic datatypes once all type parameters are erased.

2.3 Unrecoverable Information

Consider the following definition of a strange binary tree:

```
{-# Ghostbuster: synthesize a #-}   -- invalid!
data Bad a where
  Leaf :: x → Bad x
  Node :: Bad y → Bad z → Bad z
```

Here, only the rightmost leaf of the tree is usable, since every leftward branch is of some unknown, unusable type `y`. According to our policy of embedding an explicit type representation for any newly-existential types (Section 2.2) we will add a `TypeRep` to the `Leaf` constructor to record the erased type `x`. However, what type representation do we select for `y`? Since this type is already unknowable in the original structure we can not possibly construct its type representation, so such erasures are not supported.

2.4 A Policy for Allowed Erasures

As we saw in Section 2.2, the defining characteristic of which mode a type variable can be erased in is determined by whether the erased information can be recovered from what other information remains. As a more complex example (which we explore further in Section 3) consider the application case for an expression language:

```
{-# Ghostbuster: check env, synthesize ans #-}
data Exp env ans where
  App :: Exp e (a → b) → Exp e a → Exp e b
```

Why does the type variable `a`, which is existentially quantified, not cause a problem? It is because `a` is a *pre-existing* existential type (not made existential by a Ghostbuster erasure). The type `a` can be synthesized by recursively processing fields of the constructor, unlike the `Bad` example above. Thus, we will not need to embed a type representation so long as we can similarly rediscover in the simplified datatype the erased type information at runtime. This is an information-flow criterion that has to do with how the types of the fields in the data constructor constrain each other.

Checked mode: right to left In the `App` constructor, because the `env` type variable is erased in checked mode, its type representation forms an *input* to the `upExp` up-conversion function. This means that since we know the type `e` of the result `Exp e b` (on the right), we must be able to determine the `e` in the fields to the left, namely in `Exp e a` and `Exp e (a → b)`. Operationally, this makes sense if we think how the `upExp` function must call itself on each of the fields of the constructor, passing the (same) representation for the type `e` to each recursive call.

Synthesized mode: left to right Conversely, the type `ans` forms part of the *output* of the up-conversion process, since this type is synthesized by `upExp`, and we only check after the conversion that the generated type is the type that we anticipate. This means that the recursive calls on the fields of the constructor will generate the types `a → b` and `a` from the left, which in turn are used to determine the output type `b` on the right.

Fortunately, whether or not type variables `a` and `b` can be determined by examining the other types in the constructor is a purely *local* check that can be determined in isolation on a per-constructor/per-datatype basis.[6] The same local reasoning holds

[6] This is more local than other (tangentially related) features such as the "." notation in Idris [5] and Agda, which signifies a type is runtime-irrelevant

for the requirements on checked types as well as synthesized. We formalize these information flow checks in Section 5.

Erased types that escape Ghostbuster performs one final check before declaring an erasure is valid: datatypes undergoing erasure can only be used directly in the fields of a constructor, not as arguments to other type constructors. For example, what should the behavior be if we attempt to erase the type variable `a` in the following:

```
data T a where
  MkT :: [T a] → T a
```

We might expect a sufficiently clever implementation to notice that it can utilize the `Functor` instance to apply up- and down-conversion to each element of the list. But what if the type constructor does not have a `Functor` instance, or is only exported abstractly, thereby prohibiting further analysis? This appears to be a tricky design space, so our current implementation simply rules out these erasures altogether.

3. Life with Ghostbuster

In this section, we describe several scenarios in which Ghostbuster can make life easier, taking as a running example the simple expression language which we define below.

3.1 A Type-safe Expression Language

Implementing type-safe abstract syntax trees (ASTs) is perhaps the most common application of GADTs. Consider the following language representation:[7]

```
data Exp env ans where
  Con :: Int → Exp e Int
  Add :: Exp e Int → Exp e Int → Exp e Int
  Var :: Idx e a → Exp e a
  Abs :: Typ a → Exp (e, a) b → Exp e (a → b)
  App :: Exp e (a → b) → Exp e a → Exp e b
```

Each constructor of the GADT corresponds to a term in our language, and the types of the constructors encode both the type that that term evaluates to (`ans`) as well as the type and scope of variables in the environment (`env`). This language representation enables the developer to implement an interpreter or compiler which will statically rule out any ill-typed programs and evaluations. For example, it is impossible to express a program in this language which attempts to `Add` two functions.

Handling variable references is an especially tricky aspect for this style of encoding. We use typed de Bruijn indices (`Idx`) to project a type `t` out of a type level environment `env`, which ensures that bound variables are used at the correct type [3].

```
data Idx env t where
  ZeroIdx ::              Idx (env, t) t
  SuccIdx :: Idx env t → Idx (env, s) t
```

Finally, our tiny language has a simple closed world of types `Typ`, containing `Int` and `(→)`.

Using GADTs to encode invariants of our language (above) into the type system of the host language it is written in (Haskell) amounts to the static verification of these invariants every time we run the Haskell type checker. Furthermore, researchers have shown that this representation does indeed scale to realistically sized compilers: Accelerate [8, 16, 17] is an embedded language in Haskell for array programming which includes optimizations and code generation all written against a GADT AST, maintaining these well-typed invariants through the entire compiler pipeline.

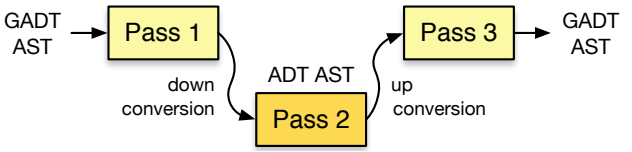

Figure 1. In this scenario, we wish to add a prototype transformation into a compiler that uses sophisticated types, but against a simpler representation. For example, we may want to verify that an optimization does indeed improve performance, before tackling the type-preservation requirements of the GADT representation.

Where then does the approach run into trouble? The problem is that manipulating this representation requires the developer to discharge a (potentially non-trivial) proof to the type system that all of these invariants are maintained. As such, the programmer's time may be spent searching for a type-preserving formulation of their algorithm, rather than working on the algorithm itself. While ultimately such effort is justified in that it rules out entire classes of bugs from the compiler, we question whether or not this effort should be required *up front*, and wonder if, without this extra initial burden, other optimizations or language features might have been implemented by the Accelerate authors and external contributors over the life of the project so far.

In the next section, we discuss how Ghostbuster can be used to realize the situation shown in Figure 1, where we wish to implement a *prototype* transformation over our expression language, without needing to discharge all of the typing obligations up front. Of course, other alternatives exist:

Competing approach #1: hand-written conversions Rather than using a tool such as Ghostbuster, a user could just as well build the same conversion functions to and from a less strictly typed AST representation themselves. Indeed, Ghostbuster itself is written entirely in Haskell, and no modifications to the GHC compiler were required to support it. However, this introduces a maintenance burden. Moreover, these conversion functions are tricky to implement, and since the Haskell type checker can not stop us from writing ill-typed conversions to or from our untyped representation, these errors will only be caught when the runtime type tests fail.

Competing approach #2: runtime eval Another approach is to avoid the fine-grained runtime type checks necessary for up-conversion entirely, by generating the GADT term we require as a *string*, and using GHC embedded as a library in our program to typecheck (`eval`) the string at runtime. Implementing a pretty-printer is arguably less complex than the method we advocate in this work, but there are several significant disadvantages to this approach which we will demonstrate in Section 8.

3.2 Example #1: Substitution

Consider the task of inlining a term into all use sites of a free variable. For our richly-typed expression language, where the types of terms track both the type of the result as well as the type and scope of free variables, this requires a type-preserving but environment changing value-level substitution algorithm. Luckily, the simultaneous substitution method of McBride [14] provides exactly that, where renaming and substitution are instances of a single traversal, propagating operations on variables structurally through terms. Listing 1 outlines the method.

Although the simultaneous substitution algorithm is very elegant, we suspect that significant creativity was required to come up with it. Compare this to the implementation shown in Listing 2,

and should be erased during compilation. However, this requires a whole-program check to verify whether the annotation can be fulfilled.

[7] `https://github.com/shayan-najd/MiniFeldspar`

```
class Syntactic f where
  varIn  :: Idx env t → f env t
  expOut :: f env t → f env t
  weaken :: f env t → f (env, s) t

instance Syntactic Idx
instance Syntactic Exp

shift :: Syntactic f
      ⇒ (∀ t'. Idx env t' → f env' t')
      → Idx (env,  s) t
      → f   (env', s) t
shift _ ZeroIdx      = varIn ZeroIdx
shift v (SuccIdx ix) = weaken (v ix)

rebuild :: Syntactic f
        ⇒ (∀ t'. Idx env t' → f env' t')
        → Exp env  t
        → Exp env' t
rebuild v exp =
  case exp of
    Var ix  → expOut (v ix)
    Abs t e → Abs t (rebuild (shift v) e)
    ...

substitute :: Exp (env, s) t → Exp env s → Exp env t
substitute old new = rebuild (subTop new) old
  where
    subTop :: Exp env s → Idx (env, s) t → Exp env t
    subTop = ...
```

Listing 1. Substitution algorithm for richly-typed terms

which is just a simple structural recursion on terms. In particular, this is implemented against the simplified representation generated by Ghostbuster using the erasure pragma:[8,9]

```
{-# Ghostbuster: check env, synthesize ans #-}
```

which yields the following expression datatype:

```
data Exp' where
  Con' :: Int → Exp'
  Add' :: Exp' → Exp' → Exp'
  Mul' :: Exp' → Exp' → Exp'
  Var' :: Idx' → Exp'
  Abs' :: Typ' → Exp' → Exp'
  App' :: Exp' → Exp' → Exp'

downExp :: (Typeable env, Typeable t)
        ⇒ Exp env t → Exp'

upExp:  :: (Typeable env, Typeable t)
        ⇒ Exp' → Maybe (Exp env t)
```

Referring to the implementation of Listing 2, note that although the `Var'` and `Abs'` cases constitute environment changing operations, we do *not* need to manipulate any embedded `TypeRep env` values; needing to do so would seriously compromise usability, and Ghostbuster is instead able to recover this information automatically (see Sections 2.2 and 2.4).

3.3 Example #2: Template Haskell and typeclass deriving

One great feature of Haskell is its ability to automatically derive certain standard typeclass instances such as `Show` and `Read` for

[8] The environment type `env` needs to be provided by the client (checked mode) because otherwise it is ambiguous. For example, the constant term `Con 42` can be typed in any environment.

[9] We simultaneously request erased versions of `Idx` and `Typ` using the same settings, but elide those for brevity.

```
shift :: Idx' → Exp' → Exp'
shift j exp =
  case exp of
    Var' ix | ix < j    → Var' ix
            | otherwise → Var' (SuccIdx' ix)
    Abs' t e → Abs' t (shift (SuccIdx' j) e)
    ...

substitute :: Exp' → Exp' → Exp'
substitute = go ZeroIdx'
  where
    go j old new =
      case old of
        Var' ix | ix == j                  → new
                | ix > j, SuccIdx' i ← ix → Var' i
                | ix < j                   → old
        Abs' t e → Abs' t (go (SuccIdx' j) e
                              (shift ZeroIdx' new))
        ...
```

Listing 2. Substitution algorithm implemented against the simplified datatype generated by Ghostbuster

Haskell'98 datatypes. Unfortunately, attempting to do the same for GADTs results only in disappointment and cryptic error messages from compiler-generated code. However, as we saw in Section 1, we can regain this capability by using Ghostbuster and leveraging the derived instances on those simpler datatypes instead.

Similarly, some libraries include Template Haskell [22] routines that can be used to automatically generate instances for the typeclasses of that library. Although these run into problems when applied to GADTs, once more we can use Ghostbuster to circumvent this limitation. As an example, we can easily generate JSON (de)serialization instances for the `aeson` package[10] applied to our richly-typed terms:

```
$(deriveJSON defaultOptions ''Exp')

instance (...) ⇒ ToJSON (Exp env t) where
  toJSON = toJSON . downExp

instance (...) ⇒ FromJSON (Exp env t) where
  parseJSON v = do
    v' ← parseJSON v :: Parser Exp'
    return $ fromMaybe (error "...") (upExp v')
```

These examples demonstrate that Ghostbuster enables a *synergy* with existing Haskell libraries and deriving mechanisms, providing a convenient method to lift these operations to GADTs.

4. Core Language Definition

Before covering host-language-specific aspects of our Ghostbuster implementation, we first formalize a core language to facilitate the precise description of the transformations performed by the Ghostbuster tool. This core language also serves as the intermediate representation of the Ghostbuster implementation. Although we implement our prototype in Haskell, it is easily extended to generate code for any language that supports GADTs.

The core language definition is given in Figure 3. The input to Ghostbuster is a set of datatype definitions, $dd_1 \ldots dd_n$. The term language is used only as an *output language* for generating up- and down-conversion functions. As such, we are not interested in the problem of type inference for GADTs, rather we assume type annotations that allow us to use the permissive, natural type system for GADTs [21], which supports decidable checking [9, 24] (but

[10] `https://hackage.haskell.org/package/aeson`

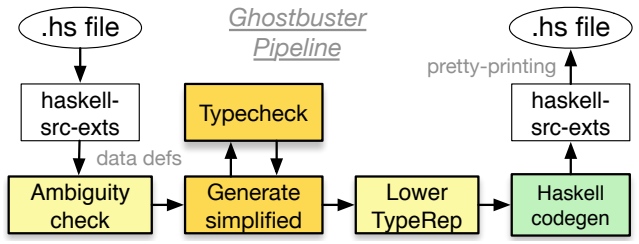

Figure 2. The architecture of the Ghostbuster tool, which processes data definitions in several passes, resulting in pretty-printed Haskell source on disk. Note that only the ingestion and code generation phases are Haskell specific: the ambiguity check through lowering phases are implemented in terms of our core language.

Programs and datatype declarations

$$
\begin{array}{rcl}
prog & ::= & dd_1 \ldots dd_n;\ vd_1 \ldots vd_m;\ e \\
dd & ::= & \texttt{data}\ T\ \overline{k}\ \overline{c}\ \overline{s}\ \texttt{where} \\
 & & \quad \texttt{K} :: \forall\ \overline{k}, \overline{c}, \overline{s}, \overline{b}. \\
 & & \qquad \tau_1 \to \cdots \to \tau_p \to T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s} \\
vd & ::= & x :: \sigma;\ x = e
\end{array}
$$

Data constructors	K		
Type constructors	T, S		
Type variables	a, b, k, c, s		
Monotypes	τ	$::=$	$a \mid \tau \to \tau \mid T\ \overline{\tau}$
		$\mid$	$\texttt{TypeRep}\ \tau$
Type Schemes	σ	$::=$	$\tau \mid \forall \overline{a}.\tau$
Term variables	x, y, z		
Constraints	C, D	$::=$	$\epsilon \mid \tau \sim \tau \mid C \wedge C$
Substitutions	ϕ	$::=$	$\varnothing \mid \phi, \{a := \tau\}$

$$
\begin{array}{rcrl}
\text{Terms} & e & ::= & K \mid x \mid \lambda x :: \tau.e \mid e\ e \\
 & & \mid & \texttt{let}\ x :: \sigma = e\ \texttt{in}\ e \\
 & & \mid & \texttt{case}[\tau]\ e\ \texttt{of}\ [p_i \to e_i]_{i \in I} \\
 & & \mid & \texttt{typerep}\ \mathbb{T} \\
 & & \mid & \texttt{typecase}[\tau]\ e\ \texttt{of} \\
 & & \mid & \quad (\texttt{typerep}\ \mathbb{T})\ x_1 \ldots x_n \to e \mid \_ \to e \\
 & & \mid & \texttt{if}\ e \simeq_\tau e\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{Patterns} & p & ::= & K\ x_1 \ldots x_n \\
\text{Type names} & \mathbb{T} & ::= & T \mid ArrowTy \mid Existential
\end{array}
$$

Figure 3. The core language manipulated by Ghostbuster

not inference). Our implementation runs a checker for this type system, and, to support checking, `case` and `typecase` forms are labelled with their return types as well, though we will elide these in the code through the rest of the paper.

4.1 Syntax

The syntax of terms and types in Figure 3 resembles Haskell syntax with extensions for type representation handling and extra conventions related to type arguments ($\overline{k}\ \overline{c}\ \overline{s}$) to indicate the erasure level (respectively, type variables which are kept unchanged in the output, and those which are erased in checked and synthesized mode, as discussed in Section 2.2). Without loss of generality, we assume that type constructor arguments are *sorted* into these kept, checked, and synthesized categories. This simplifies the discussion of which type arguments occur in which *contexts*, based on position. The implemented Ghostbuster tool does not have this restriction and the status of type arguments are specified in pragmas, as we saw earlier.

A program consists of a number of datatype declarations followed by mutually-recursive value definitions (vd) and a "main" term e. The generated up- and down-conversions will form a series of vds. Terms in our language consist of the lambda calculus, a non-recursive `let` with explicit type signatures, simple `case` expressions and ways of creating, casing on, and querying equality of runtime type representations, which we call (`typerep`, `typecase`, and $\simeq_\tau$). The $\simeq_\tau$ operator must work over arbitrary monotype representations, comparing them for equality at runtime. `typecase` also performs runtime tests on type representations, and enables *deconstructing* type representations into their component parts—for example, splitting a function type into a input type and output type.

We deviate from the standard presentation of GADTs. Typically, the return type of each constructor is normalized to the form $T\ \overline{a}$, with any constraints on the output type pushed into a per-data-constructor constraint store (C):

$$\texttt{K}_i :: \forall \overline{a}, \overline{b}.C \Rightarrow \tau_1 \to \cdots \to \tau_p \to T\ \overline{a}$$

We avoid this normalization. Because we lack type class constraints in the language (and equality constraints over existentially-bound variables can easily be normalized away), we simply omit per-data-constructor constraints. This means that when scrutinizing a GADT with `case`, we must synthesize constraints equating the scrutinee's type $T\ \overline{\tau}$ with $T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$ in each K_i clause and then add this into a constraint store C, which we will use during type-checking (Figure 5). The advantage is that avoiding per-constructor constraints greatly simplifies our definition of the allowable space of input datatypes for Ghostbuster (Section 5). The absence of per-constructor type-class constraints from our core language is also why we require type-indexed `TypeRep` *values* (rather than equivalent `Typeable` *constraints*) to observe the type of newly existential type variables (Sections 2.1 and 2.2).

4.2 Type System

The typing rules for our language are syntax-directed and are given in Figures 4 to 7. Many of the typing rules are standard, but a few—in particular `Pat`, `TypeRep`, `TypeCase`, and `IfTyEq`—are unique to our language and separated into Figure 4. Here the `TypeRep` and `TypeCase` rules only cover the type constructor T cases, the elided rules for the built-in type representations $\mathbb{T} = Existential$ and $\mathbb{T} = ArrowTy$ are nearly identical.

We lack a full kind system, but we do track the arity of constructors, with $T : \overline{\star}^n \in \Gamma$ as a shorthand for the T being an arity-n type constructor. We require that all type constructors be fully applied except when referenced by name through the (`typerep` T) form.

5. Preconditions and Ambiguity Checking

Before Ghostbuster can generate down- and up-conversion functions, it first performs a sanity check that the datatypes, together with requested parameter erasures, meet all preconditions necessary for the tool to generate well-typed conversion functions. Indeed, as we discussed in Section 2 not every erasure setting is valid. We therefore want to create sufficient preconditions such that *if* these preconditions are met, the Ghostbuster tool is guaranteed to generate a pair of well-typed functions $(down, up)$, such that down-conversion followed by up-conversion is a total identity function. This section details these preconditions and ambiguity criteria.

5.1 Ambiguity Test

The goal of the ambiguity criteria are a concise specification of the class of programs handled by Ghostbuster. These non-ambiguity prerequisites apply per-data-constructor, $\texttt{K}_i$, per-datatype that requests a type erasure (nonempty $\overline{c}$ or $\overline{s}$ variables). If all constructors of the datatypes undergoing erasure individually pass the ambiguity check, then the input program as a whole is valid. For a given

$$\frac{C, \Gamma \vdash_e \texttt{typerep}\ T : \overline{\texttt{TypeRep}\ a}^n \to \texttt{TypeRep}\ (T\ \overline{a}^n) \qquad C, \Gamma \vdash_e e : \texttt{TypeRep}\ a_0 \quad C \wedge (a_0 \sim T\ \overline{a}^n), \Gamma \cup \{x_1 : \texttt{TypeRep}\ a_1, \ldots, x_n : \texttt{TypeRep}\ a_n\} \vdash_e e' : \tau \qquad C, \Gamma \vdash_e e'' : \tau}{C, \Gamma \vdash_e \texttt{typecase}[\tau]\ e\ \texttt{of}\ ((\texttt{typerep}\ T)\ x_1 \ldots x_n) \to e' \mid \_ \to e'' : \tau}\ \text{TypeCase}$$

$$\frac{T : \overline{\star}^n \in \Gamma}{C, \Gamma \vdash_e \texttt{typerep}\ T : \overline{\texttt{TypeRep}\ a}^n \to \texttt{TypeRep}\ (T\ \overline{a}^n)}\ \text{TypeRep} \qquad \frac{C, \Gamma \vdash_e e_1 : \texttt{TypeRep}\ \tau_1 \quad C, \Gamma \vdash_e e_2 : \texttt{TypeRep}\ \tau_2 \quad C \wedge (\tau_1 \sim \tau_2), \Gamma \vdash_e e' : \tau \qquad C, \Gamma \vdash_e e'' : \tau}{C, \Gamma \vdash_e \texttt{if}\ e_1 \simeq_\tau e_2\ \texttt{then}\ e'\ \texttt{else}\ e'' : \tau}\ \text{IfTyEq}$$

Figure 4. Typing rules for type representations and operations on them

$$\boxed{C, \Gamma \vdash_p p \to e : \tau_1 \to \tau_2}$$

$$\frac{(K : \forall \overline{k}\ \overline{c}\ \overline{s}, \overline{b}.\ \overline{\tau_x}^p \to T\ \overline{\tau'}^m) \in \Gamma \qquad fv(C, \Gamma, \overline{\tau}^m, \tau_r) \cap \overline{b} = \varnothing \qquad D = \left(\bigwedge_{i=1\ldots m} \tau'_i \sim \tau_i\right) \qquad C \wedge D, \Gamma \cup \{\overline{x : \tau_x}^p\} \vdash_e e : \tau_r}{C, \Gamma \vdash_p K\ \overline{x}^p \to e : T\ \overline{\tau}^m \to \tau_r}\ \text{Pat}$$

$$\boxed{C, \Gamma \vdash_e e : \tau}$$

$$\frac{(x : \forall \overline{a}.\tau') \in \Gamma \qquad \phi = \{\overline{a := \tau}\}}{C, \Gamma \vdash_e x : \phi(\tau')}\ \text{Var} \qquad \frac{C, \Gamma \cup \{x : \tau_x\} \vdash_e e : \tau}{C, \Gamma \vdash_e \lambda x :: \tau_x . e : \tau_x \to \tau}\ \text{Lam}$$

$$\frac{C, \Gamma \vdash_e e_1 : \tau_1 \to \tau_2 \quad C, \Gamma \vdash_e e_2 : \tau_1}{C, \Gamma \vdash_e e_1\ e_2 : \tau_2}\ \text{App} \qquad \frac{C, \Gamma \vdash_e e : \tau \qquad \forall i \in I.\ C, \Gamma \vdash_p p_i \to e_i : \tau \to \tau'}{C, \Gamma \vdash_e \texttt{case}\ [\tau']\ e\ \texttt{of}\ [p_i \to e_i]_{i \in I} : \tau'}\ \text{Case}$$

$$\frac{C, \Gamma \vdash_e e_1 : \tau_1 \quad C, \Gamma \cup \{x : \forall \overline{a}.\tau_1\} \vdash_e e_2 : \tau_2}{C, \Gamma \vdash_e \texttt{let}\ x :: \forall \overline{a}.\tau_1 = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{Let} \qquad \frac{C, \Gamma \vdash e : \tau_1 \quad C \models \tau_1 \sim \tau_2}{C, \Gamma \vdash e : \tau_2}\ \text{Eq} \qquad \frac{(K : \forall \overline{a}.\tau') \in \Gamma \qquad \phi = \{\overline{a := \tau}\}}{C, \Gamma \vdash_e K : \phi(\tau')}\ \text{Con}$$

Figure 5. Typing rules for the core language

$$\frac{}{C \models \epsilon}\ \text{True} \qquad \frac{}{C \models \tau \sim \tau}\ \text{Refl} \qquad \frac{C \models \tau_2 \sim \tau_1}{C \models \tau_1 \sim \tau_2}\ \text{Sym} \qquad \frac{}{C_1 \wedge C_2 \models C_2}\ \text{GivenR} \qquad \frac{}{C_1 \wedge C_2 \models C_1}\ \text{GivenL}$$

$$\frac{C \models \tau_1 \sim \tau_2 \quad C \models \tau_2 \sim \tau_3}{C \models \tau_1 \sim \tau_3}\ \text{Trans} \qquad \frac{C \models C_1 \quad C \models C_2}{C \models C_1 \wedge C_2}\ \text{Conj} \qquad \frac{\overline{C \models \tau_i \sim \tau'_i}}{C \models T\ \overline{\tau}_i \sim T\ \overline{\tau'}_i}\ \text{TStruct}$$

$$\frac{C \models T\ \overline{\tau}_i \sim T\ \overline{\tau'}_i}{\overline{C \models \tau_i \sim \tau'_i}}\ \text{TCon} \qquad \frac{\overline{C \models \tau_i \sim \tau'_i}}{C \models \tau_1 \to \tau_2 \sim \tau'_1 \to \tau'_2}\ \text{ArrStruct} \qquad \frac{C \models \tau_1 \to \tau_2 \sim \tau'_1 \to \tau'_2}{\overline{C \models \tau_i \sim \tau'_i}}\ \text{ArrCon}$$

$$\frac{C \models \tau_1 \sim \tau_2}{C \models \texttt{typerep}\ \tau_1 \sim \texttt{typerep}\ \tau_2}\ \text{TyRepStruct} \qquad \frac{C \models \texttt{typerep}\ \tau_1 \sim \texttt{typerep}\ \tau_2}{C \models \tau_1 \sim \tau_2}\ \text{TyRepCon}$$

Figure 6. Equality Theory for the Ghostbuster Type System

$$\boxed{\Gamma \vdash_v \overline{vd} : \Gamma'}$$

$$\frac{\overline{\epsilon, \Gamma' \vdash_e e : \tau} \quad \Gamma' = \Gamma \cup \{\overline{x : \forall \overline{a}.\tau}\}}{\Gamma \vdash_v \overline{x ::\forall \overline{a}.\tau; x = e} : \Gamma'} \text{ VDef}$$

$$\boxed{\vdash_d \overline{dd} : \Gamma'}$$

$$\frac{}{\vdash_d \texttt{data}\ T\ \overline{a}^n\ \texttt{where}\ \overline{K :: \sigma} : \Gamma \cup \{\overline{T : \star^n, K : \sigma}\}} \text{ Data}$$

$$\boxed{\vdash_{prog} prog : \tau}$$

$$\frac{\vdash_d \overline{dd} : \Gamma_d \quad \Gamma_d \vdash_v \overline{vd} : \Gamma_v \quad \epsilon, \Gamma_v \vdash_e e : \tau}{\vdash_{prog} \overline{dd}; \overline{vd}; e : \tau} \text{ Prog}$$

Figure 7. Environment and program typing rules

data constructor:

$$\mathtt{K}_i :: \forall \overline{k}, \overline{c}, \overline{s}, \overline{b}.\tau_1 \to \cdots \to \tau_p \to T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$$

We refer to the types τ_i through τ_p as the *fields* of the constructor, and the $T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$ expression as the right-hand side (RHS). Types which occur in a checked or synthesized *context* means that they occur within the arguments of some type constructor T in positions corresponding to its $\overline{c}$ or $\overline{s}$ type parameters. Likewise, *kept* (or *non-erased*) context $\overline{k}$ refers to all types τ that are *not* in checked or synthesized context.

The ambiguity check is concerned with *information flow*. That is, whether the erased information can be recovered based on properties of the simpler datatype. If not, then these type variables would not be recoverable upon up-conversion and Ghostbuster rejects the program.

5.2 Type Variables Synthesized on the RHS

For each synthesized type $\tau' \in \overline{\tau_s}$ on the RHS, type variables occurring in that type, $a \in Fv[\![\tau']\!]$, must be computable based on:

- occurrences of a in any of the fields $\overline{\tau}^p$. That is, $\exists i \in [1, p]\ .\ a \in Fv_s[\![\tau_i]\!]$, using the $Fv_s[\![]\!]$ function from Figure 8; or
- $a \in Fv[\![\overline{\tau_k}]\!]$. That is, kept RHS types; or
- $a \in Fv[\![\overline{\tau_c}]\!]$. That is, a occurs in the *checked* (input) type.

Note that the occurrences of a in fields can be in kept or in synthesized contexts, but *not* checked. For example, consider our `Exp` example (Section 3.1), where the `a` variable in the type of an expression `Exp e a` is determined by the synthesized `a` component of its sub-expressions, bottoming out at leaf expressions such as constants and variables. In contrast, checked variables in the fields must be created by the up-conversion function as *inputs* to recursive up-conversion calls on the value's fields. Thus they cannot be a source of new information to determine synthesized outputs, and we use the $Fv_s[\![]\!]$ rather than $Fv[\![]\!]$ metafunction above. Conversely, notice that we do not worry about applying the above prerequisites to synthesized variables inside fields—these are the *outputs* of recursive up-conversion calls. Their computability is left to an inductive argument (bottoming out at "leaf" constructors such as `Exp`'s `Con`).

5.3 Type Variables in Checked Context

All types in checked context in $\tau_1 \ldots \tau_p$ are implicit arguments to the up-conversion function that will process that field. Thus for all τ_i in checked context, all $a \in Fv[\![\tau_i]\!]$ must be computable based on information available *at that point*, which includes:

- kept or checked variables in the RHS, $a \in Fv[\![\overline{\tau_c}]\!] \cup Fv[\![\overline{\tau_k}]\!]$
- occurrences of a in non-erased context within *any* field
- occurrences of a in $Fv[\![\tau_j]\!]$, for other fields τ_j that have already been processed before the field containing τ_i.

This last case—inter-field dependencies—can be found in the `Abs` case of our expression language (Section 3.1):

```
Abs :: Typ a → Exp (e, a) b → Exp e (a → b)
```

Recall that in our example, given `Exp e a`, we erase `e` in checked mode and `a` in synthesized mode. Thus the type `(e, a)` is in checked context, so how is it determined? It cannot be resolved using `(a → b)` on the RHS, as this is a synthesized type (meaning it is an *output* of the up-conversion function); it must be determinable from the other fields of the constructor, in this case `Typ a`.

For a type in checked context, we must be able to determine which fields to examine in order to determine what the checked type should be. This requires that any inter-field dependencies do not form a cycle. As an example, we can not erase the type `t` from `Loop`, because the types `a` and `b` in the fields of the constructor are in checked mode but depend on each other:

```
{-# Ghostbuster: synthesize t #-}   -- invalid!
data Loop t where
  MkLoop :: T a b → T b a → Loop (a, b)

{-# Ghostbuster: check a, synthesize b #-}
data T a b where
  MkT :: a → b → T a b
```

For simplicity our formal language assumes that fields are already topologically sorted so that dependencies are ordered left to right. That is, a field τ_{i+k} can depend on field τ_i. In the case of `Abs`, `a` $\in Fv_s[\![$`Typ a`$]\!]$ and τ_1 = `Typ a` occurs before τ_2 = `Exp (e,a) b`, therefore Ghostbuster accepts the definition.

5.4 Gradual Erasure Guarantee

One interesting property of the class of valid inputs described by the above ambiguity check is that it is always valid to erase *fewer* type variables—to change an arbitrary subset of *erased* variables (either $\overline{c}$ or $\overline{s}$) to *kept* ($\overline{k}$). That is:

Theorem 1 (Gradual erasure guarantee)**.** *For a given datatype with erasure settings $\overline{k}$, $\overline{c} = \overline{c_1}\ \overline{c_2}$ and $\overline{s} = \overline{s_1}\ \overline{s_2}$, then erasure settings $\overline{k}' = (\overline{k}\ \overline{c_2}\ \overline{s_2})$, $\overline{c}' = \overline{c_1}$, $\overline{s}' = \overline{s_1}$ will also be valid.*

Proof. The requirements above are specified as a conjunction of constraints over *each* type variable in synthesized or checked position. Removing erased variables removes terms from this conjunction. For the remaining erased type variables, their dependence check may have depended on formerly erased, now kept, variables. However, both the synthesized and checked dependency prerequisites include all variables in kept context. Thus, moving variables from erased to kept context never breaks any dependency. □

6. Core Translation Algorithms

We now describe the core translation algorithms used in Ghostbuster using the language defined in Section 4. The resulting pipeline of translation passes is shown in Figure 2.

Fv_s: extracting dependencies for synthesized type variables

$$
\begin{array}{rcl}
Fv_s[\![a]\!] & = & \{a\} \\
Fv_s[\![\tau_1 \ldots \tau_n]\!] & = & \bigcup_{i=1}^{n} Fv_s[\![\tau_i]\!] \\
Fv_s[\![\tau_1 \to \tau_2]\!] & = & Fv_s[\![\tau_1]\!] \cup Fv_s[\![\tau_2]\!] \\
Fv_s[\![T\,\overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}]\!] & = & Fv_s[\![\overline{\tau_k}]\!] \cup Fv_s[\![\overline{\tau_s}]\!]
\end{array}
$$

Fv_k: extracting free vars in non-erased context

$$
\begin{array}{rcl}
Fv_k[\![a]\!] & = & \{a\} \\
Fv_k[\![\tau_1 \ldots \tau_n]\!] & = & \bigcup_{i=1}^{n} Fv_k[\![\tau_i]\!] \\
Fv_k[\![\tau_1 \to \tau_2]\!] & = & Fv_k[\![\tau_1]\!] \cup Fv_k[\![\tau_2]\!] \\
Fv_k[\![T\,\overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}]\!] & = & Fv_k[\![\overline{\tau_k}]\!]
\end{array}
$$

Figure 8. Extracting free type variables in different contexts.

6.1 Simplified Datatype Generation

Creating simplified data definitions is straightforward. Fields τ_i are replaced with updated versions, τ_i', that replace all type applications $T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$ with $T'\ \overline{\tau_k}$:

$$
\begin{array}{c}
\mathtt{K}_i : \forall \overline{k}, \overline{c}, \overline{s}, \overline{b}.\tau_1 \to \cdots \to \tau_p \to T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s} \\
\Rightarrow \\
\mathtt{K}_i' : \forall \overline{k}, \overline{b}.\ \mathrm{getTyReps}(\mathtt{K}_i) \to \tau_1' \to \cdots \to \tau_p' \to T\ \overline{\tau_k}'
\end{array}
$$

Where $getTyReps$ returns any newly existential variables for a constructor (Section 2.2):

$$
\begin{array}{c}
\mathrm{getTyReps}(\mathtt{K}_i : \forall \overline{k}, \overline{c}, \overline{s}, \overline{b}.\tau_1 \to \cdots \to \tau_p \to T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}) = \\
\{\mathtt{TypeRep}\ a \mid a \in (Fv_k[\![\tau_1 \ldots \tau_p]\!] - Fv[\![\overline{\tau_k}]\!]) - \overline{b}\}
\end{array}
$$

Recall here that $\overline{b}$ are the preexisting existential type variables that do not occur in $\overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$.

6.2 Down-conversion Generation

In order to generate the down-conversion function for a type T, we instantiate the following template:

```
downT_i :: TypeRep c → TypeRep s → T_i k c s → T'_i k
downT_i c1_typerep ... sn_typerep orig =
 case orig of
 K_j x_1 ... x_p →
  let φ       = unify(T k c s, T τ_k τ_c τ_s)
      KtyRep_j = map (λτ →bind(φ, [τ], buildTyRep(τ)))
                     getTyReps(K)
  in
  K_j' KtyRep_j
    dispatch↓(φ, x_1, φ(τ_1)) ... dispatch↓(φ, x_p, φ(τ_p))
```

The Supplemental Material (Section B) includes the full, formal specification of down/up generation, but the procedure is straightforward: pattern match on each $\mathtt{K}_j$ and apply the $\mathtt{K}_j'$ constructor. The complexity is in the type representation management of the $bind$ and $dispatch_{\downarrow}$ operations. Here we follow a naming convention where a type variable k is witnessed by a type representation bound to a term variable $k\_typerep$. Ghostbuster performs a renaming of type variables in data definitions to ensure there is no collision between the variables used at the declaration head $T\ \overline{k}\ \overline{c}\ \overline{s}$, and those used within each constructor K_j. For example, this already holds in `Exp` where we used `env`/`ans` interchangably with `e`/`a`.

In the let-binding of ϕ above, we unify the type of `orig` with the expected result type of K_j. This uses a unification function that is part of a type checking algorithm based on the type system of Figures 4 to 7. Because we use the $\overline{k}\ \overline{c}\ \overline{s}$ variables to refer to the type of the input, `orig`, this gives us a substitution binding these type variables. For example, in the `Abs` case of our expression language (Section 3.1):

```
Abs :: Typ r → Exp (e, r) s → Exp e (r → s)
```

unification yields:

$$\phi = \{env := e, ans := (r \to s)\ \}$$

It is the job of $bind$ to navigate this substitution in order to create type representations for type variables mentioned in ϕ, such as r. Here, getting to r requires digging inside the type representation for ans using a `typecase` expression. Because the type representation added to K_j' will always be of the form `TypeRep` a (for type variable a), this is all the call to $bind$ must do to create the type representations that decorate K_j'. Note that there may be multiple occurrences of $r \in \phi$, and thus multiple *paths* that bind might navigate; which path it chooses is immaterial.

Type representation construction in dispatch The $dispatch_{\downarrow}$ function is charged with recursively processing each field `f` of $\mathtt{K}_j$. Based on the type of `f` this will take one of two actions:

- Opaque object: return it unmodified.
- Ghostbusted type T: call `down`T.

In the latter case, it is necessary to build type representation arguments for the recursive calls. This requires not just accessing variables found in ϕ, but also building compound representations such as for the pair type `(e, r)` found in the `Abs` case of `Exp`.

Finally, when building type representations inside the $dispatch_{\downarrow}$ routine, there is one more scenario that must be handled: representations for pre-existing existential variables, such as the type variable `a` in `App`:

```
App :: Exp e (a → b) → Exp e a → Exp e b
```

In recursive calls to `downExp`, what representation should be passed in for `a`? We introduce an explicit `ExistentialType` in the output language of the generator which appears as an implicitly defined datatype such that `(typerep Existential)` is valid and has type `∀ a. TypeRep a`.

Theorem 2 (Reachability of type representations)**.** *All searches by $bind$ for a path to v in ϕ succeed.*

Proof. By contradiction. Assume that $v \notin \phi$. But then v must not be mentioned in the $T_i\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$ return type of K_j. This would mean that v is a preexisting existential variable, whereas only newly existential variables are returned by $getTyReps$. □

6.3 Up-conversion Generation

Up-conversion is more challenging. In addition to the type representation binding tasks described above, it must also perform runtime type tests ($\simeq_\tau$) to ensure that constraints hold for formerly erased (now restored) type variables. The type signature of an up-converter takes type representation arguments only for checked type variables; synthesized types must be computed:

$$\mathtt{up}T_i\ ::\ \overline{\mathtt{TypeRep}\ c} \to T_i'\ \overline{k} \to \mathtt{Sealed}T_i'\ \overline{k}\ \overline{c}$$

If the set of synthesized variables is empty, then we can elide the `Sealed` return type and return $T_i'\ \overline{k}\ \overline{c}$ directly. This is our strategy in the Ghostbuster implementation, because it reduces clutter that the user must deal with. However, it would also be valid to create sealed types which capture no runtime type representations, and we present that approach here to simplify the presentation.

To invert the `down` function, `up` has the opposite relationship to the substitution ϕ. Rather than being *granted* the constraints ϕ by virtue of a GADT pattern match, it must test and witness those same

constraints using $(\simeq_\tau)$. Here the initial substitution ϕ_0 is computed by unification just as in the down-conversion case above.

```
upTi c1_typerep ... cm_typerep lower =
  case lower of
    K'j ex_typerep ... f1 ... fp →
      let φ0 = ... in
      openConstraints(φ0, openFields(f1...fp))
  where
  openConstraints(∅, bod) = bod

  openConstraints(a := b : φ, bod) =
    if a_typerep ≃τ b_typerep
    then openConstraints(φ, bod)
    else genRuntimeTypeError

  openConstraints(a := T τ1 ... τn : φ, bod) =
    typecase a_typerep of
      (typerep T) a1_typerep ... an_typerep →
        openConstraints(a1 := τ1, ..., an := τn : φ, bod)
      _ → genRuntimeTypeError
```

Again, a more formal and elaborated treatment can be found in the Supplemental Material (Section B). Above we see that *openConstraints* has two distinct behaviors. When equating two type variables, it can directly issue a runtime test. When equating an existing type variable (and corresponding `_typerep` term variable) to a compound type $T\ \overline{\tau}^n$, it must break down the compound type with a different kind of runtime test (`typecase`), which in turn brings more `_typerep` variables into scope. We elide the $(\rightarrow)$ case, which is isomorphic to the type constructor one. Note that $(\simeq_\tau)$ works on any type of representation, but this algorithm follows the convention of only ever introducing variable references (e.g. `a_typerep`) to "simple" representations of the form `TypeRep` a.

Following *openConstraints*, *openFields* recursively processes the field arguments $\mathtt{f}_1 \ldots \mathtt{f}_p$ from left to right:

```
openFields(f::T τk τc τs : rst) =
  case openRecursion(φ0,f) of
    SealedTq s'_typerep f' →
      openConstraints(unify(s'_typerep, τs)
                      , openFields(rst))

openFields(f::τ : rst) =
  let f' = f in openFields(rst)
```

Here we show only the type constructor $(T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s})$ case and the "opaque" case. We again omit the arrow case, which is identical to the type constructor one.

As before with $dispatch_\downarrow$, the *openRecursion* routine must construct type representations to make the recursive calls. Unsealing the result of a recursive call reveals more constraints that must be checked. For example, in the `Add` case of `Exp`, both recursions must synthesize a return type of `Int` and thus a type representation inside the `Sealed` type of `(typerep Int)`. Likewise, in the `App` case the function input and the argument types must match. *openConstraints* ensures these synthesized values are as expected before returning control to *openFields* to process the rest of the arguments.

Finally, in its terminating case, *openFields* now has all the necessary type representations in place that it can build the type representation for $\mathtt{SealedT}_i$. Likewise, all the necessary constraints are present in the typing environment—from previous `typecase` and $(\simeq_\tau)$ operations—enabling a direct call to the more strongly typed $\mathtt{K}_j$ constructor.

```
openFields(∅) =
  SealedTi buildTyRep(s_typerep) (Kj f'1 ··· f'p)
```

The result of code generation is that Ghostbuster has augmented the *prog* with up- and down-conversion functions in the language of Figure 3, including the `typecase` and $(\simeq_\tau)$ constructs. What remains is to eliminate these constructs and emit the resulting program in the target language, which, in our prototype, is Haskell.

6.4 Validating Ghostbuster

We are now ready to state the main Ghostbuster theorem: down-conversion followed by up-conversion is the identity after unsealing synthesized type variables.

Theorem 3. ***Round-trip*** *Let* `prog` *be a program, and let* $\mathbf{T} = \{(T_1, k_1, c_1, s_1), \ldots, (T_n, k_n, c_n, s_n)\}$ *be the set of all datatypes in* `prog` *that have variable erasures. Let* $\mathbf{D} = \{D_1, \ldots, D_n\}$ *be a set of dictionaries such that* $D_i = (D_i s, D_i c)$ *contains all needed typeReps for the synthesized and checked types of* T_i*. We then have that if for each* $(T_i, k_i, c_i, s_i) \in \mathbf{T}$ *that* T_i *passes the ambiguity criteria, then Ghostbuster will generate a new program* $\mathtt{prog}'$ *with busted datatypes* $\mathbf{T}' = \{(T'_1, k_1), \ldots, (T'_n, k_n)\}$*, and functions* $\mathtt{down}T_i$ *and* $\mathtt{up}T_i$ *such that*

$$\forall e \in \mathtt{prog}.\ \mathtt{prog} \vdash e :: T_i\ k_i\ c_i\ s_i \wedge (T_i, k_i, c_i, s_i) \in \mathbf{T} \implies \mathtt{prog}' \vdash (\mathtt{down}T_i\ D_i\ e) :: T'_i\ k_i, \text{ where } (T'_i, k_i) \in \mathbf{T}' \tag{1}$$

and

$$\forall e \in \mathtt{prog}.\ \mathtt{prog} \vdash e :: T_i\ k_i\ c_i\ s_i \wedge (T_i, k_i, c_i, s_i) \in \mathbf{T} \implies \mathtt{prog}' \vdash (\mathtt{up}T_i\ D_i c\ (\mathtt{down}T_i\ D_i\ e)) \equiv (\mathtt{Sealed}T_i\ D_i s\ e :: \mathtt{Sealed}T_i\ k_i\ c_i) \tag{2}$$

The full proof including supporting lemmas are listed in the Supplemental Material (Section C). We provide a brief proof-sketch here.

Proof Sketch. We first show by the definition of down-conversion that given any data constructor K of the correct type, that the constructor will be matched. We then proceed by induction on the type of the data constructor, and by case analysis on *bind* and $dispatch_\downarrow$. We then show that the `map` of *bind* over the types found in the constructor K succeeds in building the correct `typeReps` needed for the checked fields of K. We then show that every individual type-field is down-converted successfully and that this down-conversion preserves values. From this we conclude that since we have managed to construct the correct type representations needed for the down-converted data constructor K', and since we can successfully down-convert each field of K, that the application of K' to the typeReps for the newly-existential types and the down-converted fields is well typed, and that the values that we wish to have preserved have been kept.

To show that up-conversion succeeds, we first show that given any data constructor K' of the correct type that the up-conversion function will match it. We then proceed by case analysis on the code-path executed on the right-hand-side of the `case` clause that matched the data constructor: we show that *openConstraints* succeeds in deriving suitable type representations for the call to *openRecursion* to succeed in constructing the correct up-converted datatypes for each of the busted recursive datatypes in the fields of K'. We then use this to show that *openFields* will succeed in up-converting the busted types that it encounters. We then use the fact that *openFields* has successfully up-converted the types it has encountered, and by the fact that we have succeeded in constructing suitable type representations as we progressed ensures that we are finally able to successfully construct the up-converted sealed type. □

7. Implementing Ghostbuster for Haskell

The Ghostbuster prototype tool is a source-to-source translator, which currently supports Haskell but could be extended to other languages that incorporate GADTs. To build a practical tool implementing Ghostbuster, we need to import data definitions from, and generate code to, a target host language. Because our prototype targets Haskell, we extended our core language slightly to accommodate certain Haskell features of data definitions such as bang patterns. For the most part, code generation is a straightforward translation from our core-language into Haskell using the `haskell-src-exts` package,[11] which we subsequently pretty-print to file. If erasure results in Haskell'98 datatypes, we add `deriving` clauses to the simplified datatypes for the standard instances such as `Show`.

7.1 Current Limitations

Our current prototype comes with some limitations. Yet, as we will see in Section 8.2, a great many of the datatypes found in the wild are supported.

Runtime type representation As mentioned in Section 2.1, we require type-indexd `TypeRep` values, which are scheduled to appear in GHC-8.2. In the meantime, we use our own representation of runtime types synthesized on demand by the Ghostbuster tool and described in the Supplemental Material (Section A.3).

Advanced type system features There are some features we support indirectly by allowing them in the "opaque" regions of the datatype which Ghostbuster-generated code need not traverse, but we do not model explicitly in our core language. This currently includes type families [7, 20] and type classes [11, 19].

Erased datatypes as type parameters As we saw in Section 2.4, Ghostbuster does not allow datatypes undergoing erasure to be used as arguments to other type constructors, for example `[]`. If available, we could lean on a `Functor` instance for that type, but in general there is not a single, clearly defined behaviour. Future work may allow a user to specify how Ghostbuster should traverse under type constructors to continue the erasure and conversion processes.

8. Evaluation

8.1 Runtime Performance

This section analyzes the performance of the conversion routines generated by Ghostbuster. Benchmarks were conducted on a machine with two 12-core Xeon E5-2670 CPUs (64-bit, 2.3GHz, 32GB RAM) running GNU/Linux (Ubuntu 14.04 LTS), using GHC version 7.10.1 at `-O2` optimization level. Each data point is generated via linear regression using the `criterion` package.[12]

Figure 9 compares the performance of the Ghostbuster generated conversion routines for our simple expression language (Section 3.1). We generated large random programs that included all of the important cases of down- and up- conversion (`Abs`, `App`, etc.), and report the time to convert programs containing that number of terms.

Ghostbuster achieves comparable performance to a manually written down-conversion routine. The hand-written written up-conversion routine, however, which uses embedded `Typeable` class constraints is significantly slower than the Ghostbuster generated version with embedded `TypeRep` values. Profiling reveals that our generated `TypeRep` encodings were more efficient than dictionary passing with `Data.Typeable`. However, this may be an artifact of the closed-world simplification we used to generate our `TypeRep` values, so this performance advantage may disappear once we transition to open-world, type-indexed `Typeable` arriving in GHC-8.2.

Even so, the size of the Ghostbuster generated up- and down-conversion functions are comparable to the `Data.Typeable` based implementation:

Contender	SLOC	Tokens	Binary size
Ghostbuster	198	1426	1MB
Data.Typeable	122	1011	1MB
Hint	78	451	45MB

For the up-conversion process, we also compare against using GHC's interpreter as a library via the `Hint` package.[13] Due to the difficulty of writing the up-conversion process manually, it is appealing to be able to re-use the GHC Haskell type-checker itself in order to generate expressions in the original GADT. In this method, a code generator converts expressions in the simplified type into an equivalent Haskell expression using constructors of the original GADT, which is then passed to `Hint` as a string and interpreted, with the value returned to the running program. Unfortunately: (1) as shown in Figure 9, this approach is significantly slower than the alternatives; (2) the conversion must live in the `IO` monad; (3) generating strings of Haskell code is error-prone; and (4) embedding the entire Haskell compiler and runtime system into the program increases the size of the executable significantly.

Nevertheless, before Ghostbuster, this runtime interpretation approach was the only reasonable way for a language implemented in Haskell with sophisticated AST representations to read programs from disk. One DSL that takes this approach is Hakaru.[14]

8.2 Package Survey

We conclude our experimental evaluation by testing our prototype implementation against all 9026 packages currently available on `hackage.haskell.org`, the central open source package archive of the Haskell community. We seek to gather some insight into how many GADTs exist "in the wild" which might benefit from the automated up- and down-conversions explored in this work.

In this survey, we extract all of the ADT and GADT datatype declarations of a package, and group these data declarations into connected components. We elide any connected components where none of the data declarations are parameterised by a type variable, or do not contain at least one GADT. For each connected component, we then vary which type variables are kept, checked, or synthesized, and attempt to run ghostbuster on each configuration. For connected components containing many data types and/or type variables this can yield a huge search space, so we explore at most ten thousand erasure variants for each connected component. A summary of the results are shown in Table 1.

As discussed in Section 5, our current design has some restrictions on what datatypes and erasure settings it will accept. However, out of the variants explored, ghostbuster was successfully able to erase at least one type variable in 2,582,572 cases. Moreover, out of the 8773 "real" GADTs surveyed[15], we were able to successfully ghostbust 5525 (63%) of these down to regular ADTs.

9. Related work

Ornaments [10, 12, 15], from the world of dependent type theory, provides an interesting theoretical substrate for moving between inductive data structures that share the same recursive structure,

[11] `http://hackage.haskell.org/package/haskell-src-exts`

[12] `http://hackage.haskell.org/package/criterion`

[13] `http://hackage.haskell.org/package/hint`

[14] `https://hackage.haskell.org/package/hakaru`

[15] Some types were written in GADT syntax that didn't need to be.

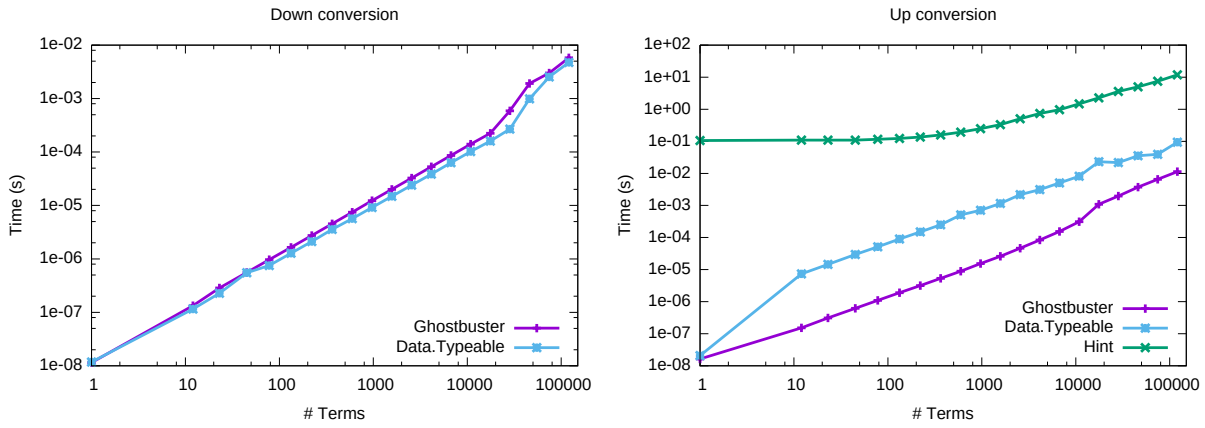

Figure 9. Time to convert a program in our richly-typed expression language (Section 3.1) with the given number of terms (i.e. nodes in the AST), from original GADT to simplified ADT (left) and vice-versa (right). Note the log-log scale.

Metric	
Total # packages	9026
Total # source files	94,611
Total # SLOC	16,183,864
Total # datatypes using ADT syntax	9261
Total # datatypes using GADT syntax	18,004
Total # connected components	15,409
ADTs with type variable(s)	1341
GADTs with type variable(s)	11,213
GADTs with type indexed variable(s)	8773
Actual search space	185,056,322,576,712
Explored search space	9,589,356
Ghostbuster succeeded	2,582,572
GADTs turned into ADTs	5525
Ambiguity check failure	5,374,628
Unimplemented feature in Ghostbuster	1,632,156

Table 1. Summary of package survey

where one type is refined, or ornamented, by adding and removing information. Unlike Ornaments, we focus on *bidirectional* conversions from a richer to simpler type. Recent progress has been made in bringing Ornaments from a theoretical topic to a practical language [28]. This prototype is semi-automated and leaves holes in the generated code for the user to fill in, rather than being an entirely *in language* and *fully-automatic* abstraction like Ghostbuster.

The `eqT` of Haskell's `Typeable` class and the (`typecase`/$\simeq_\tau$) and `TypeRep` of our core language, are both similar to `typecase` and `Dynamic` from Abadi et al. [1, 2]. However, while `typecase` (from dynamic) allows querying the type of expressions, it does not inject type-level evidence about the scrutinee into the local constraints the way that GADT pattern matching (and our `typecase`) do.

Another closely related work is on *staged inference* [23], which formulates dynamic typing as staged checking of a single unified type system. While the mechanism is different, functions over Ghostbusted types defer type-checking obligations until up-conversion. Likewise, Haskell's *deferred type errors* [27] are related, but are a coarse-grained setting at the module level and hence not practical for writing code against GADTs while deferring type-checking obligations.

The Yoneda lemma applied to Haskell provides a method of encoding GADTs as regular ADTs.[16] However, this encoding does not offer the benefits of Ghostbuster simplified types because: (1) the encodings include function types, which preclude `Show/Read` deriving, and (2) the encoding cannot actually enforce its guarantees in Haskell due to laziness (lack of an initial object).

F# type providers [26] are related to Ghostbuster in that both automatically generate datatype definitions against which developers are expected to write code. Type providers do not include GADTs, but deal with type schemas that are too large (e.g. all of Wikipedia) or externally maintained (e.g. in a database) and must be populated dynamically, whereas Ghostbuster deals with maintaining simplified types for existing GADTs.

Checking whether input-output tags are consistent in a logic program is often approximated in practice based on a dependency graph of the variables. For example, the Mercury programming language [25] has tags: input, output, deterministic. Our ambiguity checking process is similar.

Ou et al. [18] define a language that provides interoperability between simply-typed and dependently-typed regions of code. Both regions are encoded in a common internal language (also dependently-typed), with runtime checks when transitioning between regions. Similarly, the Trellys project [6] includes a two-level language design where each definition is labelled logical or programmatic. Because of the shared syntax, one can migrate code from programmatic to logical when ready to prove non-termination.

It is folklore in dependently typed programming communities (Idris, Agda, etc.) that if you need to write a parser for a compiler, you would parse to a raw, untyped term and write a type-checking function (i.e. up-conversion) manually. To our knowledge there are not currently any tools that automate this process. However, most fully dependent languages make these type checkers easier to write than they are in Haskell.

10. Conclusion

We've shown how Ghostbuster enables the automatic maintenance of simplified datatypes that are easier to prototype code against. This resulted in some performance advantages in addition to software engineering benefits. Because of these advantages, we believe that in the coming years gradualization of type checking obligations for advanced type systems will become an active area of work and widely-used language implementations may better support gradualization of type-checking obligations directly.

[16] The Yoneda lemma in Haskell is currently best explained in blog posts: `http://www.haskellforall.com/2012/06/gadts.html` and `http://bartoszmilewski.com/2013/10/08/lenses-stores-and-yoneda/`.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *POPL'98: Principles of Programming Languages*, pages 237–268, 1989.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5:111–130, 1995.

[3] T. Altenkirch and B. Reus. Monadic Presentation of Lambda Terms Using Generalised Inductive Types. In J. Flum and M. Rodriguez-Artalejo, editors, *CSL'99: Computer Science Logic*, pages 453–468, 1999.

[4] A. I. Baars and S. D. Swierstra. Typing dynamic typing. *ICFP'02: International Conference on Functional Programming*, pages 157–166, 2002.

[5] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *TYPES'03: Types for Proofs and Programs*, pages 115–129. Springer, 2004.

[6] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *POPL'14: Principles of Programming Languages*, pages 33–45, 2014.

[7] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *POPL'05: Principles of Programming Languages*, pages 241–253, 2005.

[8] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP'11: Declarative Aspects of Multicore Programming*, pages 3–14, 2011.

[9] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[10] P.-E. Dagand and C. McBride. Transporting functions across ornaments. In *ICFP'12: International Conference on Functional Programming*, pages 103–114, 2012.

[11] C. V. Hall, K. Hammond, S. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *TOPLAS'96: Transactions on Programming Languages and Systems*, 18(2):109–138, Mar. 1996.

[12] H.-S. Ko and J. Gibbons. Relational algebraic ornaments. In *DTP'13: Dependently-Typed Programming*, pages 37–48, 2013.

[13] X. Leroy and M. Mauny. Dynamics in ML. In *Functional Programming Languages and Computer Architecture*, pages 406–426, 1991.

[14] C. McBride. Type-Preserving Renaming and Substitution. *Journal of Functional Programming*, 2006.

[15] C. McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*, (to appear).

[16] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *ICFP'13: International Conference on Functional Programming*, pages 49–60, 2013.

[17] T. L. McDonell, M. M. T. Chakravarty, V. Grover, and R. R. Newton. Type-safe Runtime Code Generation: Accelerate to LLVM. In *Haskell Symposium*, pages 201–212, 2015.

[18] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types (extended abstract). In *TCS'04: International Conference on Theoretical Computer Science*, pages 437–450, August 2004.

[19] J. Peterson and M. Jones. Implementing type classes. In *PLDI'93: Programming Language Design and Implementation*, pages 227–236, June 1993.

[20] T. Schrijvers, S. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP'08: International Conference on Functional Programming*, pages 51–62, 2008.

[21] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP'09: International Conference on Functional Programming*, pages 341–352, 2009.

[22] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, pages 1–16, 2002.

[23] M. Shields, T. Sheard, and S. Peyton Jones. Dynamic typing as staged type inference. In *POPL'98: Principles of Programming Languages*, pages 289–302, 1998.

[24] V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *TOPLAS'07: Transactions on Programming Languages and Systems*, 29(1):1, 2007.

[25] Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an efficient purely declarative logic programming language. *Australian Computer Science Communications*, 17:499–512, 1995.

[26] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *DDFP'13: Data Driven Functional Programming*, pages 1–4, 2013.

[27] D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality Proofs and Deferred Type Errors: A Compiler Pearl. In *ICFP'12: International Conference on Functional Programming*, pages 341–352, 2012.

[28] T. Williams, P.-E. Dagand, and D. Rémy. Ornaments in practice. In *WGP'14: Workshop on Generic Programming*, pages 15–24, 2014.

Appendix A Runtime type representations

Ghostbuster's generated code must choose an approach to dynamic type checks. One method is that of the Haskell `Typeable` class, which we saw in Section 2.1. However, this is only one of several possible approaches, as described by the substantial literature on dynamic type checking in statically typed languages [1, 2, 4, 13].

In this section, we detail for the interested reader some of the trade-offs in runtime type representation, including the current non-type-indexed `Typeable` library, as well as our interim method for generating type-indexed `TypeRep` values.

A.1 Data.Typeable

As we saw in Section 2.1, Haskell's current (GHC-7.10) `Typeable` class uses the following representation for generating type representations at runtime:

```
class Typeable a where
  typeRep :: proxy a → TypeRep
```

Here, `TypeRep` is a simple (non-indexed) datatype which provides a concrete representation of a monomorphic type, which is more restrictive than other designs [2]. As mentioned previously, until the Haskell `Typeable` library is upgraded to a type-indexed representation in GHC-8.2, we generate our own indexed type representations, as we describe next.

A.2 Runtime Types in Ghostbuster

The Ghostbuster core language includes three extensions for working with runtime type representations: `typerep`, `typecase`, and $\simeq_\tau$. By leaving them abstract, we retain some flexibility in how we ultimately generate runtime type tests for these constructs, and portability to target languages other than Haskell.

However, generating code against the current non-type-indexed `Typeable` class is challenging, so we currently use a simple approach for generating a closed-world of type-indexed `TypeRep` values for all types mentioned in the datatypes passed to Ghostbuster. For example, the following is the `TypeRep` for representing Boolean, integer, and tuple types.

```
data TypeRep a where
  IntType  :: TypeRep Int
  BoolType :: TypeRep Bool
  Tup2Type :: TypeRep a → TypeRep b → TypeRep (a,b)
```

A.3 Lowering Type Representation Primitives

Including explicit type representation operations in our core language allows us to defer commitment to a particular representation of runtime type representations. Here we describe how to desugar explicit type representation operations such as `typecase` into the other operations of the core language as a core-to-core transformation. This allows us to lower those operations into operations more directly expressible in the target language (e.g. Haskell).

First, the "Lower TypeRep" pass must introduce a new data definition, `TypeRep a`, with one constructor for each T mentioned anywhere in a `typerep` or `typecase` form, plus the built-in types:

```
data TypeRep a where
  Type_{T_1} :: TypeRep ā^{n_1} →TypeRep T_1
  Type_{T_2} :: TypeRep ā^{n_2} →TypeRep T_2
  ...
  ArrowType :: TypeRep a → TypeRep b
            → TypeRep (a → b)
  ExistentialType :: ∀ a . TypeRep a
```

This datatype, plus propositional type equality `(:∼:)` that we saw earlier, are used by the generated code for the desugared forms, which appears as follows:

```
⟦ typerep T ⟧ ⇒TypeRep_T

⟦ typecase e_1 of
  ((typerep T) a_1 ... a_n) → e_2; _ → e_3 ⟧ ⇒
  case e_1 of
    Type_{T_1} a_1 ... a_n →⟦ e_2 ⟧
    Type_{T_2} _ ...       → ⟦ e_3 ⟧
    ...
```

Here we encounter a tension with `typecase` desugaring. As specified in our core language definition, we do not have "catch all" pattern matches along with the `case` form. Thus the `case` expression generated must match on *every* possible $\texttt{Type}_\tau$ constructor. If generating these exhaustive cases, and $\texttt{e}_3$ produces nontrivial code, it is also important to `let`-bind it to avoid excessive code duplication, which slightly complicates the translation above.

Finally, the third form, $(\simeq_\tau)$, desugars into a call to a type representation equality testing function, `eqTT`:

```
⟦ if e_1 ≃_τ e_2 then e_3 else e_4 ⟧ ⇒
  case eqTT ⟦ e_1 ⟧ ⟦ e_2 ⟧ of
    Just Refl → ⟦ e_3 ⟧
    Nothing → ⟦ e_4 ⟧
```

This `eqTT` value definition (vd) is also produced by the type representation lowering pass and added to the output program. For example, below is an excerpt of generated, pretty-printed code for this function:

```
eqTT :: TypeRep t → TypeRep u → Maybe (t :∼: u)
eqTT x y =
  case x of
    UnitType → case y of
      UnitType       → Just Refl
      Tup2Type a2 b2 → Nothing
      ...
    ...
```

The `eqTT` function performs a simple, recursive traversal of both type representation values. Without catch-all clauses this function will grow quadratically with the number of cases in the type representation sum type.

Appendix B Formalism

B.1 Type System

The main judgement forms are the following:

Well-typed Expressions	Well-typed Patterns
$C, \Gamma \vdash_e e : \tau$	$C, \Gamma \vdash_p p \to e : \tau_1 \to \tau_2$

We also have judgment forms for how to extend Γ for data definitions ($\Gamma \vdash_d dd : \Gamma'$), and value definitions ($\Gamma \vdash_v vd : \Gamma'$), and finally for typechecking whole programs ($\Gamma \vdash_{prog} prog : \tau$). We use the following syntactic sugar for sequences.

$$\overline{\tau}^n \equiv \tau_1, \ldots, \tau_n$$
$$\overline{\tau}^n \to \tau \equiv \tau_1 \to \ldots \to \tau_n \to \tau$$

B.2 GenUp/GenDown

This section gives the formal description of the algorithm and code-generation process performed by Ghostbuster. In order to make the description more concise, we have adopted certain notational conventions which are detailed here:

1. Instead of using ellipsis to represent repeated iteration of an expression parametrized over a set (e.g., constructors or environments) we instead use $(\!|--|\!)$
2. Many times we will have to convert from a non-relational substitution ϕ to a relational substitution ψ. This translation is

largely trivial and so we represent a 'cast' from ϕ to ψ via the operator $(\!|-|\!)$;

3. Code that is $\mathit{generated}$ is distinguished from code that is run in the generation process by the use of a grey background.
4. For a given type-constructor T, we denote the corresponding up-conversion and down-conversion functions by $\widehat{T}$ and $\check{T}$ respectively.
5. To each type τ there is an associated type representation which we write as $\tau_{typerep}$. Thus in the code, when we see $a_{typerep}$ this is the type representation for the type-variable a that is in scope. Finding these type representations is not hard, but is tedious, and thus we elide the code for this and simply remark that we have a function $findPath(\psi, a)$ which, given a relational substitution ψ and a type variable a *generates code* that will access the correct substitution in ψ.
6. For a given data type definition d with type constructor T we denote the type-constructor for the sealed data type of d by $\mathcal{T}$ and overload this symbol to also represent the data constructor for the sealed data type definition.
7. We denote continuations by the Greek letter κ.

Beyond these notational conventions, the definition of unification is standard but is included here for completeness. All definitions can be found in Figures 10 to 12.

Appendix C Full proof for round-trip theorem

In the following section, note that the function $getTyReps$ returns the $TypeRep$s in a *canonical order*. Thus, when we call $getTyReps$ in multiple places, we are guaranteed that $getTyReps$ will always return the type representations in the same order whenever it is fed the same arguments.

Furthermore, throughout the proof we denote the set of type representations for checked types by $\mathcal{D}_c$ and the set of type representations for synthesized types by $\mathcal{D}_s$. These sets may intersect.

Definition 1. Lower Types *Let $prog$ be a program with a to-be busted type constructor T. Let*

$$\texttt{def}(prog, T) \quad = \quad \texttt{data}\ T\ \overline{k}\ \overline{c}\ \overline{s}\ \texttt{where}\ \overline{\texttt{K} :: \forall\ \overline{k}, \overline{c}, \overline{s}, \overline{b}.\ \tau_1 \to \cdots \to \tau_p \to T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}}$$

We then denote the Lower Type *(LT) of T by T' and define T' to be the following data definition:*

$$\texttt{data}\ T'\ \overline{k}\ \texttt{where}\ \overline{\texttt{K}' :: \forall\ \overline{k}.\ \{getTyReps(K)\} \to \tau'_1 \to \cdots \to \tau'_p \to T'\ \overline{\tau_k}}$$

Where each τ'_i is computed via the following rule:

- *$\tau_i = (S\ \tau_{k_i}\ \tau_{c_i}\ \tau_{s_i})\{S$ has erased variables$\}$: We compute the lower type S' for S and return $\tau'_i = S'\ \tau_{k_i}$*
- *$\tau_i = TypeRep\ \tau'$ Then we return τ_i unchanged*
- *$\tau_i = (T\ \overline{\tau})\ \{T$ has no erased variables$\}$: Then we return τ_i unchanged*
- *$(ArrowTy\ \tau'\tau'')$: Then we return τ_i unchanged*
- *$\tau_i = a$ is a type variable: Then we return τ_i unchanged*

and $getTyReps(K)$ is computed as in Figure 10

Lemma 1. $\texttt{map}\ (\lambda\ \tau \to bind(\psi, [\tau], buildTyRep(\tau)))$ *over the result of $getTyReps(K)$ returns all needed type representations for K' and is well-typed.*

Proof. We have by the definition of T'/K' in Definition 1 that the extra type representation fields for K' are exactly those found in $getTyReps(K)$. All that remains to show is that, if $getTyReps(K) = \{TypeRep\ a_1 \ldots, TypeRep\ a_n\}$ then for each a_i we can find a path in ψ such that we can extract the necessary type representation information for each type variable a_i. Now, since we are mapping over $getTyReps$ all we need to show is for each $TypeRep\ a_i \in getTyReps$ that $bind(\psi, [a_i], buildTyRep(a_i))$ will succeed. Note that if $getTyReps(K) = \{\}$ then we return nothing, which is what the LT expects by Definition 1. So now we consider the case in which $getTyReps(K)$ is non-empty.

We proceed by case-analysis on the definition of $bind$ in Figure 10 specialized to $[a_i]$.

1. Case: $a_i \in \psi$ Then we execute the following code:

   $$\texttt{let}\ \tau' = \texttt{lookup}\ a_i\ \psi\ \texttt{in}\ \texttt{let}\ a_{i_{typerep}} = buildTyRep(\tau')\ \texttt{in}\ a_{i_{typerep}}$$

   Then we know that the lookup will succeed by the assumption that $a_i \in \psi$. To see that the call to $buildTyRep$ will indeed return a type representation for τ' we proceed by induction on τ' and case-analysis on $buildTyRep$:
   - Case: $\tau' = a$ In this case we simply return the type representation $a_{typerep}$ for that type variable, which must be in scope due to the fact that we can find $a \in \psi$ (the only way it could be in ψ is if we already have a witness for it)
   - Case: $\tau' = \mathbb{T}\ \overline{\tau''}^m$ For each $\tau'_j \in \overline{\tau''}^m$ we call $buildTyRep(\tau'_j)$ which by the inductive hypothesis will give us a type representation for τ'_j. We then have by our definition of `typerep` that since we can construct the type representation for each $\tau'_j \in \overline{\tau''}^m$ that we can construct

     $$(typerep\ \mathbb{T})\ (TypeRep\ \tau'_1) \ldots\ (TypeRep\ \tau'_m)$$

     which is the type representation for $\tau' = \mathbb{T}\ \overline{\tau''}^m$
   - Case: $\tau' = ArrowTy\ \tau_1\ \tau_2$ This is simply a special case of the type constructor case; by the inductive hypothesis, we have that $buildTyRep$ returns type representation for both τ_1 and τ_2 and therefore that we can construct

     $$(typerep\ ArrowTy)\ (TypeRep\ \tau_1)\ (TypeRep\ \tau_2)$$

     which is the type representation for $\tau' = ArrowTy\ \tau_1\ \tau_2$.

   We therefore have that $buildTyRep(\tau')$ will return the correct type representation for τ'. We then have that we simply create an alias $a_{i_{typerep}}$ for τ's type representation and then return this.
2. Case: $a_i \notin \psi$ This case happens when we may have a witness for a type representation $a_{typerep}$ buried inside some other type representation that is inside ψ e.g., $a_{typerep}$ could be found inside $((typerep\ ArrowTy)\ a_{typerep}\ b_{typerep})$ but we must "dig out" the witness for it from the larger type representation. This is what $findPath$ does. Now, moving on from this digression, in this case, we execute the following code:

   $$\texttt{let}\ a_{i_{typerep}} = findPath(\psi, a_i)\ \texttt{in}\ a_{i_{typerep}}$$

   We then have by Theorem 2 that $findPath$ will find the corresponding type representation for a_i in ψ by looking through the type representations contained in ψ. We then return this new type representation $a_{i_{typerep}}$.

We therefore have that

$$\texttt{map}\ (\lambda\ \tau \to bind(\psi, [\tau], buildTyRep(\tau)))getTyReps(K)$$

returns all needed type representations for K'.

To see that this mapping is well-typed, simply note that the *only* things that bind can generate are those things of type $(TypeRep\ a)$, and therefore it generates the types expected by K'. What's more, due to the canonicity of ordering for $getTyReps$ (and since `map`

$$\boxed{unify(-,-)}$$

$$\begin{array}{rcl}
unify(\texttt{TypeRep}\ \tau_1, \texttt{TypeRep}\tau_2) & = & \varnothing \\
unify(v,t) & = & varBind\ v\ t \\
unify(t,v) & = & varBind\ v\ t \\
unify(\texttt{ConTy}\ name\ typs_1, \texttt{ConTy}\ name\ typs_2) & = & \texttt{let}\ substs = \texttt{zip}\ unify(-,-)\ typs_1\ typs_2\ \texttt{in fold}\ \circ\ \varnothing\ substs \\
unify(l_1 \to r_1, l_2 \to r_2) & = & \texttt{let}\ s_1 = unify(l_1, l_2)\ \texttt{in let}\ s_2 = unify((s_1 r_1), (s_2 r_2))\ \texttt{in}\ s_1 \circ s_2
\end{array}$$

$$\boxed{varBind(-,-)}$$

$$\begin{array}{rcl}
varBind(\tau, \tau) & = & \varnothing \\
varBind(u, \tau) & = & \{u := \tau\},\ \ u \notin FV(\tau)
\end{array}$$

$$\boxed{getTyReps(-)}$$

$$getTyReps((\texttt{K}_i : \forall \overline{k}, \overline{c}, \overline{s}, \overline{b}.\tau_1 \to \cdots \to \tau_p \to T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s})) \quad = \quad \{\texttt{TypeRep}\ a \mid a \in (Fv_k[\![\tau_1 \ldots \tau_p]\!] - Fv[\![\overline{\tau_k}]\!]) - \overline{b}\}$$

$$\boxed{buildTyRep(-)}$$

$$\begin{array}{rcl}
buildTyRep(a) & = & a_{typerep} \\
buildTyRep(\mathbb{T}\ \overline{\tau}) & = & \texttt{let}\ (ts : t : []) = \texttt{map}\ \ buildTyRep(-)\ \overline{\tau}\ \texttt{in}\ (\texttt{typerep}\ \mathbb{T})\ ts\ t \\
buildTyRep(ArrowTy\ \tau_1\ \tau_2) & = & (\texttt{typerep}\ ArrowTy)\ buildTyRep(\tau_1)\ buildTyRep(\tau_2)
\end{array}$$

$$\boxed{bind(-,-,-)}$$

$$\begin{array}{rcl}
bind(\psi, (a : as), e) & = & \texttt{let}\ a_{typerep} = findPath(\psi, a)\ \texttt{in}\ bind(\psi, as),\ \ a \notin \psi \\
bind(\psi, (a : as), e) & = & \texttt{let}\ \tau = \texttt{lookup}\ a\ \psi\ \texttt{in let}\ a_{typerep} = buildTyRep(\tau)\ \texttt{in}\ bind(\psi, as),\ \ a \in \psi \\
bind(\psi, [], e) & = & e
\end{array}$$

$$\boxed{dispatch_{\downarrow}(-,-)}$$

$$\begin{array}{rcl}
dispatch_{\downarrow}(\phi, x, (T\ \overline{\tau})) & = & \texttt{let}\ \overline{v} = \texttt{map}\ (\lambda\ \tau.bind(\langle\!|\phi|\!\rangle, FV(\tau), buildTyRep(\tau)))\ getTyReps(K) \\
 & & \texttt{in}\ \breve{T}\ \overline{v}\ x\ ,\ \ T \text{ has erased variables and } x = K\ x_1 \ldots x_n \\
dispatch_{\downarrow}(\phi, x, (T\ \overline{\tau})) & = & x\ ,\ \ T \text{ doesn't have erased variables} \\
dispatch_{\downarrow}(\phi, x, a) & = & x \\
dispatch_{\downarrow}(\phi, x, ArrowTy\ \tau_1\ \tau_2) & = & x \\
dispatch_{\downarrow}(\phi, x, \texttt{TypeRep}\ \tau) & = & x
\end{array}$$

$$\boxed{(-,-)_{\Downarrow}}$$

$$\begin{array}{rcl}
(((\texttt{K}_j\ \overline{\tau} \to T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}) : ks), T\ k\ c\ s)_{\Downarrow} & = & \breve{T}\ dd\ =\ \texttt{case}\ dd\ \texttt{of} \\
 & & \langle\!|\ \texttt{K}_j\ \overline{a} \to \\
 & & \quad \texttt{let}\ \phi = unify(T\ \overline{k}\ \overline{c}\ \overline{s}, T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s})\ \texttt{in} \\
 & & \quad \texttt{let}\ \overline{b'} = \texttt{map}\ (\lambda\tau.\ bind(\langle\!|\phi|\!\rangle,\ [\tau],\ buildTyRep(\tau)))getTyReps(\texttt{K}_j)\ \texttt{in} \\
 & & \quad \texttt{let}\ \overline{a'} = \texttt{zipWith}\ dispatch_{\downarrow}(\phi, -, -)\ \overline{a}\ \overline{\tau}\ \texttt{in}\ \ K'_j\ \overline{b'}\ \overline{a'}\ \to T'_i\ \overline{\tau_k}\ |\!\rangle
\end{array}$$

Figure 10. Down-conversion and helper functions

$$\boxed{openConstraints(-,-,-,-)}$$

$$\begin{array}{rcl}
openConstraints(\phi, [], \kappa) & = & \kappa\ \phi \\
openConstraints(\phi, (\tau_{typerep}, \tau) : \phi'', \kappa) & = & \texttt{let}\ \phi' = \{\tau := \tau_{typerep}, \phi\}\ \texttt{in} \\
 & & \texttt{case}\ \tau\ \texttt{of} \\
 & & \quad \nu \qquad\qquad \rightarrow \quad \texttt{if}\ \tau \in \phi \\
 & & \qquad\qquad\qquad\quad \texttt{then}\ \texttt{if}\ \tau_{typerep} \simeq_\tau\ \beta_{typerep}\ \ \{\text{where } \beta_{typerep} = lookup\ \tau\ \phi\} \\
 & & \qquad\qquad\qquad\qquad\quad \texttt{then}\ openConstraints(\phi', \phi'', \kappa) \\
 & & \qquad\qquad\qquad\qquad\quad \texttt{else}\ \bot \\
 & & \qquad\qquad\qquad\quad \texttt{else}\ \texttt{let}\ \nu_{typerep} = a \\
 & & \qquad\qquad\qquad\qquad\quad \texttt{in}\ openConstraints(\phi', \phi'', \kappa) \\
 & & \quad (T\tau_1 \ldots \tau_n) \quad \rightarrow \quad \texttt{typecase}\ \tau_{typerep}\ \texttt{of} \\
 & & \qquad\qquad\qquad\quad (\texttt{typerep}\ T)\ a_{1_{typerep}} \ldots a_{n_{typerep}}\ \rightarrow \\
 & & \qquad\qquad\qquad\quad (openConstraints(\\
 & & \qquad\qquad\qquad\qquad \{\tau_1 := a_{1_{typerep}}, \ldots, \tau_n := a_{n_{typerep}}, \phi'\}, \phi'', \kappa))
\end{array}$$

$$\boxed{openFields(-,-,-,-)}$$

$$\begin{array}{rcl}
openFields([], \phi, vars, [], \kappa) & = & \kappa\ \phi\ vars \\
openFields((f : fs), \phi, vars, (\tau : \tau_{rest}), \kappa) & = & \texttt{let}\ a = \texttt{freshvar}\ \texttt{in} \\
 & & \texttt{let}\ a = f\ \texttt{in} \\
 & & openField(\phi, a, \tau, (\lambda\ \phi' v.\ openFields(fs, \phi', (vars, v), \tau_{rest}, \kappa)))
\end{array}$$

$$\boxed{openField(-,-,-,-)}$$

$$\begin{array}{rcl}
openField(\phi, a, (T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}), \kappa) & = & openRecursion(\phi, (T, \overline{\tau_c}), a, \\
 & & \quad (\lambda\ \phi'\ rec.\ unseal(T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}, \phi', rec, \\
 & & \quad\quad (\lambda\ \phi''\ \mathcal{D}_s\ a.\ openConstraints(\phi'', [(s_{typerep}, \tau_s)]), \quad \text{where } s_{typerep} \in \mathcal{D}_s \text{ and } \tau_s \in \overline{\tau_s} \\
 & & \quad\quad (\lambda\ \phi'''.\ (\kappa\ \phi'''\ a)))))) \\
openField(\phi, a, \tau, \kappa) & = & \kappa\ \phi\ a
\end{array}$$

$$\boxed{openRecursion(-,-,-,-)}$$

$$\begin{array}{l}
openRecursion(\phi, (T, (\tau_1, \ldots, \tau_n)), v, \kappa) = \\
\quad \kappa\ \phi\ (\ \widehat{T}\ \widetilde{buildTyRep}(\phi, \tau_1) \ldots \widetilde{buildTyRep}(\phi, \tau_n), v)
\end{array}$$

$$\boxed{\widetilde{buildTyRep}(-,-)}$$

$$\widetilde{buildTyRep}(\phi, \tau) = bind(\langle\!|\phi|\!\rangle, FV(\tau), buildTyRep(\tau))$$

Figure 11. Up-conversion and helper functions

$$\boxed{unseal(-,-,-,-,-)}$$

$$\begin{array}{lcl} unseal(T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}, \phi, rec, \kappa) & = & \kappa\ \phi\ []\ rec \quad \text{when } \overline{\tau_s} = \varnothing \\ unseal(T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}, \phi, rec, \kappa) & = & \texttt{let } f' = \texttt{freshvar in} \\ & & \quad \texttt{case } rec \texttt{ of} \\ & & \quad\quad \mathcal{T}\ \mathcal{D}_s\ f' \rightarrow\ (\kappa\ \phi\ \mathcal{D}_s\ f') \end{array}$$

$$\boxed{(-,-)_{\Uparrow}}$$

$$\begin{array}{lcl} (\phi, T)_{\Uparrow} & = & \widehat{T}\ \mathcal{D}_c\ lower\ = \quad \text{where } K\ ::\ \overline{\tau}^n \rightarrow T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s} \\ & & \quad \texttt{case } lower \texttt{ of} \\ & & \quad\quad (\!|\ K'\ K'_{typerep}\ \overline{p} \rightarrow \quad \texttt{let } typeEnv\ =\ K'_{typerep} \cup \mathcal{D}_c \texttt{ in} \\ & & \quad\quad\quad openConstraints(typeEnv, [(c_{typerep}, \tau_c)], \\ & & \quad\quad\quad\quad (\lambda\ \phi_1\ .\ openFields(\overline{p}, \phi_1, [], \overline{\tau}^n, \\ & & \quad\quad\quad\quad\quad (\lambda\ \phi_2\ \overline{a}^n\ . \\ & & \quad\quad\quad\quad\quad\quad bind(\overline{(\!|\phi_2 \cup unify(s, \tau_s)|\!)}, \overline{s},\ \mathcal{T}\ \overline{buildTyRep(s)}\ (K\ \overline{a}^n)\ ))))\ \ |\!) \end{array}$$

Figure 12. Up-conversion and unsealing functions, continued

doesn't permute its arguments), we have that the type representations expected by K' will be in the same order as the ones provided by $bind$. □

Lemma 2. *If $\mathcal{D}_c$ is well formed and consistent, and if $\phi \supseteq \mathcal{D}_c$ is a well-formed and consistent extension of $\mathcal{D}_c$, then $openConstraints(\phi, D, \kappa)$ will, for each type representation and corresponding to-be-checked type $(c_{typerep}, \tau_c) \in D$ create a substitution to it's corresponding* `TypeRep` *$\{\tau_c := c_{typerep}\}$, and will ensure that these constraints are checked before passing control with these new constraints to κ.*

Proof. We proceed by case analysis and induction on the type τ in the binding, and on the number of substitutions in D for our checked type-variables.

D empty: In this case we simply have no (more) types in checked context to unify with their corresponding `TypeReps`, and thus we have no constraints to check, and no substitutions to make, so we may simply pass control to κ.

$D = \{\tau_{typerep} :: \tau, D'\}$: In this case, we have two cases to analyze (the $ArrowTy$ is exactly the same as the type constructor case).

- Case: $\tau = a$: Then, if $\tau \in \phi$ we execute the following code:

  $$\begin{array}{l} \texttt{if } \tau_{typerep} \simeq_\tau \beta_{typerep} \quad \{\text{where } \beta_{typerep} = lookup\ \tau\ \phi\} \\ \quad \texttt{then } openConstraints(\phi', D', \kappa) \\ \quad \texttt{else } \bot \end{array}$$

  Where $\simeq_\tau$ is a runtime witness of type equality between the two type representations.
  To see that we never reach the undefined case, assume for contradiction that $\tau_{typerep} \not\simeq_\tau \beta_{typerep}$. Then we have by the definitions of $openConstraints$, that $\{\tau := \tau_{typerep}, \phi'\} = \phi$, that $\tau \in \phi$ and that $(lookup\ \tau\ \phi) = \beta_{typerep}$, this implies that we also have the substitution of $\{\tau := \beta_{typerep}\}$ in ϕ. However, since $\tau_{typerep} \not\simeq_\tau \beta_{typerep}$ we must have that $\tau \neq \beta$ which implies that ϕ is inconsistent which violates our hypothesis that ϕ was consistent. So we never reach the undefined case.
  Therefore, since we cannot reach the undefined case with a well-formed and consistent starting ϕ, we will recur under the additional constraint that $\tau_{typerep} \simeq_\tau \beta_{typerep}$ and we add the substitution that τ unifies with $\tau_{typerep}$ to ϕ. We now apply the inductive hypothesis, to get that before we pass control to κ that we have created `TypeRep` witnesses for each new substitution $\tau := \tau_{typerep}$ in ϕ, where each τ corresponds to a monotype.
  Now, if $\tau \notin \phi$ then we execute the following code:

  $$\begin{array}{l} \texttt{let } a_{typerep} = \tau_{typerep} \\ \texttt{in } openConstraints(\phi', D', \kappa) \end{array}$$

  In this case, since τ is a type variable, we simply need to create an alias from the type-variable representation $a_{typerep}$ and the type representation that we already have in hand from ϕ. We then recur under the additional constraint that $\tau := \tau_{typerep}$ and that the type variable that is τ – namely a – can be found as $a_{typerep}$ and corresponds to the type representation for τ that we've already been given. This way we can later on use the type representation for the type variable a and get back the correct type representation. We now apply the inductive hypothesis, to get that before we pass control to κ that we have created `TypeRep` witnesses for each new substitution $\tau := \tau_{typerep}$ in ϕ, where each τ corresponds to a monotype.
- Case: $\tau = T\ \overline{\tau'}$ In this case, we execute the following code:

  $$\begin{array}{l} \texttt{typecase } \tau_{typerep} \texttt{ of} \\ \quad (\texttt{typerep } T)\ a_{1_{typerep}} \ldots a_{n_{typerep}} \ \rightarrow \\ \quad (openConstraints(\\ \quad\quad \{\tau'_1 := a_{1_{typerep}}, \ldots, \tau'_n := a_{n_{typerep}}, \phi'\}, D', \kappa)) \end{array}$$

  We then have by a similar argument as in the `if` case, that this `typecase` must succeed otherwise it would contradict our hypothesis on ϕ.
  Now, by the definition of `typecase`, we have that this pattern match on $(\texttt{typerep } T)$ and the various type representations that it's applied to will ensure (runtime) unification between τ'_i and $a_{i_{typerep}}$ – i.e. point wise type equality ($\simeq_\tau$) between $\tau_{typerep}$ and $a_{i_{typerep}}$ as well as unification of type construc-

tors – and thus, we do not need to add these constraints into D', and instead can add them into our done-constraints ϕ'. We then continue under the additional substitutions.

$$\{\tau_1' := a_{1_{typerep}}, \ldots, \tau_n' := a_{n_{typerep}}\}$$

We thus open up all the needed constraints for $T\ \overline{\tau'}$ since we have opened up all the constraints on $\overline{\tau'}$ and in the process created the needed `TypeRep` witnesses for the substitution $\tau := \tau_{typerep}$.
We now apply the inductive hypothesis, to get that before we pass control to κ that we have created `TypeRep` witnesses for each new substitution $\tau := \tau_{typerep}$ in ϕ, where each τ corresponds to a monotype.

□

Notation: We make use of meta-functions `busted` which given a program returns all the datatypes that are marked as ghostbusted. We define a meta-function `def` which given a type constructor returns the data-definition that corresponds to the type constructor. We also make use of the meta-function `ghostbust` which given a program returns:

$$\begin{aligned}\texttt{ghostbust}(prog) = \quad & (\texttt{def}(S, prog))_{\Downarrow} \cup (\mathcal{D}, \texttt{def}(S, prog))_{\Uparrow} \\ & \cup \texttt{def}(S', prog), \quad S \in \texttt{busted}(prog)\end{aligned}$$

We also define `amb-test` to be a meta-function that ensures that given a program p that all data definitions $dd \in prog$ marked as ghostbusted pass the ambiguity check.

We also define a meta-function `canonicalDict` such that

$$\texttt{canonicalDict}(T, prog) = (\mathcal{D}_s, \mathcal{D}_c)$$

where $\mathcal{D}_s$ and $\mathcal{D}_c$ are the dictionaries for the synthesized and checked types of T respectively.

Throughout we use $\widehat{T}$ to represent the up-conversion function for the datatype with type constructor T and likewise we use $\check{T}$ to represent the down-conversion function for the datatype with type constructor T.

Theorem 4. *For all programs prog, type constructors $T \in prog$, ground types k, c, s, values e, and type representation $\mathcal{D}$, it holds that if*

- $T \in \texttt{busted}(prog)$
- $prog'$ = `ghostbust(`$prog$`)`
- `amb-test(def(`$prog, S$`))` *for all* $S \in \texttt{busted}(prog)$
- $\texttt{canonicalTyRep}(T, prog) = (\mathcal{D}_s, \mathcal{D}_c)$ *and* $\mathcal{D}_s \cup \mathcal{D}_c \sqsubseteq \mathcal{D}$
- $prog \vdash e :: T\ k\ c\ s$

then

1. $prog' \vdash \check{T}\ \mathcal{D}\ e \equiv e'$ *and* $prog' \vdash e' :: T'\ k$
2. $prog' \vdash \widehat{T}\ \mathcal{D}_c\ (\check{T}\ \mathcal{D}\ e) \equiv (\mathcal{T}\ \mathcal{D}_s\ e :: \mathcal{T}\ k\ c)$

Proof. Assume the proviso of the theorem. Since $prog \vdash e :: T\ k\ c\ s$ then there exists a clause

$$K :: \tau_1 \to \tau_2 \to \ldots \to \tau_n \to T\tau_k\tau_c\tau_s \in \texttt{def}(prog, T)$$

and a unification ϕ such that $\phi(T\tau_k\tau_c\tau_s) = T\ k\ c\ s$, $e = (K\ e_1 \ldots\ e_n)$ and also such that $prog \vdash e_i : \phi(\tau_i)$.

Since we have $T \in prog$ and T is annotated as a ghostbusted data definition, then $prog'$ contains a function definition $\check{T} :: TypeRep\ a_1 \to TypeRep\ a_2 \to \ldots TypeRep\ a_m$ where m is the length of c.

We then have by the definition of $\check{T}$, that $\check{T}$ must contain a *unique* pattern that matches this constructor. What's more we have that this pattern-match will be of the following form:

```
K x_1 ... x_n →
   let φ = unify(T k c s, T τ_k τ_c τ_s)
   K'_typereps = map (λ τ →bind((|φ|), [τ], buildTyRep(τ)))
                     getTyReps(K)
   in K' K'_typereps
         dispatch↓(φ, x_1, φ(τ_1))
         ...
         dispatch↓(φ, x_n, φ(τ_n))
```

When calling $\check{T}\ d\ e$, with $e = K\ e_1 \ldots\ e_n$ such pattern-matching clause be instantiated with $x_i = e_i$.

Recalling that the clause is of the form $K :: \tau_1 \to \ldots \to \tau_n \to T\tau_k\tau_c\tau_s$, we proceed by case analysis on the type τ_i of x_i and of the function $dispatch_{\downarrow}$:

- case $\tau_i = (S\ \tau_{k_i}\ \tau_{c_i}\ \tau_{s_i})\{S$ has erased variables$\}$: Now recall that since S is a busted data definition, we have that $\texttt{amb-test}(prog, S)$ holds, and likewise by the proviso of the theorem that there exists a ϕ' such that $prog' \vdash e_i : \phi'(\tau_i)$ and since $\tau_i = S\ \tau_{k_i}\ \tau_{c_i}\ \tau_{s_i}$ we have that $prog' \vdash x_i : \phi'(S\ \tau_{k_i}\ \tau_{c_i}\ \tau_{s_i})$ which implies that $x_i = K_S\ y_1 \ldots y_m$ for some data constructor $K_S \in \texttt{def}(prog, S)$
  We then have by the definition of $dispatch_{\downarrow}$ that $dispatch_{\downarrow}(\phi, x_i, \phi(\tau_i))$ will result in the following code being executed:

  $$\begin{aligned}&\texttt{let}\ \overline{v} = \texttt{map}\ (\lambda\, \tau.bind((\!|\phi|\!), FV(\tau), buildTyRep(\tau))) \\ &\qquad\qquad getTyReps(K_S) \\ &\texttt{in}\ \check{S}\ \overline{v}\ x_i,\end{aligned}$$

  Since we execute the code with the instatiation $x_i = e_i$, we now apply the inductive hypothesis, to get that

  $$prog' \vdash \check{S}\ \overline{v}\ e_i \equiv e_i'$$

  and that

  $$prog' \vdash x_i' :: S'\tau_{k_i} \text{ and } x_i' = K_S'\ K_{S'_{typereps}}\ y_1' \ldots\ y_n'$$

- case $\tau = TypeRep\ \tau'$ By the definition of $dispatch_{\downarrow}$, we have that it will simply return x_i which has type τ_i. Therefore this case is satisfied.
- case $(T\ \overline{\tau})$ $\{T$ has no erased variables$\}$: By the definition of $dispatch_{\downarrow}$, we have that it will simply return x_i which has type τ_i. Therefore this case is satisfied.
- case $(ArrowTy\ \tau_1\tau_2)$: By the definition of $dispatch_{\downarrow}$, we have that it will simply return x_i which has type τ_i. Therefore this case is satisfied.
- case $\tau = a$ is a type variable: By the definition of $dispatch_{\downarrow}$, we have that it will simply return x_i which has type τ_i. Therefore this case is satisfied.

We then have by the definition of LT K' in Definition 1 that for each $x_i :: \tau_i$ that $x_i' :: \tau_i'$ where this τ_i' comes from the fields of K';

$$K' :: K'_{typereps} \to \tau_1' \to \ldots \to \tau_n' \to T'\tau_k$$

So all that's left in order to show that the constructor application is well-typed and produces the right value, is to show that

```
K'_typereps = map (λ τ →bind((|φ|), [τ], buildTyRep(τ)))
                  getTyReps(K)
```

produces the correct type representations of the right type. This is proven in Lemma 1.

We therefore have by the above, and by Lemma 1 that the constructor application of K' to $K'_{typereps}$ will apply K' to all needed type representation fields of K' and that each of these fields will be of the appropriate type and be the correct type representation for the given newly-existential variable(s) in K'. In the case of zero

new existential variables, this map of $bind$ will simply return nothing, since $getTyReps(K) = \{\}$ which agrees with what our LT constructor expects.

We also have by the above that each of the applications of K' to $dispatch_{\downarrow}(\phi, a_i)$ will be well typed, that each of the values will be preserved modulo ghostbusted data types, and that the application of K' to $K'_{typereps}$ and $x'_1 \ldots x'_n$ will result in a total (non-partial) application of K'. From this, since K' is a data-constructor for $T'\ k$ we get that e' has type $T'\ k$. Furthermore, since the pattern matching on e is unique, e' must be of the form $K'\ K'_{typereps}\ e'_1 \ldots\ e'_n$. Where K' is the corresponding LT data constructor for K. Thus (1.) is proved.

In order to prove up-conversion, note that since $T \in prog$, that $\widehat{T} \in prog'$ by our hypothesis. What's more this function $\widehat{T}$ takes $\mathcal{D}_{check}$ which serves as our type representation for each checked type variable.

Now, assume that e' is of the form $K' K'_{typereps}\ e'_1 \ldots\ e'_n$. Then we have by the definition of LT, and by the definition of $\widehat{T}$ that $\widehat{T}$ will have a pattern clause of the form

```
K' K'typereps x'1 ... x'n →
    ...
```

We claim that, the output form of this branch will be of the form $\mathcal{T}\ \mathcal{D}_{synth}\ (K\ x_1 \ldots x_n)$. To see this, we proceed by case-analysis of code-path on the RHS of this pattern match in Figure 11:

- We call $openConstraints(typeEnv, [(c_{typerep}, \tau_c)], \kappa_1)$, where
$$typeEnv = \mathcal{D}_{check} \cup K'_{typereps}$$
we then have by Lemma 2 that before $openConstraints$ relinquishes control to κ_1, that for each checked variable $\tau_c \in \overline{\tau_c}$ we have a checked substitution with its corresponding type representation $\tau_{c_{typerep}}$; $\{\tau_c := \tau_{c_{typerep}}\}$ in ϕ^1 before $(\kappa_1\ \phi^1)$ is called.
- In the above case, we have that
$$\kappa_1 = \quad (\lambda\ \phi_1\ .\ openFields((x'_1, \ldots, x'_n), \phi_1, [], (\tau'_1, \ldots, \tau'_n), \kappa_2))$$
where each x'_i has type τ'_i. We now make the following claims:

Claim 1. *Given a type representation ϕ from $openConstraints$, $openRecursion(\phi, (S, (\tau_1, \ldots, \tau_n)), (J' J'_{typereps}\ e'_1 \ldots e'_m), \kappa)$ will succeed in building the needed type representations for up-conversion of S – where $S\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$ is a type field for a constructor K in $\texttt{def}(prog, T)$ – and hence that the call to $\widehat{S}$ will succeed and return $(\mathcal{S}\ \mathcal{D}_S\ (J\ e_1 \ldots e_m))$ – where $\mathcal{D}_S$ is the set of type representations for the synthesized variables in $(J\ e_1 \ldots e_m)$ – before passing control to κ.*

Proof. This proof hinges on whether or not, for each τ_i we can construct a type representation for τ_i—i.e., that $\widetilde{buildTyRep}(\phi, \tau_i)$ succeeds. Now, we presupposed the ambiguity criteria was fulfilled on T and hence fulfilled on K. Since $(S\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s})$ appears in a field of K, we have that the checked fields $\overline{\tau_c}$ of $(S\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s})$ must be computable from ϕ by our ambiguity criteria—i.e., that $\widetilde{buildTyRep}(\phi, \tau_i)$ succeeds for each τ_i since these τ_i correspond to the $\overline{\tau_c}$ for S.
Now, since we can construct type representations for each τ_i we have by our inductive hypothesis, that since S satisfies our ambiguity criteria, we have been able to compute the canonical type representations for S, and since this field $(J'\ J'_{typereps}\ e'_1 \ldots\ e'_m)$ has come from a down-conversion, we have by our inductive hypothesis, that the call
$$\widehat{S}\ \{\widetilde{buildTyRep}(\phi, \tau_1), \ldots \widetilde{buildTyRep}(\phi, \tau_n)\}\ (J'\ J'_{typereps}\ e'_1 \ldots\ e'_m)$$
will succeed, and will return $(\mathcal{S}\ D_S\ (e_1 \ldots e_m))$ □

Claim 2. *Given a ϕ such that $\phi \supseteq \widetilde{\phi}$ where $\widetilde{\phi}$ comes from $openConstraints$, $openFields(varList, \phi, vars, tyList, \kappa)$ will succeed in up-converting all variables in $varList$ of a busted type to their non-busted types. What's more, for any newly discovered synthesized types in the fields of $varList$ (e.g., within a field whose type is a busted type constructor) it is able to add their corresponding type representations to ϕ before passing control to κ.*

Proof. We proceed by induction on $varList$.
In the case that $varList = []$ we have no variables to up-convert, and thus can simply pass control on to κ.
In the case that $varList = (f : fs)$ and $tyList = (\tau' : \tau_{rest})$, we create a fresh variable (which we assume we can do) and then call $openField$. Thus, if we show that $openField(\phi, a, \tau, \kappa')$ where
$\kappa' = (\lambda\ \phi'\ v.\ openFields(fs, \phi', (vars, v), \tau_{rest}, \kappa))$ succeeds in up-converting the field represented by f, and adds all newly discovered type-information for synthesized variables to ϕ', then the recursive call to $openFields$ within κ' will succeed by the inductive hypothesis.
We thus proceed by case-analysis of $openField$:

Case: $\tau' = \tau$: {where τ is not a busted type} In this case, since τ' was not busted, we have by the definition of down-conversion that τ' in the LT will be the same as in the original type. Thus, we simply just bind the new value to the old value and return this new fresh variable. We then pass control to κ'.

Case: $\tau' = (S\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s})$: {where S is a busted type constructor} In this case, we have by Claim 1 that before we pass control to
$$(\lambda\ \phi'\ rec.\ unseal(T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}, \phi', rec, (\lambda\ \phi''\ \mathcal{D}_s\ a.\ openConstraints(\phi'', [(s_{typerep}, \tau_s)] (\lambda\ \phi'''.\ (\kappa'\ \phi'''\ a))))))$$
that we will be able to open up any new constraints, and that rec will be of the form $(\mathcal{S}\ \mathcal{D}_s\ (e_1 \ldots e_m))$ and that
$$\phi' \supseteq \phi \supset \widetilde{\phi} \implies \phi' \supseteq \widetilde{\phi}$$
by the definition of $openRecursion$. We then have proceeding by case-analysis on $unseal$ that since S is a type constructor (i.e., datatype) that is busted, that we will call the following code
$$(\lambda\ \phi''\ \mathcal{D}_S\ a.\ openConstraints(\phi'', [(s_{typerep}, \tau_s)], (\lambda\ \phi'''.\ (\kappa'\ \phi'''\ a))))$$
where
$$\phi'' = \phi' \supseteq \phi \supseteq \widetilde{\phi} \implies \phi'' \supseteq \widetilde{\phi}$$
We now have by Lemma 2 that the call to $openConstraints$ will create new substitutions $\{\tau_s := s_{typerep}\}$ (and witness them) for each $\tau_s \in \overline{\tau_s}$ and $s_{typerep} \in D_S$. What's more, we have by this same lemma that $openConstraints$ will then pass these new substitutions on to
$$(\lambda\ \phi'''.\ (\kappa'\ \phi'''\ a))$$
But this κ' is just
$$(\lambda\ \phi'\ v.\ openFields(fs, \phi', (vars, v), \tau_{rest}, \kappa))$$
and we can now apply the inductive hypothesis to the inner call of $openFields$ after β-reduction.
We have thus shown that $openField$ succeeds in up-converting the type field represented for f – renaming it a fresh name v –

and also adds all newly-discovered (synthesized) type information from τ' to ϕ''' and therefore by the inductive hypothesis, that $openFields(fs, \phi''', (vars, v), \tau_{rest}, \kappa)$ succeeds within κ'.
Therefore $openFields$ will succeed in up-converting all fields in $varList$ and, for all newly-discovered synthesized type-variables in these fields, will witness and add a substitution between the synthesized type-variable and it's corresponding type representations to ϕ before passing control to κ. □

We then have by Claim 2 that each individual field $x'_i :: \tau'_i$ will be converted to a field x_i of type τ_i where τ_i is either τ'_i in the case that τ'_i wasn't a busted type, or $T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$ in the case that $\tau'_i = T'\ \overline{\tau_k}$. We also have by the lemma, that for all newly discovered synthesized type variables τ_{i_s} for τ_i, that we will be able to create a type representation for them and that these substitutions $\{\tau_{i_s} := \tau_{i_{s-typerep}}\}$ will be added to ϕ^2 before being passed to κ_2; $(\kappa_2\ \phi^2\ (x_1, \ldots, x_n))$.

- In the above case, we have that

$$\begin{aligned}\kappa_2 = \quad & (\lambda\ \phi_2\ \overline{a}^n. \\ & \quad bind((\!|\phi_2 \cup unify(\overline{s}, \overline{\tau_s})|\!|), \overline{s}, \\ & \quad\quad \mathcal{T}\ \overline{buildTyRep(s)}\ (K\ \overline{a}^n)))\end{aligned}$$

We now have by the ambiguity criteria—and in particular that synthesized types are computable from the fields of τ'_i—and since we have taken $\psi = (\!|\phi_2 \cup unify(\overline{s}, \overline{\tau_s})|\!|$ in our call to $bind$, that we will only have the following two cases to consider. We proceed by induction on the size of $\overline{s}$

  - $\overline{s'} = (x : xs)$: We have by the way we have constructed ψ that $x \in \psi$. We therefore have that we execute the following code:

$$\begin{aligned}&\texttt{let}\ \tau = \texttt{lookup}\ x\ \psi\ \texttt{in} \\ &\texttt{let}\ x_{typerep} = buildTyRep(\tau) \\ &\quad \texttt{in}\ bind(\psi, xs)\end{aligned}$$

    The τ that is returned from the lookup will be either a kept, checked, or synthesized type. If τ is kept or checked, then we have by the above parts (specifically, bullet one and two of this part of the proof) that we will have a type representation witness for τ. Likewise if τ is a synthesized type, then we have that the ambiguity criteria will ensure that we are able to construct a type representation witness for τ as well. We thus have that we can always construct the type representation needed, and that we have the appropriate substitutions and constraints within scope to ensure that these type representations are closed. We then have by the inductive hypothesis that we can construct type representations for xs.
  - $\overline{s'} = []$: In this case we have no more synthesized variables to find type representations for, we return

$$\mathcal{T}\ \overline{buildTyRep(s)}\ (K\ \overline{a}^n)$$

    and at this point either s was initially empty and we don't do anything, or it has some synthesized variables in it. In the latter case we have by the ambiguity criteria, by the definition of $bind$, and the way we constructed the ψ that we called $bind$ with that for each $s \in \overline{s}$ that s's type representations will be in scope. (i.e., we will have created a binding from $s_{typerep}$ to its corresponding type representation in a `let` above this whose scope extends over this expression.)

Thus, we see that when we construct

$$\mathcal{T}\ \overline{buildTyRep(s)}\ (K\ \overline{a}^n)$$

that for each $s \in \overline{s}$ that $s_{typerep}$ is in scope, and that each of these is computable from ϕ_2 by our ambiguity criteria. We therefore have that the call to

$$\mathcal{T}\ \overline{buildTyRep(s)}\ (K\ \overline{a}^n)$$

will succeed. Now recall that $\mathcal{D}_s$ is just notational short-hand for $\overline{buildTyRep(s)}$.

Thus we see that

$$\begin{aligned}\widehat{T}\ \mathcal{D}_c\ e' &= \widehat{T}\ \mathcal{D}_c(K'\ K'_{typereps}\ x'_1 \ldots x'_n) \\ &= \mathcal{T}\ \mathcal{D}_s\ (K\ x_1 \ldots\ x_n)\end{aligned}$$

This proves (2.) □